Università degli Studi di Milano

Doctor of Philosophy in Computer Science

Cycle XXXV

Department of Computer Science "Giovanni degli Antoni"

Scalable Graph Representational Learning Algorithms for Network Medicine

R18

Luca Cappelletti

Supervisor: Prof. Giorgio Valentini

Co-supervisor: Prof. Elena Casiraghi

Course coordinator: Prof. Roberto Sassi

A. A. 2021-2022

Contents

Abstract

In the era of big data, advanced computational techniques are needed to process, analyze and visualize increasing amounts of data generated by high-throughput technologies. In this context, analyzing biomedical Knowledge Graphs that embrace biological and medical concepts structured in ontologies and data generated from high-throughput bio-technologies represents a central Machine Learning and Computational Biology challenge.

Indeed several compelling problems in Network Medicine, ranging from gene-disease prioritization to drug-target prediction or drug repurposing, can be modelled as node label or edge prediction problems in graphs, where nodes represent bio-medical entities as genes, drugs or diseases and edges interactions or relationships between them. Recently Graph Representation Learning (GRL) methods opened new possibilities for addressing complex, real-world problems represented by graphs. However, many graphs used in these applications comprise millions of nodes and billions of edges and are beyond the capabilities of current methods and software implementations.

To deal with this open problem, the first contribution of this thesis is the design and development of the *GRAPE* (Graph Processing and Embedding) resource for GRL, able to scale with big graphs thanks to specialized and innovative data structures and algorithms, efficiently implemented through parallel computation. Compared with state-of-the-art software resources, *GRAPE* shows an improvement of orders of magnitude in empirical space and time complexity, as well as a substantial and statistically significant improvement in edge prediction and node label prediction performance. *GRAPE* provides over $80,000$ graphs from the literature and other sources, standardized interfaces allowing a straightforward integration of third-party libraries, 61 node embedding methods, 25 inference models, and 3 modular pipelines to allow a findable, accessible, interoperable, and reusable (FAIR) and reproducible comparison of methods and libraries for graph processing and embedding. *GRAPE* can quickly generate billions of sampled random walks for random-walk-based GRL algorithms, such as DeepWalk of Node2vec, thus leading to very accurate embedded graphs and boosting machine learning methods' performance that learns from the embedded vector representation of nodes and edges. The scaling properties of *GRAPE* with respect to state-of-the-art resources have been analyzed through extensive experiments with real-world big graphs, including, e.g. Wikipedia, the Comparative Toxicogenomic Database (CTD) and a big biomedical Knowledge Graph generated by PheKnowLator. *GRAPE* significantly outperforms state-of-the-art libraries in terms of empirical time and space complexity, edge prediction performance, and can process big graphs even when the other competing state-of-the-art resources fail.

As a second contribution for efficiently processing and analyzing big graphs, this thesis proposes a novel algorithmic framework, efficiently implemented in GRAPE, that

we named ALPINE: Abstract Landmark Properties-Inferred Node Embedding. The breakthrough characteristics of this algorithmic framework allow us to deal with several issues affecting SOTA GRL methods:

1. The embedding features are independently computed, each from the others, thus overcoming the space and time complexity limitations due to their dependent computation.
2. Feature computation is based on the "landmarks", i.e. sets of nodes representing meaningful concepts about the structure or the semantics of the underlying graph, thus assuring the interpretability of the embeddings.
3. Small integers are used for embedded features values: this assures a small memory footprint, hardware acceleration, and a good compression ratio because of the scale-free distribution of node degrees that often characterize biomedical networks
4. "Democratic" feature representation, thus avoiding the bias towards high degree nodes characteristic of embedding methods based on topological sampling.

We present two algorithms based on the ALPINE framework: a) SPINE (Shortest Paths Inferred Node Embedding), based on the efficient computation of the shortest path distance from the landmarks; b) WINE (Windows Inferred Node Embedding) based on the efficient computation of the co-occurrences of each node with the landmarks within windows of a given size during a breadth-first search. The breakthrough scaling properties of the proposed algorithms are shown in experiments with real-world big graphs.

GRAPE and ALPINE implementations are available from `https://github.com/AnacletoLAB/grape`.

The thesis is organized as follows. In chapter 2, the architecture of *GRAPE*, the graph-processing, and graph-representation learning algorithms it provides are described. In chapter 3, we sketch both the open libraries for graph processing and the datasets that have been used to perform the experiments for comparing *GRAPE* to state-of-the-art works. In chapter 4, all the experimental settings and all the achieved comparative evaluation results are described in detail. In chapter 5, we focus on the novel *ALPINE* algorithmic framework and compare its performance with those of to state-of-the-art models. The conclusions summarize the main contributions of the thesis, as well as the drawbacks and limitations of the proposed methods and depict ongoing and future work on scalable GRL methods and their application to Network Medicine.

Introduction

In fields such as biology, medicine, data and network science, graphs can naturally model available knowledge as interrelated concepts, represented by a network of (possibly attributed) nodes connected by edges. The wide range of graph applications has motivated the development of a rich literature on Graph Representation Learning (GRL) and inference models. GRL models compute embeddings, i.e. vector representations (usually in a metric space) of the graph and its constituent elements, capturing their topological, structural, and semantic relationships. Graph inference models can use such embeddings and available additional features for several tasks, e.g., visualization, clustering, node-label, edge-label, and edge prediction problems [57, 77].

GRL methods available in the literature include, among others, matrix factorization-based methods (section 2.3.1), random walk-based methods (section 2.3.2) and triple-sampling methods (section 2.4) [57]. They have shown their effectiveness in analyzing networks from sociology, biology, medicine, and many other disciplines [77].

Although a great deal of research has been devoted to the development of libraries to process and analyze graphs (e.g., iGraph [32], GraphLab [74], NetworkX [55], GraphX [50] and SNAP [70]), as well as GRL software resources (e.g., PecanPy [72], PyKeen [3], DGL [127], Pytorch Geometric [43], Spektral [51]), real-world networks often include millions of nodes and billions of edges, thus raising the problem of the scalability of existing software resources implementing Graph Representation Learning algorithms [77, 131].

One of the limitations of current state-of-the-art random walk-based graph embedding algorithms is the inability to generate enough data to accurately represent the topology of the underlying graph. This can be a significant issue, as the performance of node and edge label prediction methods heavily depends on the informativeness of the embedded graph representation.

ìTo address this limitation, it is necessary to develop methods for efficiently generating a large number of random walks from the graph. This will allow for the creation of more accurate embedded representations of the graph, which can in turn improve the performance of machine learning methods that rely on these representations for node and edge label prediction tasks.

By generating billions of sampled random walks, we can construct more comprehensive and informative representations of the underlying graph. This can be beneficial for a wide range of downstream tasks, such as node classification, link prediction, and community detection. Overall, the efficient generation of large amounts of data is crucial for improving the performance of graph embedding algorithms and the machine learning methods that depend on these representations.

One of the important challenges in the field of graph machine learning is the lack of tools for fairly and reproducibility comparing different methods under different experimental conditions. This issue is often not addressed by current state-of-the-art libraries, making it difficult to accurately evaluate the performance of different algorithms and libraries for graph-based data analysis.

To address this issue, it is necessary to develop standardized interfaces that allow for the easy integration of methods from different libraries, as well as standardized experimental pipelines that can be used to evaluate the performance of these methods in a consistent and reproducible manner. This will enable researchers to easily compare different algorithms and libraries, and provide a more accurate and comprehensive understanding of the strengths and limitations of each approach.

The ability to easily and fairly compare different graph-based methods is crucial for the advancement of the field of graph machine learning. Standardized interfaces and experimental pipelines can facilitate this comparison, making it more accessible and enabling researchers to make informed decisions about which methods and libraries to use for different tasks and datasets.

In this thesis, we present the *GRAPE* library for working with graph representation learning, which involves learning a low-dimensional representation of nodes in a graph that captures the underlying structure of the graph. This can be used for a variety of tasks, such as predicting the labels of nodes or edges in the graph, or for unsupervised analysis of the graph structure.

Ensmallen and *Embiggen* are two core modules within the *GRAPE* library. *Ensmallen* is designed to efficiently load and process large graphs, using efficient data structures and parallel computation to scale to graphs with billions of nodes and edges. This module also provides a range of graph processing methods, such as algorithms for computing random and minimum spanning arborescence, and methods for node and edge filtering. *Ensmallen* allows graphs to be loaded from a variety of formats, and includes a large collection of pre-loaded graphs from the literature and elsewhere.

Embiggen, on the other hand, focuses on generating node embeddings from processed graphs. Node embedding is the process of learning a low-dimensional representation of nodes in a graph, and *Embiggen* implements exact and approximated versions of popular node embedding algorithms, such as DeepWalk, Walklets, and Node2Vec. In particular, the approximated versions of the algorithms are able to handle graphs with high-degree nodes, which are otherwise difficult to work with using the exact versions of these algorithms. The resulting node embeddings can be used for tasks such as graph visualization or prediction tasks, such as predicting node labels or edge labels.

Together, *Ensmallen* and *Embiggen* provide a powerful set of tools for working with large graphs and generating high-quality node embeddings. The emphasis on scalability and performance makes the *GRAPE* library well-suited for working with even very large graphs on commodity hardware.

One of the benefits of using the *GRAPE* library for graph machine learning is that it facilitates the fair and reproducible comparison of different methods under different experimental conditions. This is possible through the use of standardized interfaces that allow for the easy integration of methods from different libraries, as well as standardized experimental pipelines that can be used to evaluate the performance of these methods in a consistent and reproducible manner.

The use of the *GRAPE* library can greatly enhance the ability of researchers to compare different graph-based methods and make informed decisions about which algorithms and libraries to use for different tasks and datasets. This can be a valuable tool for advancing the field of graph machine learning and improving the performance of these methods on real-world data.

GRAPE provides re-implementations of many graph representation learning methods, and it is significantly faster than the implementations available from the state of the art, yet this is not enough to run embedding of billion-scale graphs. As last result of this thesis, we will introduce ALPINE. ALPINE is a framework for extremely scalable graph representation learning built on top of *GRAPE* that addresses these limitations and improves upon previous methods, such as DeepWalk. ALPINE improves upon these methods by addressing limitations in their applicability to real-world graphs with many nodes and edges. By overcoming these limitations, ALPINE enables the use of graph representation learning in a wider range of applications, and can embed graphs with billions of nodes and edges, such as the internet.

One of the core ideas of the ALPINE framework is the novel concept of abstract landmarks and their properties, which we use to ensure feature interpretability by design in ALPINE. A landmark is a group of nodes in a graph that share a distinctive trait or characteristic, such as similar node degree or common labels. This group of nodes can be thought of as a single, abstract node that represents one of the graph's key ideas or central themes. We stress that the interpretability present in ALPINE embedding is not inherent in ALPINE itself nor an emergent property, but derives from the selection of meaningful landmarks and features with a clear interpretable meaning.

In the first chapter of the thesis, we will introduce the *GRAPE* library, which provides highly efficient and scalable implementations of popular graph representation learning methods. We will describe the design and implementation of *GRAPE*, and present a number of experiments that demonstrate its performance and capabilities.

In the subsequent chapter, we will introduce the ALPINE framework, which allows for the execution of node embedding tasks on graphs with billions of nodes. ALPINE addresses the limitations of existing methods, such as gradient descent-based models and matrix factorization techniques.

The work presented in this thesis aims to improve the scalability and performance of graph representation learning methods, and to enable the analysis of large-scale, real-world graphs in a variety of applications.

Chapter 1

State of the art

In this section, we provide an overview of the state of the art in graph representation learning, focusing on the most relevant approaches and their limitations. We begin by describing graphs, ontologies, and knowledge graphs, the building blocks of graph representation learning tasks, and their role in various fields such as natural language processing, social network analysis, and computational biology. We then introduce the concept of graph embedding, which aims to learn low-dimensional representations of nodes in a graph that capture the underlying structure and semantics of the graph. Finally, we discuss the various algorithms and software libraries that have been developed for graph representation learning and their limitations in terms of scalability and performance.

1.1 Graphs, ontologies and knowledge graphs

A graph is a mathematical structure used to represent relationships between objects. It consists of a set of vertices V, which represent the objects, and a set of edges E, which represent the relationships between the objects.

Graphs are widely used in many different fields, including computer science, engineering, and social sciences. In the field of machine learning, graphs are often used to represent data that has a natural structure or hierarchy, such as networks of interconnected nodes or relationships between entities.

One of the key applications of graph machine learning is in the analysis of complex networks, such as social networks or the internet. These networks are often too large and complex to be understood using traditional methods, so machine learning algorithms can be used to analyze and understand the structure and dynamics of the network.

Another important application of graph machine learning is in the analysis of structured data, such as customer transactions or interactions between proteins in a biological system. In these cases, the graph structure can be used to encode the relationships between the data points, allowing machine learning algorithms to learn and make predictions about the data.

In addition to the applications mentioned above, graphs are also widely used in the field of network medicine and biology. In these fields, graphs are often used to represent networks of biological entities, such as proteins, genes, or cells.

In network medicine, graphs can be used to represent the interactions between different proteins or genes in a biological system, such as a cell or an organism. By analyzing the structure of these networks, researchers can gain insights into the functions of the proteins and genes, and how they contribute to the overall functioning of the system.

In biology, graphs are also used to represent the relationships between different species in an ecosystem. By analyzing the structure of these ecological networks, researchers can gain insights into the interactions between different species and how they affect each other's populations and habitats.

An ontology is a formal representation of a set of concepts within a domain, and the relationships between those concepts. It can be used to represent the shared understanding of a domain that is agreed upon by a group of people, and can be used to reason about the objects, concepts, and relationships within that domain [54, 53, 87].

A knowledge graph is a graphical representation of knowledge that is structured in the form of interconnected nodes and edges. The nodes represent entities, such as people, places, or things, and the edges represent relationships between those entities. A knowledge graph can be used to represent a wide range of knowledge, including information about people, places, events, and concepts. A knowledge graph is a powerful tool for organizing and making sense of large amounts of information, and can be used to answer complex queries, provide recommendations, and support various types of decision-making tasks [6].

Ontologies and knowledge graphs are both representations of knowledge that can be used to organize and make sense of large amounts of information. However, there are some key differences between the two.

One key difference is the way in which the knowledge is represented. Ontologies are typically represented using a formal language, such as the Web Ontology Language (OWL), and are used to define the structure and meaning of the concepts within a domain. Knowledge graphs, on the other hand, are graphical representations of knowledge that are structured in the form of interconnected nodes and edges. The nodes represent entities, such as people, places, or things, and the edges represent relationships between those entities. Another key difference is the purpose for which the knowledge representation is used. Ontologies are often used to represent the shared understanding of a domain that is agreed upon by a group of people, and can be used to reason about the objects, concepts, and relationships within that domain. Knowledge graphs, on the other hand, are often used to support complex queries, provide recommendations, and support various types of decision-making tasks [87].

In the field of network medicine and biology, ontologies and knowledge graphs can be used to represent and organize the complex relationships between different biological entities, such as genes, proteins, diseases, and pathways. This can facilitate the integration and analysis of large-scale biological data and support the development of new computational tools and approaches for studying the mechanisms of disease and the effects of treatments.

For example, the Gene Ontology (GO) [29] is a widely used ontology for representing knowledge about the functions and relationships of genes in different organisms. The GO includes a hierarchy of terms that describe the different aspects of gene function, such as cellular component, molecular function, and biological process. This allows researchers to annotate and classify genes based on their known or predicted functions, and to analyze the data in a consistent and standardized way.

Another example is the Human Phenotype Ontology (HPO) [103], which is an ontology for representing knowledge about human diseases and phenotypic abnormalities. The HPO includes a hierarchy of terms that describe different aspects of disease, such as symptoms, signs, and diagnostic findings. This allows researchers to annotate and classify diseases based on their phenotypic characteristics, and to analyze the data in a consistent and standardized way.

Examples of large-scale knowledge graphs include Google Knowledge Graph and WikiData. These knowledge graphs require scalable implementations of software to process and manage the large amounts of information they contain. For example, Google Knowledge Graph uses machine learning algorithms and natural language processing techniques to extract and organize information from the web and to generate answers to user queries in a way that is easy for people to understand.

In the field of network medicine and biology, large-scale knowledge graphs can provide a rich and comprehensive source of information about genes, proteins, diseases, and other biological entities. This can support the development of new computational tools and approaches for studying the mechanisms of disease and the effects of treatments, and can facilitate the integration and analysis of large-scale biological data.

For example, researchers could use a large-scale knowledge graph to identify connections between different genes and proteins, and to explore the relationships between different diseases and their underlying mechanisms. This could help to identify potential targets for new therapies and to design more effective treatments for a variety of diseases.

Large-scale knowledge graphs are important resources for representing and organizing knowledge and information on a broad range of topics. In the field of network medicine and biology, they can provide a rich and comprehensive source of information that motivates the development of new computational tools and approaches for studying the mechanisms of disease and the effects of treatments.

1.2 Graph representation learning

In recent years, the task of learning meaningful representations of the nodes and edges in a graph has garnered significant attention, as it allows for the application of machine learning techniques to graph structured data. This field, known as Graph Representation Learning (GRL), has a wide range of applications, including node classification, link prediction, and graph classification. In this subsection, we will review the state of the art in GRL and we will then delve into the various methods that have been proposed for learning graph representations, including matrix factorization, random walk based approaches, and neural network based approaches [57, 77].

1.2.1 Random walk-based models

Random-walk based models for node embedding are a class of algorithms that are used to represent the nodes in a network in a low-dimensional matrix, i.e. the embedding. An embedding matrix should capture the structural relationships between the nodes in the network, such as the patterns of connectivity and the strength of the connections. Moreover, the embeddings should capture the overall structure of the network, rather than just the local structure around individual nodes. Random-walk based node embedding models are important because they allow us to analyze and understand the structure of complex networks. For example, they can be used to identify groups of nodes that are densely connected, or to find nodes that are similar to each other in terms of their connections. Additionally, random-walk based models are often used as a preprocessing step for other machine learning algorithms, as they can provide a more structured representation of the data. In summary, these models are useful for analyzing and understanding complex networks, and are frequently used in a variety of applications.

Word2Vec

Word2Vec is a widely used model for learning distributed word representations [79]. It is commonly used in natural language processing tasks and has been extensively employed in network representation learning algorithms such as DeepWalk, Walklets, and Node2Vec, which we will describe shortly in the following sections. These algorithms rely on the ability of Word2Vec to capture the semantic relationships between words, which allows them to learn latent representations of nodes in a network. We will focus on the use of Word2Vec in network representation learning and discuss its effectiveness in this context.

Word2Vec is a method for representing words in a way that captures their meanings and relationships with other words. It does this by creating dense, numerical vectors (or "embeddings") for each word. These vectors capture the meanings of words in a way that can be used in natural language processing tasks such as translation, text classification, and text similarity comparison.

In Word2Vec, words that have similar meanings or are often used in similar contexts are represented by vectors that are close together in vector space. This allows the model to capture the relationships between words and use these relationships to make predictions about the meanings of new words that it encounters.

Word2Vec is trained on large amounts of text data and uses a neural network to learn the vector representations of words. The training process involves predicting the words that are likely to appear in a context surrounding a given word, based on the vector representations of those words. This allows the model to learn the relationships between words and improve its predictions over time.

DeepWalk

The DeepWalk model is a graph representation learning algorithm that was introduced in 2014 by Perozzi et al. [94]. It is based on the concept of random walks, which are sequences of nodes in a graph that are generated by randomly transitioning from a current node to one of its neighboring nodes. These random walks are used to generate node sequences, which are then treated as sentences in natural language processing. A Word2Vec model is trained on these node sequences, which results in a low-dimensional vector representation for each node in the graph. This vector representation, also known as a node embedding, captures the local and global structure of the graph.

One of the key advantages of the DeepWalk model is that it is able to capture the local structure of the graph, which is important for tasks such as node classification and link prediction. This is because the random walks allow the model to explore the neighborhood of each node, which provides information about the connections and relations between nodes. In contrast, traditional graph representation learning methods such as matrix factorization only capture the global structure of the graph, which can be less effective for these tasks.

The DeepWalk model has proven to be a successful approach for learning node representations in large networks. However, there are some limitations to this model, such as the inability to model the hierarchy existing in nodes neighbours, that can be addressed by the introduction of the Walklets model.

Walklets

The Walklets model [96] is a variant of the DeepWalk model that uses a multi-scale approach, where multiple random walks of varying lengths are generated from each node in the network. This allows for a more comprehensive representation of the network, capturing both local and global structure. Additionally, the Walklets model uses a hierarchical sampling strategy, which often results in more improved performance on downstream tasks.

More specifically, to generate node sequences, the Walklets model first samples a set of starting nodes from the graph. For each starting node, the model performs a random walk on the graph, sub-sampling the nodes by skipping a provided amount of nodes for each sampled node. This process generates a hierarchical sequence of nodes, which is then used to learn a set of latent features that capture the underlying structure of the graph.

Node2Vec

Node2Vec [52] is yet another model for learning low-dimensional representations of nodes in a graph. Both DeepWalk and Walklets are similar to the Node2Vec model in that they all use random walks to learn node representations. Node2Vec uses a biased sampling strategy that allows for the control of the trade-off between local and global structure in the learned representations.

The bias in the random walk is controlled by two parameters, p and q, which determine the likelihood of the walk returning to the current node or moving to a node that is farther away in the graph.

The relationship between the p and q parameters and the graph's triangles (i.e. sets of three nodes connected by three edges) is an important factor in the performance of Node2Vec. In general, a smaller value of p and a larger value of q will lead to more exploration of the graph's triangles, resulting in embeddings that are more sensitive to the local structure of the graph. Conversely, a larger value of p and a smaller value of q will lead to less exploration of triangles, resulting in embeddings that are more sensitive to the global structure of the graph.

1.2.2 Spectral and matrix factorization methods

Spectral and matrix factorization methods for node embedding start by computing weighted adjacency matrices, which encode information about the connectivity and similarity between pairs of nodes in the graph. These matrices are, most commonly, extremely sparse. These methods may include one or more factorization steps, in which the adjacency matrix is factorized into several matrices that capture different aspects of the graph's structure.

Given a target embedding dimensionality, these methods generally use as node embeddings the eigenvectors or singular vectors corresponding to spectral or singular values of interest. For example, spectral methods such as Laplacian Eigenmap (LE) and Geometric Laplacian Eigenmap Embedding (GLEE) use the eigenvectors corresponding to the smallest or largest eigenvalues of the graph's Laplacian matrix as node embeddings. Matrix factorization methods such as High-Order Proximity preserved Embedding (HOPE) and Social Dimensions (SocioDim) use the singular vectors corresponding to the most significant singular values of a proximity matrix or modularity matrix as node embeddings.

In this section, we will describe and compare several spectral and matrix factorization node embedding methods, including LE, GLEE, HOPE, SocioDim, Alternating Direction Method of Multipliers for Non-Negative Matrix Factorization (NMFADMM), Iterative Random Projection Network Embedding (RandNE), Graph Representations (GraRep), and Network Matrix Factorization (NetMF). We will discuss how each method computes weighted adjacency matrices and performs factorization steps, and how they use spectral or singular vectors to compute node embeddings. We will also discuss the strengths and weaknesses of each method, and compare their performance on various graph datasets. All of these methods are available in *GRAPE*, both as new implementations from scratch, and also as integrated implementations from third party libraries to facilitate comparisons.

Laplacian matrices

Since many of the following methods are based on Laplacian matrices derived from the graph adjacency matrix, we define first what a Laplacian matrix is.

The Laplacian matrix of a graph is a square matrix that encodes the connectivity of the graph. It is defined as the difference between the degree matrix and the adjacency matrix of the graph. The degree matrix is a diagonal matrix that contains the degree of each vertex in the graph, and the adjacency matrix is a matrix that contains the edge weights between all pairs of vertices in the graph, or a one in the case of an unweighted graph.

The symmetrically normalized Laplacian matrix is a variant of the Laplacian matrix that is often used in spectral graph theory. It is defined as the inverse square root of the degree matrix times the Laplacian matrix. This normalization helps to ensure that the eigenvalues of the matrix lie in the range $[-1, 1]$, which can be useful for some applications, including data clustering, network analysis, and machine learning.

The symmetrically normalized Laplacian has several important properties, including being positive semi-definite and having the smallest possible number of zero eigenvalues among all possible Laplacian matrices for a given graph. This makes it a useful tool for studying the structure of a graph and for solving various problems in graph theory [27].

Laplacian Eigenmap

Laplacian Eigenmap (LE) [11] computes the symmetrically normalized Laplacian of the graph, which is a matrix that encodes information about the connections between nodes in the graph. Next, LE computes the eigenvectors of the Laplacian matrix that correspond to the smallest eigenvalues. These eigenvectors are then used as the embeddings of the nodes in the graph.

One potential strength of Laplacian Eigenmap (LE) is that it uses the eigenvectors corresponding to the smallest eigenvalues, which are often believed to capture the most important structural information about the graph. This can lead to LE producing node embeddings that accurately reflect the underlying structure of the graph, with direct applications in protein structure prediction [28].

However, a potential weakness of LE is that the choice of which eigenvectors to use as node embeddings is somewhat arbitrary. In particular, there is no theoretical guarantee that the eigenvectors corresponding to the smallest eigenvalues will always produce the most accurate node embeddings.

Geometric Laplacian Eigenmap Embedding

Geometric Laplacian Eigenmap Embedding (GLEE) [122] is a variation of LE that builds on the same principles. Like LE, GLEE computes the symmetrically normalized Laplacian of the graph and uses this to compute eigenvectors. However, instead of using the eigenvectors corresponding to the smallest eigenvalues, GLEE uses the eigenvectors corresponding to the largest eigenvalues. This difference in the eigenvectors used leads to GLEE producing node embeddings with different properties than those obtained by using LE.

Geometric Laplacian Eigenmap Embedding (GLEE) can potentially overcome this weakness by using the eigenvectors corresponding to the largest eigenvalues. This choice of eigenvectors is less arbitrary, as it is based on the idea that the largest eigenvalues are the most "spread out" in the graph, and therefore capture the most global information about the graph. However, a potential weakness of GLEE is that it may not always produce the most accurate node embeddings, as the eigenvectors corresponding to the largest eigenvalues may not always capture the most important structural information about the graph.

High-Order Proximity preserved Embedding

High-Order Proximity preserved Embedding (HOPE) [90] starts by computing a node-proximity matrix, which encodes information about the proximity or similarity between pairs of nodes in the graph. The proximity between two nodes can be defined in different ways, such as by using the number of common neighbors or the Adamic-Adar index.

Next, HOPE computes the singular vectors of the proximity matrix that correspond to the most significant singular values. These singular vectors are then used as the embeddings of the nodes in the graph. In particular, the left and right product of the singular values with the singular vectors are used as the embeddings of the source and destination nodes, respectively.

HOPE differs from LE and GLEE in that it uses a node-proximity matrix to compute node embeddings, rather than the Laplacian matrix. This means that HOPE can potentially produce more accurate node embeddings by using more detailed and task-related information about the proximity between nodes in the graph. On the other hand, the choice of which proximity measure to use in the node-proximity matrix is somewhat arbitrary, which could potentially limit the accuracy of HOPE's node embeddings.

Social Dimensions

The Social Dimensions (SocioDim) [120] Like High-Order Proximity preserved Embedding (HOPE), SocioDim computes a node-proximity matrix that encodes information about the similarity between pairs of nodes in the graph. However, instead of using an arbitrary proximity measure, SocioDim uses the modularity matrix, which is a dense matrix that encodes information about the modular structure of the graph.

Next, SocioDim computes the eigenvectors of the modularity matrix that correspond to the largest eigenvalues. These eigenvectors are then used as the node embeddings.

One potential advantage of SocioDim over LE and GLEE is that it uses the modularity matrix, which encodes information about the modular structure of the graph, rather than the Laplacian matrix, which encodes information about the connectivity of the graph. This could potentially lead to SocioDim producing more accurate node embeddings than LE and GLEE. On the other hand, the use of the modularity matrix is somewhat arbitrary, and there is no guarantee that it will always produce the most accurate node embeddings.

Alternating Direction Method of Multipliers for Non-Negative Matrix Factorization

Alternating Direction Method of Multipliers for Non-Negative Matrix Factorization (NMFADMM) [116] is yet another method for mapping nodes in a graph to a low-dimensional space in a way that preserves the distances between nodes. Unlike the other methods discussed so far, NMFADMM uses non-negative matrix factorization (NMF) to compute node embeddings.

Non-negative matrix factorization [67] is a technique in linear algebra that allows a given matrix to be approximated as the product of two non-negative matrices. This factorization has a number of useful properties and applications, particularly in the fields of data mining and machine learning.

NMF is often used as a dimensionality reduction technique, as it can decompose a high-dimensional matrix into two matrices with fewer dimensions, while preserving the non-negative property of the original matrix. This can be useful for tasks such as clustering and feature extraction, as it allows the original data to be represented in a lower-dimensional space.

NMF has also been applied to a variety of other problems, including collaborative filtering, document clustering, and data visualization. It has been shown to be effective in a range of applications, and has become a popular tool in the field of data mining and machine learning.

NMFADMM first computes the left Laplacian matrix of the graph, which encodes information about the connectivity of the graph. Next, NMFADMM uses NMF to factorize the left Laplacian matrix into two non-negative matrices, which correspond to the embeddings of the source and destination nodes, respectively.

Iterative Random Projection Network Embedding

Similar to NMFADMM, Iterative Random Projection Network Embedding (RandNE) [133] applies an iterative procedure to factorize the dot product of the left Laplacian and an (initially) random matrix; after a user defined number of factorization the matrix is used as the node embeddings.

Graph Representations

Graph Representations (GraRep) [24] analogously factorizes the left Laplacian matrix and, at every iteration, computes the singular vectors corresponding to the k most significant singular values, hence producing several embeddings equal to the number of iterations.

One potential advantage of GraRep over the other methods is that it produces several sets of node embeddings, which can potentially improve the accuracy of the embeddings by averaging over multiple sets of embeddings.

Network Matrix Factorization

The Network Matrix Factorization (NetMF), given a window size, first computes a sparse log co-occurrence matrix by using first-order random walks and then proceeds to compute the singular vectors corresponding to the k largest singular values [99].

The use of a sparse log co-occurrence matrix allows NetMF to capture more detailed information about the relationships between nodes in the graph.

1.2.3 Edge sampling methods

The key idea behind edge-sampling based methods is to use edge information to learn node embeddings. This is typically done by training a shallow neural network on triples consisting of a source node, a destination node, and a property of the edge that connects them. For example, the property might be a binary value indicating whether the edge exists, or it might be a weight that represents the strength of the connection between the nodes.

Large-scale Information Network Embedding

One example of an edge-sampling based method is the LINE (Large-scale Information Network Embedding) model [119]. This model samples node tuples and evaluates whether they are connected by an edge in the graph.

There are two main variations of the LINE model: first-order and second-order. In first-order LINE, only a single node embedding embedding layer is optimized. This means that connected nodes will have similar embeddings, while disconnected nodes will have dissimilar embeddings, with no possibility for the model to learn the directionality of the edge interaction. In second-order LINE, two node embeddings layers are optimized: one for the source nodes and one for the destination nodes (or contextual nodes). This allows the model to capture more complex patterns in the graph structure, such as higher-order connectivity between nodes.

1.2.4 Corrupted triple sampling methods

Similar to edge sampling methods, *corrupted*-triple sampling methods are shallow neural networks trained on the (*true*) triples (v, ℓ, s) defined by the existing edges in the graph (where v is the source node, ℓ is the property of the edge (v, s), and s is the destination node, see section 2.4), but also on *corrupted triples*, that are obtained by corrupting the original triples by substituting the source and/or destination nodes $\{v, s\}$ with randomly sampled nodes $\{v', s'\}$, while maintaining the attribute unchanged (v', ℓ, s').

The shallow neural network models used on corrupted-triple sampling batches include a weight matrix representing the node embedding, plus one or more matrices for representing the edge attributes, which are composed to capture the attribute meaning as algebraic operations (e.g. *woman+is_ royal* = *queen*). For this reason, they are particularly well suited to compute node and edge properties embedding of attributed graphs were the edge properties represent meaningful directed transitions (e.g. *is_ royal*), while being out of scope when dealing with local undirected properties (e.g. *interacts with*). Given a distance metric defined for the triples the shallow models are generally optimized to minimize the distance of *true* triples while maximizing the distance of the *corrupted* ones.

The distance defined for triples is often a feature-wise distance, whose advantage is that the computation of the gradient of each feature is independent from any other feature. This allows for particularly effective data-racing-aware and synchronization-free parallel implementations [132].

TransE

TransE [132] is among the first and possibly one of the most commonly used of the corrupted-triple sampling methods presented in the literature, from which a large family of variations has been defined. The model trains a shallow neural network composed of two weight matrices representing the node embedding and the edge type embedding. It generally uses as distance metric a feature-wise euclidean distance (though any element-wise distance metric may be used) and defines its energy loss as:

$$\mathcal{L}_{TransE} = \sum_{(v,v',\ell,s,s')} \text{ReLU}\left[\text{constant} + (v + \ell - s)^2 - (v' + \ell - s')^2\right]$$

There are a number of graph representation learning models that are derived from the TransE model. These derivative models include TransH, TransR, TransD, and others, which seek to improve upon the original TransE model by introducing additional mechanisms for modeling relationships between entities. For example, TransH introduces a hyperplane to capture the different types of relationships that can exist between entities, while TransR introduces a rotation matrix to allow for more flexible relationships.

The field of graph representation learning has seen significant advances in recent years, with a range of approaches being proposed to effectively capture the underlying structure and semantics of graphs. These approaches range from traditional matrix factorization methods, to random walk based methods, to more recent approaches such as graph neural networks. However, many of these approaches struggle with scalability and are unable to effectively process large real-world graphs. In the next chapter, we begin with the novelties introduced with this thesis, starting with the introduction of the GRAPE library, a software library developed to address these scalability issues and enable the effective processing of large graphs.

Chapter 2

The *GRAPE* resource

GRAPE is a graph representation learning library for executing node embedding, obtaining novel predictions and evaluating the obtained predictions. *GRAPE* has been developed in Rust, with Python bindings to enhance user accessibility, and its main focus is scalability, i.e. making possible the execution of tasks such as node embedding on large graphs even on commodity hardware, through an holistic attention to synchronization-free parallelism, instruction-level parallelism based on SSE and AVX, efficient data structures, numerical stability, and where necessary mixed precision and MMAP. Among the many high-performance algorithms provide by *GRAPE*, the library implements *approximated* weighted DeepWalk, Walklets & Node2Vec embedding models, able to process graphs containing high-degree nodes (degree $> 10^6$), an otherwise unmanageable task when using the analogous exact algorithms, and allow one to obtain edge-prediction performance comparable to those achieved by using the *exact* version (Section 4.1.1).

Finally, *GRAPE* can optionally employ succinct data structures based on Elias-Fano [38] (see section 2.2) to load graphs that would not otherwise fit within main memory, with memory usage close to the theoretical minimum. Rank and Select operations on succinct data structures are slower than their CSR analogues, nevertheless these operations have average constant-time complexity.

The node embeddings are then used for graph visualization or to solve graph-prediction tasks [77], including node-label prediction, edge and edge-label prediction, unsupervised graph analysis (e.g. node clustering).

2.1 *GRAPE* overview

GRAPE is a fast graph processing and embedding library; it extensively uses parallel computation and efficient data structures to scale with big graphs. The library's high-level structure, overall functionalities, and its two core modules, *Ensmallen* (ENabler of SMAll computational resources for LargE Networks) and *Embiggen* (EMBeddInG GENerator), are depicted in figure 2.1a.

Ensmallen efficiently loads big graphs and executes graph processing operations, owing to the design of efficient data structures and its Rust [93] implementation, with fully documented Python bindings for ease of use.

Rust is a compiled language gaining importance in the scientific community [93] thanks to its robustness, reliability, and speed. Rust allows threads and data parallelism to be exploited robustly and safely. To further improve efficiency, some core functionalities of the library, such as the generation of pseudo-random numbers and sampling procedures from a discrete distribution, use traditional map-reduce thread-based parallelism and branch-less Single Instruction Multiple Data (SIMD) parallelism (see Appendixes A and B).

Among the wide spectrum of implemented graph processing methods, *Ensmallen* also provides Bader and Kruskal algorithms for computing random and minimum spanning arborescence and connected components, stress and betweenness centrality [8], node and edge filtering methods, and algebraic set operations on graphs. *Ensmallen* allows graphs to be loaded from a wide variety of node and edge list formats (section 4.1).

As a result, users can automatically load data from an ever-increasing list of over $80,000$ graphs from the literature and elsewhere (Fig. 2.1b, detailed in section 4.2.2).

Importantly, *Ensmallen* also provides multiple (graph) holdout techniques, efficient node embedding methods (sections 2.3.1, 2.3.2, 2.4, based on, e.g., first and second-order random walks [70], triple [119] and corrupted-triple sampling [132], plus a wide range of graph processing algorithms that nicely scale with big graphs. Importantly *Ensmallen* also allows to compute edge embedding from the source and destination node embeddings using several different methods, such as Hadamard, concatenation, and element-wise L1 and L2 (section 2.5).

Once computed, the edge/node embeddings can be used as input of dimensionality reduction techniques (e.g. *t-SNE* [124]), to obtain lower-dimensional representations allowing to visualize graphs and their properties (fig. 2.1 **c**) or may be ingested any of the node-label, edge-label and edge prediction models (section 2.6) implemented into the *Embiggen* module or integrated by third-party libraries.

Embiggen provides GRL and inference models (sections 4.2.1), including an exhaustive set of node embedding methods, e.g., spectral and matrix factorization models such as HOPE [90], NetMF [99] and their variations (GLEE [122], SocioDim [120] - section 2.3.1), CBOW, SkipGram and GloVe embedding methods [80, 92] exploiting random walk-based methods such as DeepWalk, Node2Vec and Walklets [52, 96] (section 2.3.2), triple sampling methods such as LINE [119] and corrupted-triple sampling methods such as TransE [132] (section 2.4), and, more generally, a wide range of inference methods (sections 2.6).

GRAPE currently provides 49 unique node embedding models (61 considering redundant implementations, important for benchmarks), with 19 being "by-scratch" implementations and 30 integrated from third-party libraries. The list of available node embedding methods is constantly growing, with the ultimate goal to provide a complete set of efficient node embedding models. The input for the various models (e.g. random walks and triples) are provided by *Ensmallen* in a scalable, highly efficient, and parallel way (Fig. 2.1a). All models were designed according to the "composition over inheritance" paradigm, to ensure a better user experience through increased modularity and polymorphic behaviour [47]. More specifically, *Embiggen* provides interfaces, specific for either the embedding or each of the prediction-tasks, that must be implemented by all models; third-party models, such as PyKeen [3], KarateClub [106] and Scikit-Learn [91] libraries, are already integrated within *GRAPE* by implementing these interfaces. *GRAPE* users can straightforwardly create their models and wrap them by implementing the appropriate interface.

GRAPE provides three modular pipelines to compare and evaluate node-label, edge-label and edge prediction performance under different experimental settings (section 4.2.2, fig. 2.1b), as well as utilities for graph visualization (fig. 2.1c). These pipelines allow non-expert users to tailor their desired experimental setup and easily obtain actionable and reproducible results (Fig. 2.1b). Furthermore, *GRAPE* provides interfaces to integrate third-party models and libraries (e.g., KarateClub [106] and PyKeen [3] libraries). In this way, the evaluation pipelines can be used to obtain a fair comparison between models implemented or integrated in *GRAPE.*

The possibility to integrate external models and the availability of graphs for testing them on the same datasets allows to answer a still open and crucial issue in literature, which regards the FAIR (Findable, Accessible, Interoperable, and Reusable), objective, reproducible, and efficient comparison of graph-based methods and software implementations (Section 4.2.2).

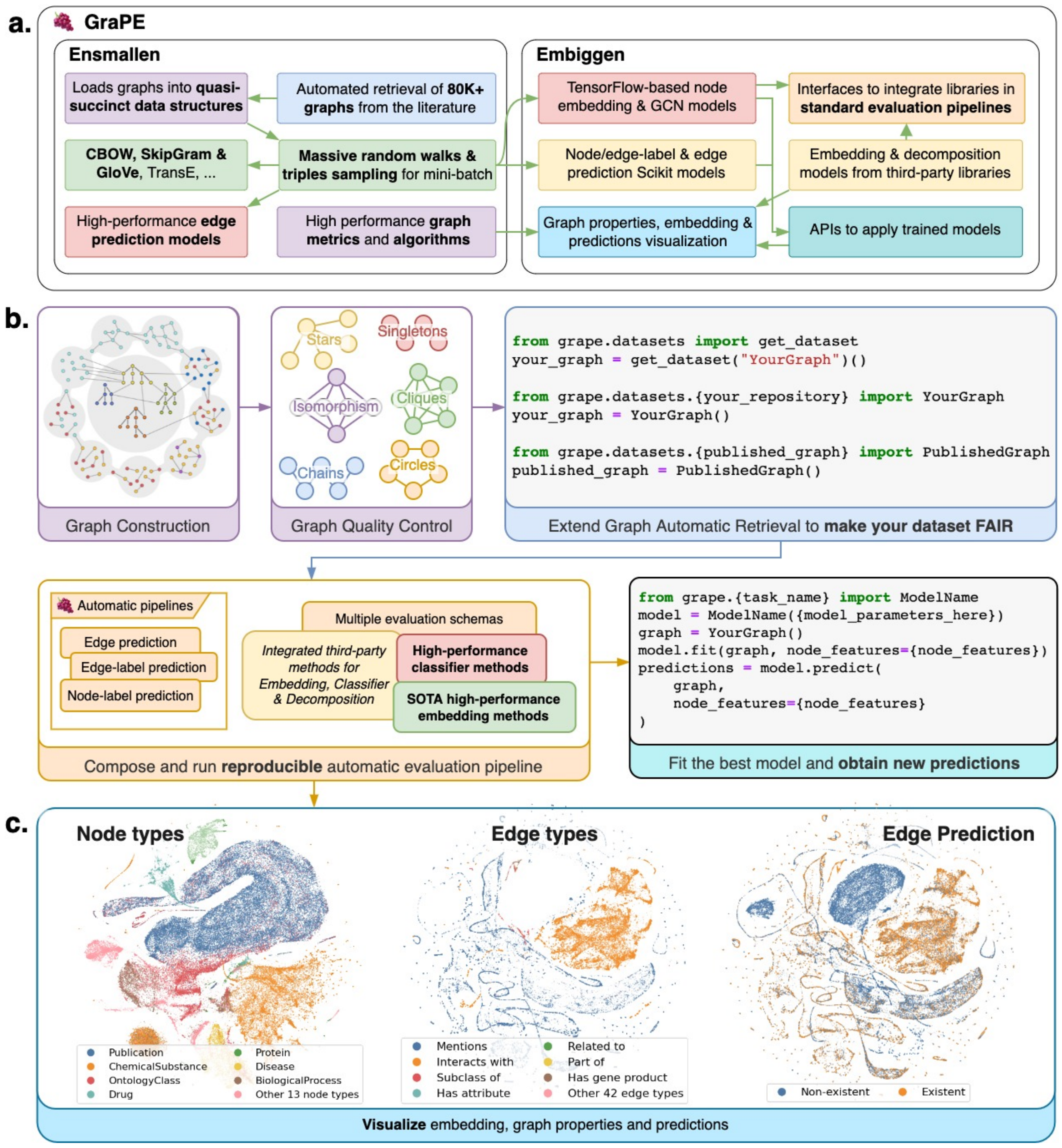
GRAPE has a comprehensive test suite. However, to thoroughly test it against many scenarios, we also employed fuzzers, that is tools that iteratively generate inputs to find corner cases in the library.

In the next section we describe the succinct data structures used in the library and detail their efficient *GRAPE* implementation (Section 2.2). We then summarize the spectral and matrix factorization (Section 2.3.1), the random walk-based (Section 2.3.2), the triple and corrupted triples-based (Section 2.4) embedding methods and their *GRAPE* implementation. In section 2.5 we describe the edge embedding methods and in Section 2.6 the node and edge label prediction methods available in *GRAPE.* Finally in Section 2.7 we detail the *GRAPE* standardized pipelines to evaluate and compare models for graph prediction tasks.

In Chapter 4 we report some of the experimental results we conducted to assess the efficiency and efficacy of *GRAPE.*

2.2 Succinct data structures for adjacency matrices

Besides heavy exploitation of parallelism, the second pillar of our efficient implementation is the careful design of the data structures for using as little memory as possible and quickly performing operations on them. The naive representation of graphs explicitly stores its adjacency matrix, with a $\mathcal{O}(|V|^2)$ time and memory complexity, being $|V|$ the number of nodes, which leads to intractable memory costs on large graphs. However, since most large graphs are highly sparse, this problem can be mitigated by storing only the existing edges. Often, the adopted data structure is a Compressed Sparse Rows matrix (CSR [107]), which stores the source and destination indices of existing edges into two sorted vectors. In *Ensmallen* we further compressed the graph adjacency matrix by adopting the Elias-Fano succinct data scheme, to efficiently store the edges (Supplementary section 2.2.2). Since Elias-Fano representation stores a sorted set of integers using memory close to the information-theoretical limit, we defined a bijective map from the graph-edge set and a sorted integer set. To define such encoding, we firstly assigned a numerical id from a dense set to each node, and then we defined the encoding of an edge as the concatenation of the binary representations of the numerical ids of the source and destination nodes. This edge encoding has the appealing property of representing the neighbours of a node as a sequential and sorted set of numeric values, and can therefore be employed in the Elias-Fano data structure. Elias-Fano has faster sequential access than random access (Supplementary section S7.1.1) and is well


a.
GraPE
Ensmallen
Loads graphs into quasi-succinct data structures
Automated retrieval of 80K+ graphs from the literature
CBOW, SkipGram & GloVe, TransE, ...
Massive random walks & triples sampling for mini-batch
High-performance edge prediction models
High performance graph metrics and algorithms
Embiggen
TensorFlow-based node embedding & GCN models
Interfaces to integrate libraries in standard evaluation pipelines
Node/edge-label & edge prediction Scikit models
Embedding & decomposition models from third-party libraries
Graph properties, embedding & predictions visualization
APIs to apply trained models
b.
Graph Construction
Stars
Singletons
Isomorphism
Cliques
Chains
Circles
Graph Quality Control
Extend Graph Automatic Retrieval to make your dataset FAIR
Automatic pipelines
Edge prediction
Edge-label prediction
Node-label prediction
Multiple evaluation schemas
Integrated third-party methods for Embedding, Classifier & Decomposition
High-performance classifier methods
SOTA high-performance embedding methods
Compose and run reproducible automatic evaluation pipeline
Fit the best model and obtain new predictions
c.
Node types
Publication
ChemicalSubstance
OntologyClass
Drug
Protein
Disease
BiologicalProcess
Other 13 node types
Edge types
Mentions
Interacts with
Subclass of
Has attribute
Related to
Part of
Has gene product
Other 42 edge types
Edge Prediction
Non-existent
Existent
Visualize embedding, graph properties and predictions

```python
from grape.datasets import get_dataset
your_graph = get_dataset("YourGraph")()

from grape.datasets.{your_repository} import YourGraph
your_graph = YourGraph()

from grape.datasets.{published_graph} import PublishedGraph
published_graph = PublishedGraph()
```

```python
from grape.{task_name} import ModelName
model = ModelName({model_parameters_here})
graph = YourGraph()
model.fit(graph, node_features={node_features})
predictions = model.predict(
    graph,
    node_features={node_features}
)
```

Figure 2.1: Schematic diagram of *GRAPE* (*Ensmallen* + *Embiggen*) functionalities. **a.** High level structure of the *GRAPE* software resource. **b.** Pipelines for an easy, fair, and reproducible comparison of graph embedding techniques, graph-processing methods, and libraries. **c.** Visualization of KGCOVID19 graph, obtained by displaying the first two components of the *t-SNE*decomposition of the embeddings computed by using a Node2Vec SkipGram model that ignores the node and edge type during the computation. The clusters colors indicate: (left) the Biolink category [123] for each node; (center) the Biolink category for each edge; (right) the predicted edge existence.

suited for graph processing tasks such as retrieving neighbours during random walk computation and executing negative sampling using the outbound or inbound node degrees distributions.

2.2.1 Edge Encoding

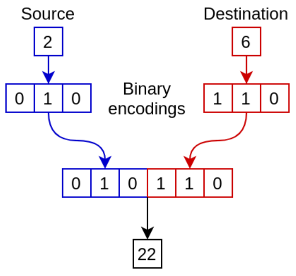
In this subsection we describe how *Ensmallen* converts all the edges of a graph $G(V, E)$ into a sorted list of integers. In particular, considering an edge $e = (v, x) \in E$ connecting nodes v and x represented with, respectively, integers a and b, the binary representation of a and b are concatenated through the function $\phi_k(a, b)$ to generate an integer index uniquely representing the edge e itself:

$$\phi_k(a,b) = a\ 2^k + b,\ \text{where}\ k = \lceil \log_2 |V| \rceil\ \Rightarrow\ a = \left\lfloor \frac{\phi_k(a,b) - b}{2^k} \right\rfloor,\quad b = \phi_k(a,b) - a\ 2^k$$

This implementation is particularly fast because it requires only few bit-wise instructions:

$$\phi_k(a,b) = a << k|b \quad \Rightarrow \quad a = \phi_k(a,b) >> k, \quad b = \phi_k(a,b)\ \&\ (2^k - 1)$$

where $<<$ is the left bit-shift, $|$ is the bit-wise OR and & is the bit-wise AND. Since the encoding uses $2k$ bits, it has the best performances when it fits into a CPU word, which is usually 64-bits on modern computers, meaning that the graph must have less than 2^{32} nodes and and less than 2^{64} edges. However, by using multi-word integers it can be easily extended to even larger graphs [63]. As an example, considering a graph with at most 8 nodes, encoded with integers numbers ($v \in [0, \ldots, 7]$) In figure 2.2 we schematize the encoding of the edge $(2, 6)$ which has 2 as source node, and 6 as destination node. On the right we report the Rust implementation of the edge encoding and decoding. Once the edges are encoded, we can sort them and use Elias-Fano to store them.



```
fn encode(src: u64, dst: u64, k: u64) -> u64 {
    (src << k) | dst
}
fn decode(edge: u64, k: u64) -> (u64, u64) {
    let src = edge >> k;
    let dst = edge & ((1 << k) - 1);
    (src, dst)
}
```

Figure 2.2: On the left, an example of the encoding. On the right its implementation in Rust.

2.2.2 Elias-Fano scheme

Once all the edges listed in the adjacency matrix have been converted to a sorted list of integers as defined in subsection 2.2.1), *Ensmallen* uses the Elias-Fano scheme [38, 125] to represent the edges in memory. This not only allows minimizing the memory costs for storing the graph, but also allows implementing fast loading and processing operations.

In the following, we aid comprehension of Elias-Fano schema by supporting the description with a graphical example, depicted in figure 2.3. The figure shows how the monotone list of integers [5, 8, 8, 15, 32] may be represented in memory by using Elias-Fano [38] quasi-succint representation.

More precisely, the binary representation of the i-th value of the list, $b_i,\ i = 1, \ldots, n$, is initially split into two parts:

- a low-bits part, l_i (light blue in figure 2.3), containing the lower $\lfloor \log_2 \frac{u}{n} \rfloor$ bits,
- an high-bits part, h_i (purple in figure 2.3), containing the remaining bits (referred to as high-bits).

The low-bit-parts and and high-bit-parts are then used to compose,

- a low-bits array (light-blue and referred to as L in figure 2.3): it is formed by sequentially concatenating the explicit copies of the low-bits parts.
- an high-bits array (red, and named H in figure 2.3): it is composed by sequentially concatenating the unary representation of the differences (gaps) between consecutive high-bits parts (where the preceding value of the first element is assumed to be equal to zero). It is important to note that each gap, is represented in memory by using the inverted unary representation, that is by encoding an integer value equal to k by using k zeros followed by a one (for example, 0 in inverted unary representation is 1, 3 is 0001).

Instead of computing the gaps, Pibiri et al. [97] propose a faster method to build the high-bits array H, which obtains the same encoding as Elias-Fano's. Pibiri's et al. show that H can be created as a bit-vector with all zeros, except for the elements at indexes $h_i + i$ ($i = 1, \ldots, n$, where i indexes the high-bits parts, h_i) that are set to 1. This method is faster because the encoding and decoding of each value no longer depend on the previous values (as gaps would) so that the representation may be built in parallel to further speed up the graph representation. Moreover, once the index of each value is known, exploiting atomic integers we can **build Elias-Fano fully in parallel** [1] without any lock. An example of this method is in figure 2.4, where the high-bits index is no longer computed using the gaps.

Elias-Fano representation allows storing n non-negative integers, sorted in increasing orders and bounded by u, with at most $\mathcal{EF}(n, u) = 2n + n \left\lceil \log_2 \frac{u}{n} \right\rceil$ bits, and a memory usage that is less than half a bit away [39, 38, 40] from the succinct bound that is $Z + o(Z)$, where Z is the theoretical minimum number of bits needed to store the data: $Z = \left\lceil \log_2 \binom{u}{n} \right\rceil = n \log_2 \frac{u}{n} + \mathcal{O}(n)$ [98].

[1] full implementation at `https://github.com/zommiommy/elias_fano_rust/blob/develop/src/concurrent_builder.rs`

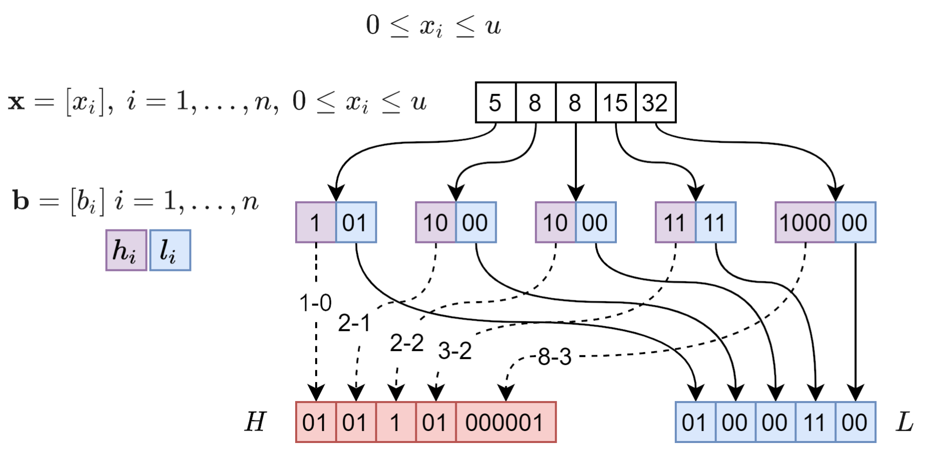


Figure 2.3: Example of Elias-Fano. Elias-Fano splits the sorted values into high and low bits; the low-bits parts are then consecutively copied in the low-bits array, L (on the right, blue color), while the high-bits parts are coded into an high-bits array H (on the left, red color) by consecutively storing the gaps between consecutive high-bits parts, encoded by using the inverted unary representation.

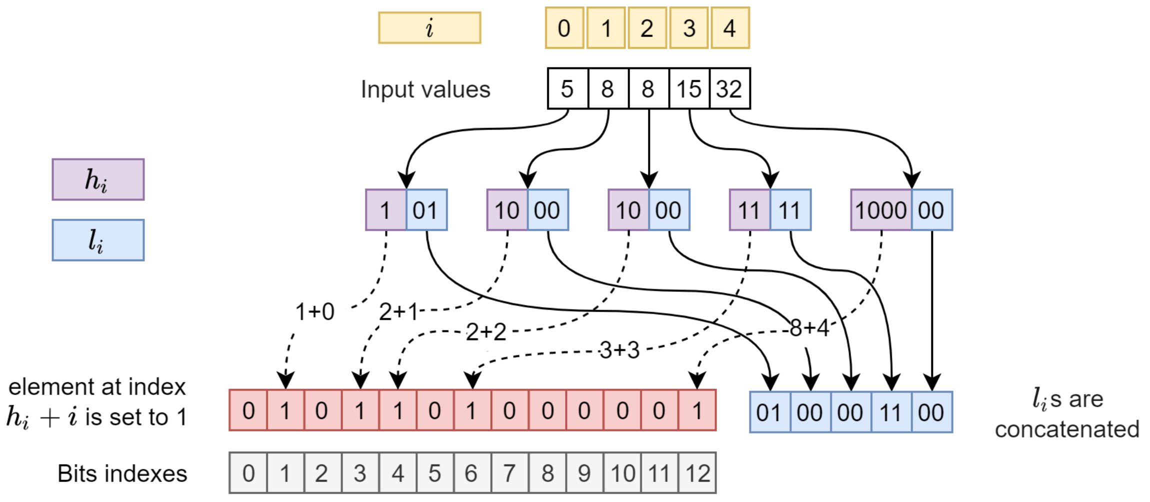


Figure 2.4: Algorithm for Elias-Fano encoding presented in [97]. While the low-bits array (on the right, blue color) is simply composed by sequentially concatenating the l_is for each $i = 1, \ldots, n$, the high-bits array is composed by setting the bit at index $h_i + i$ to 1.

2.2.3 Operations on Elias-Fano.

When paired with Elias-Fano representation, the aforementioned encoding allows efficient computation of random-walk samples.

Indeed, Elias-Fano representation allows performing **rank** and **select** operations by requiring on average constant time. These two operations were initially introduced by Jacobson to simulate operations on general trees, and were subsequently proven fundamental to support operations on data structures encoded through efficient schemes.

In particular, given a set of integers S, Jacobson defined the **rank** and **select** operations as follows [85, 98]:

rank(S, m)	returns the number of elements in S less or equal than m
select(S, i)	returns the i-th smallest value in S

As explained below, to speed up computation, we deviate from this definition by defining the rank operation as the number of elements strictly lower than m.

To compute the neighbours of a node using the rank and select operations, we observe that, for every pair of nodes α, β with numerical ids a, b respectively, for $a, b < 2^k$ where k is generally 32, it holds that:

$$a\ 2^k \leq a\ 2^k + b < (a+1)\ 2^k \qquad \Rightarrow \qquad \phi_k(a,0) \leq \phi_k(a,b) < \phi_k(a+1,0)$$

Thus, the encoding of all the edges with source α will fall in the discrete range

$$\Big[\phi_k(a,0),\ \phi_k(a+1,0)\Big) = \Big[a\ 2^k,\ (a+1)\ 2^k\Big)$$

Thanks to our definition of the **rank** operation and the aforementioned property of the encoding, we can easily derive the computation of the degree $d(a)$ of any node v with numerical id a for the set of encoded edges Γ of a given graph, which is equivalent to the number of outgoing edges from that node:

$$d(a) = \mathbf{rank}(\Gamma, \phi_k(a+1,0)) - \mathbf{rank}(\Gamma, \phi_k(a,0))$$

Moreover, we can retrieve the encoding of all the edges Γ_a starting from v encoded as a, by selecting every index value i falling in in the range $[\phi_k(a,0), \phi_k(a+1,0)$:

$$\Gamma_a = \Big\{\mathbf{select}(\Gamma, i) \mid \mathbf{rank}(\Gamma, \phi_k(a,0)) \leq i < \mathbf{rank}(\Gamma, \phi_k(a+1,0))\Big\}$$

We can then decode the numerical id of the destination nodes from Γ_a, thus finally obtaining the set of numerical ids of the neighbours nodes $N(a)$:

$$N(a) = \Big\{\mathbf{select}(\Gamma, i)\ \&(2^k - 1) \mid \mathbf{rank}(\Gamma, \phi_k(a,0)) \leq i < \mathbf{rank}(\Gamma, \phi_k(a+1,0))\Big\}$$

In this way, by exploiting the above integer encoding of the graph and the Elias-Fano data scheme, we can efficiently compute the degree and neighbours of a node using rank and select operations.

```rust
/// Return number of elements smaller than value
fn rank(&self, value: u64) -> u64 {
    // split the value into
    // its higher and lower bits
    let high = value >> self.low_bits_size;
    let low = value & self.low_bits_mask;
    // find the index of the
    // high-th zero in the bit-vector
    let mut index = self.high_bits.find_zero_of_index(
        high
    );
    // start scanning the lower
    // bits to find the first element
    // bigger or equal than value
    while (
        self.high_bits[index] == 1 &&
        self.read_lowbits(index - high) < low
    ){
        index += 1;
    }
    // the number of elements is equivalent
    // to the index of the value
    // in the high-bits
    // minus the higher bits because
    // the count is the number
    // of ones before index and
    // high is the number of zeros before index.
    index - high
}
```

Figure 2.5: Rust implementation of a simplified rank operation: in this implementation we omitted the logic needed to handle all the corner-cases. The full implementation can be found at `https://github.com/zommiommy/elias_fano_rust/blob/master/src/elias_fano.rs`

```rust
/// Find index-th smallest value
fn select(&self, index: u64) -> u64 {
    // find the index of the high-th
    // one in the bit-vector
    // and subtract the index
    // to obtain the number of
    // zeros before the index-th one.
    let high = self.high_bits.find_one_of_index(
        index
    ) - index;
    // get the lower bits of the value
    let low  = self.read_lowbits(index);
    // merge the high and low bits
    (high << self.low_bits_size) | low
}
```

Figure 2.6: Rust implementation of a simplified select operation: in this implementation we omitted the logic needed to handle all the corner-cases. The full implementation can be found at `https://github.com/zommiommy/elias_fano_rust/blob/master/src/elias_fano.rs`

The Rust implementations of rank and select operations on our Elias-Fano representation are sketched in Figures 2.5 and 2.6.

This implementation may be further optimized by considering that The complexity of the rank (select) operation mainly depends on the implementation of the *find_ zero_ of_ index* (*find_ one_ of_ index*) function since all the other operations take constant time.

Therefore, to have **rank** and **select** in constant time we need to obtain a constant computational time for those functions.

Both functions need to find the i-th one/zero in a bit-array. Therefore, the naive way to solve the problem would be to scan the array from the start and, at each step, count how many values $v \in \{0, 1\}$ have been encountered so far. This algorithm scales linearly with the length of the bit-vector and thus is not practical for large arrays. A simple solution to improve the linear scan is to get a starting point that is closer to the result.

In the following paragraphs, we will adopt the term "quantum" from [125] to describe an integer value q, most commonly equal to a power of two for hardware efficiency reasons, as logical operations are significantly faster.

To do so, given a bit-array to scan, we choose a quantum q and store the position of every q-th value $v \in \{0, 1\}$ into an auxiliary array $O = [o_0, o_1, \dots]$. To find the i-th value v in Elias-Fano's high-bits array, the linear scan will start from position o_k, where $k = \lfloor i/q \rfloor$, and it will have to scan at most until the next position o_{k+1}.

Considering that Elias-Fano's high-bits array contains approximately half uniformly distributed ones and half uniformly distributed zeros, the average distance between two consecutive values $v \in \{0, 1\}$ is equal to 2 bits, which implies that the average distance between two consecutive positions o_k and o_{k+1}, that is the expected maximum number of bits to scan for searching a the i-th value v, is $\mathbb{E}[o_{k+1} - o_k] = 2q$.

Therefore, if the high-bits array has n bits, by using the auxiliary vector O, the average time complexity in the worst case is reduced from $\mathcal{O}(n)$ to $\mathbb{E}[o_{k+1} - o_k] = \mathcal{O}(q) = \mathcal{O}(1)$.

In computer science and computer engineering, an assembler is a low-level programming language that is used to write programs that can be executed directly by a computer's central processing unit (CPU). Assembler instructions are specific to a particular type of CPU and are used to perform basic operations such as moving data, performing arithmetic calculations, and controlling the flow of a program.

The **popcnt** instruction is a specific type of assembler instruction that is used to compute the number of "1" bits (also known as "set bits") in a given word of memory. This instruction can be useful for various purposes, such as for counting the number of occurrences of a particular value within a data structure.

The BMI2 instruction set is a collection of assembler instructions that are supported by certain types of CPUs. These instructions provide various useful operations for working with bit strings, such as the ability to extract or insert individual bits from a word of memory, or to find the position of the first set bit in a word.

The instructions **pdep** and **tzcnt** are part of the BMI2 instruction set and are used to find the wanted value v in a word of memory in constant time. The **pdep** instruction is used to extract the bits of a word that correspond to the wanted value, while the **tzcnt** instruction is used to find the position of the first set bit in the resulting bit string. By combining these instructions with other operations, it is possible to find the wanted value v in constant time, providing a significant speed-up over more general-purpose algorithms.

For exhaustiveness, a simplified version of the optimized Rust implementation of *GRAPE* **select** operation is reported in Figures 2.7 and 2.8.

For what regards the *memory complexity* of the proposed index, we need on average $\mathcal{O}\left(\frac{n}{2q}\right)$ integers for storing the positions of the ones (or of the zeros), and each integer position needs at most $\lceil \log_2 n \rceil$ bits to be stored. Therefore, each of the two indexes (one index for the ones to speed the select, and one index for the zeros to speed the rank) cause a memory occupation, in the worst case, of $\mathcal{O}\left(\frac{n}{2q} \log_2 n\right)$ bits, which resolves to a total worst-case memory occupation of $\mathcal{O}\left(\frac{n}{q} \log_2 n\right)$ bits.

Note that while indices with memory complexity of $o(n)$ exist, a careful implementation allows the use of a relatively high value for q, which practically results in low overhead, while having the advantage of lowering the computational costs.

In the following examples, we will use the mebibytes (MiBs) instead of megabytes (MB). A mebibyte is a unit of measurement for digital information. It is commonly used to measure the amount of data stored on computer hard drives or the amount of data transferred over a network. A mebibyte is equal to 1,048,576 bytes, which is slightly larger than a megabyte, which is equal to 1,000,000 bytes. In other words, one mebibyte is equal to approximately 1.049 megabytes. The prefix "mebi" comes from the binary system of measurement, where "bi" stands for "binary digit," and the prefix "mega" comes from the decimal system of measurement, where "mega" stands for "million." Mebibytes are used instead of megabytes because they are more precise and accurate when used in the context of digital information. In the binary system of measurement, which is commonly used in computing, 1,048,576 is the closest whole number approximation of 1,000,000. This is why 1 mebibyte is slightly larger than 1 megabyte. Because mebibytes are based on the binary system of measurement, they are more precise and more commonly used in computing contexts, whereas megabytes are based on the decimal system of measurement and are more commonly used in everyday contexts.

Indeed, our tests on a knowledge graph (KG-COVID [101], section 3) with around 450'000 nodes and 32'000'000 edges, showed that Elias-Fano uses 56Mib to store the adjacency matrix, of which 0.6 Mib are for the **1s** and **0s** indices (see table 2.1. Therefore, the overhead ratio of the indices is 1.07% when compared to the size of the whole structure.

Moreover, rank and select operations on KG-COVID require in average in 50ns and 118 ns when executed on a Ryzen 9 3900x.

```
/// Returns position of `index`-th bit set to one.
pub fn select1(&self, index: u64) -> u64 {
    // use the index to find
    // in which block the value is
    let mut reminder_to_scan = index & INDEX_MASK;
    let idx = (index >> INDEX_SHIFT) as usize;
    // the bit position of the biggest
    // multiple of INDEX_SIZE which is
    // smaller than the choosen index,
    // this is were we will start our search
    let pos = self.high_bits_index_ones[idx];
    // find in which word the start value is
    let mut block_id = (pos >> WORD_SHIFT) as usize;
    let in_word_reminder = pos & WORD_MASK;
    // build the standard word to start scanning
    let mut code = self.high_bits[block_id];
    // clean the "already parsed lower bits"
    code &= u64::MAX << in_word_reminder;
    // use popcnt to find the right word
    loop {
        let popcnt = code.count_ones() as u64;
        if popcnt > reminder_to_scan {
            break
        }
        block_id += 1;
        reminder_to_scan -= popcnt;
        code = self.high_bits[block_id];
    }
    // Find index of `reminder_to_scan`-th
    // one in `code`
    let in_word_index = select1_in_word(
        code,
        reminder_to_scan
    );

    (block_id * WORD_SIZE) + in_word_index
}
```

Figure 2.7: Select implementation with Rust in this implementation we omitted the logic needed to handle all the corner-cases. The full implementation can be found at `https://github.com/zommiommy/elias_fano_rust/blob/master/src/elias_fano.rs`

```
/// Find index of `index`-th one in word
pub fn select1_in_word(
    word: u64,
    index: u64
) {
    // If the cpu supports
    // the BMI2 instruction set
    // use the optimized version
    // that exploits PDEP
    #[cfg(target_feature="bmi2")]
    unsafe {
        return core::arch::x86_64::_pdep_u64(
            1_u64 << n, x
        ).trailing_zeros() as u64;
    }
    // otherwise fall down
    // to the generic version
    #[cfg(not(target_feature="bmi2"))]
    {
        for _ in 0..reminder_to_scan {
            // reset the lowest set bits
            // if the cpu supports BMI1
            // this is transalted to
            // a `BLSR` instruction
            code &= code - 1;
        }
        return code.trailing_zeros() as u64;
    }
}
```

Figure 2.8: Select implementation with Rust in this implementation we omitted the logic needed to handle all the corner-cases. The full implementation can be found at `https://github.com/zommiommy/elias_fano_rust/blob/master/src/elias_fano.rs`

DataStructure	Select Time (ns)
Vec	17
EliasFano	**120**
Indexed BitVec	906
Rank9	2362
Jacobson	3266
Fid	8722
RsDict	12021

DataStructure	Rank Time (ns)
Rank9	9
Jacobson	19
EliasFano	**50**
Indexed BitVec	51
Vec	72
Fid	96
RsDict	113

DataStructure	Memory (Mib)
EliasFano	**56**
Vec	256
RsDict	13'344

Table 2.1: Time (top tables) and memory (bottom table) complexities required by our implementation of Elias-Fano and by other well-known data-structures. Time and memory costs were computed by using the KgCovid graph [101]. Elias-Fano offers good performances in both time (top tables) and memory (bottom table). On note, methods, such as RsDict, that are characterized to obtain good compression for sparse bit-vectors, fail on a graph as sparse as KgCovid, whose adjacency matrix has only the 0.015% of 1s.

2.2.4 Memory Complexity

Elias-Fano is a quasi-succinct data representation scheme, which provides a memory efficient storage of a monotone list of n sorted integers, bounded by u, by using at most $\mathcal{EF}(n, u) = 2n + n \left\lceil \log_2 \frac{u}{n} \right\rceil$ bits, which was proven to be less than half a bit per element away from optimality [38] and assures random access to data in average constant-time. Thus, when Elias-Fano is paired with the previously presented encoding, the final memory complexity to represent a graph $G(V, E)$ is $\mathcal{EF}_\phi(|V|, |E|) = \mathcal{O}\left(|E| \log \frac{|V|^2}{|E|}\right)$; this is asymptotically better than the $\mathcal{O}\left(|E| \log |V|^2\right)$ complexity of the CSR scheme.

2.2.5 Optional memory-time trade-offs

Even if the Elias-Fano quasi-succinct data structure enables efficient operations on graphs, *Ensmallen* provides the following three options that may be set to further speed-up the computation at the expense of a more expensive usage of the main memory.

Explicit destinations vector The first and most important option is to explicitly create the vector of the destination nodes, avoiding to execute a select from the Elias-Fano data structure each time a given destination node must be chosen. This allows achieving a speedup during the random walks (on average an x3-4 speedup) while spending twice as much memory.

Explicit out-bound degrees vector The second most important option is to create the vector of the out-bound node degree, which avoids extracting the degree of a source node from the Elias-Fano data structure. While spending a relatively limited amount of RAM, this grants on average an additional 10% speedup in the computation of random walks. When combined with the explicit destinations vector, it can achieve a combined speedup of up to x5-6 of random walks' computation. We suggest enabling this option when computing a random walk-based model, such as CBOW or SkipGram.

Explicit sources vector In the context of the generation of edge-prediction batches, the explicitly creation of both the vector of sources and the vector of destinations avoids accessing the Elias-Fano data-structure at all. In this way we spend around three times more memory, but we can achieve between three to four times speedup for the generation of edge-prediction batches.

2.3 Embedding Methods available from *GRAPE*

In this section we sketch the embedding methods made available through *GRAPE*.

More precisely, we summarize embedding methods based on Spectral or matrix decompositions (subsection 2.3.1), random-walk based embedding techniques (subsection 2.3.2)

In many of the graph representation learning models we describe in the following section, we are faced with a variety of different methods and models that can be used to solve a particular prediction problem. These methods often involve making arbitrary choices about the metrics and parameters that are used to evaluate their performance. This can make it difficult to compare and evaluate the different approaches in a fair and unbiased manner.

To address this issue, it is important to develop a simple and standardized pipeline for comparing and evaluating different methods. This pipeline should be designed to clearly evaluate the influence of arbitrary choices and eliminate biases, and to provide a fair and objective assessment of the performance of different approaches. By doing so, we can more accurately compare and evaluate the various methods and models that are available, and select the ones that are most likely to be effective for a given task. We will further discuss the evaluation pipelines available in the *GRAPE* library in section 2.7.

2.3.1 Spectral and matrix factorization node embedding methods

Spectral and matrix factorization methods start by computing weighted adjacency matrices, and may include one or more factorization steps. Secondly, given a target embedding dimensionality k, these models generally use as embeddings the k eigenvectors or singular vectors corresponding to spectral or singular values of interest.

GRAPE provides all of the spectral and matrix factorization methods detailed in the state of the art section of this thesis.

GRAPE provides efficient parallel methods to compute the initial weighted adjacency matrix of the various implemented methods, which are computed either as dense or sparse matrices depending on how many non-zero values the metrics are expected to generate. The computation of the singular vectors and eigenvectors are currently computed using the state-of-the-art LAPACK library [5].

LAPACK is a software library for numerical linear algebra. It provides routines for solving systems of linear equations, least-squares solutions of linear systems, eigenvalue problems, and singular value decomposition. LAPACK is written in Fortran and is designed to be efficient on a wide variety of computer architectures. It is commonly used in scientific and engineering computing, as well as in other fields that require high-performance numerical computing.

2.3.2 First and second-order random walk methods.

First- and second-order random-walk embedding models are shallow neural networks generally composed by two layers and trained on batches of random-walk samples. Given a window size, these models learn some properties of the sliding windows on the random walks, such as the co-occurrence of two nodes in each window, the window central node given the other nodes in the window, or vice-versa the nodes in the window from the window central node. The optimal window size value may vary considerably depending on the graph diameter and overall topology. Once the shallow model has been optimized, the weights in either the first or the second layer can be used as node embeddings.

DeepWalk and its Walklets [96] extension to a multiscale random-walk representation (detailed below) are first-order random-walk sampling methods.

Node2Vec [52], is a second-order random walk method that uses weights to bias the walk towards breadth-first search or depth-first search (section 2.3.2). Node2Vec random walks are more computationally expensive than first-order random walks (see figure 4.1 **c** and **e**), since they require to tune two parameters, and our experimental results showed that models trained on Node2Vec walks do not necessarily outperform models trained on first-order walks, when a sufficient amount of training samples is made available (see figure 4.3). This of course depends also on the characteristics of the graph, since it is well-known that by tuning the return and in-out parameters of Node2vec we can capture different topological and structural features of the underlying graph [52].

GloVe [92] trains a two-layer neural network to predict the logarithm of the co-occurrence frequency of two nodes within the contextual window of size w in random walks. CBOW [80] also trains a two-layer neural network to predict the central node of a random walk sequence given the other contextual nodes. SkipGram [80] resembles a transposed version of CBOW: it predicts the contextual nodes of a sequence given its central node. SkipGram has a computational complexity w times higher than CBOW, as it executes w times more weights updates for each training sample. As a result, SkipGram models often achieve better performance than CBOW models. Glove, CBOW and SkipGram may be trained with sequences sampled using either DeepWalk or Node2Vec.

Walklets-based SkipGram (or CBOW) computes w times a DeepWalk-based SkipGram (or CBOW) embedding with window size 1. For each $0 \dots i \dots w$ embedding, Walklets filters the random-walks by keeping only nodes whose position within the random walk belongs to the congruence class in module i [96]. This is done to obtain node embedding that learn multi-scale random walk representations.

Role2Vec with Weisfeiler-Lehman Hashing [2, 111, 106] uses first-order random walks to approximate the point-wise mutual information matrix obtained by multiplying the pooled adjacency power matrix with a structural feature matrix (in this case, Weisfeiler-Lehman features) to obtain a structural node embedding.

SkipGram and CBOW models are trained using degree-based negative sampling, which is efficiently implemented using the Elias-Fano data structure rank method.

To obtain reliable embeddings, the training phase of the shallow model would need an exhaustive set of random-walk samples to be provided for each source node, so as to fully represent the source-node context. When dealing with big graphs, the computation of a proper amount of random-walk samples needs efficient routines to represent the graph into memory, retrieve and access the neighbors of each node, randomly sample an integer, and, in case of (Node2Vec) second-order random walks [52], compute the transition probabilities, which must be recomputed at each step of the walk.

The approximated random walk is implemented using an optimized algorithm for sampling k unique sorted integers among n integers according to an approximate uniform distribution. The algorithm, called SUSS - 2.3.2-1 exploit a SIMD routine (B). When the graph is weighted, another SIMD routine is used to compute the cumulative sum of the unnormalized probability distribution (subsection A).

The implementation of the second-order random walk requires more sophisticated routines described in sections , and 2.3.2-2 and 2.3.2-3. Moreover, in section 2.3.2-4 we present an approximated weighted and second-order random walk that allows to deal with high-degree nodes.

SUSS: Sorted Unique Sub-Sampling

Within the approximated random walks, at each step, it is necessary to sample k destination nodes of the current node. The ids associated to these nodes need to be sorted in ascending order, as the procedures for efficient sampling of second-order random walks described in section 2.3.2 have this requirement. It is conceivable to straightforwardly sample uniformly k nodes, and then sort the obtained vector, which would produce the desired output. To avoid having to sort the nodes vector, we propose a procedure to sample the nodes semi-uniformly and sorted by design, and therefore avoid altogether the need for the sorting step. We call this procedure Sorted Unique Sub-Sampling algorithm, or SUSS.

SUSS is not necessarily better than more general random integer procedures, as they serve different purposes. SUSS is a procedure designed to efficiently sample semi-uniformly distributed and sorted integers, while general random integer procedures are functions that returns random integers from a given range. In the context of approximated random walks, where it is necessary to sample destination nodes of a given node and sort them in ascending order, SUSS may be more efficient and reliable than simply using random integer procedures and sorting the resulting vector of integers. However, in other contexts where the requirements are different, random integer procedures may be more suitable. Additionally, the performance of SUSS and random integer procedures may vary depending on the specific implementation and parameters used.

SUSS has been designed in *Ensmallen* to allow sub-sampling k unique sorted integers among n integers, by following an approximate uniform distribution. After splitting the range $[0, \ldots, n-1]$ into k equal segments (buckets) with length $\lfloor \text{delta}/k \rfloor$, SUSS samples an integer from each bucket by using Xorshift random number generator (appendix B). The implementation of the algorithm is reported in Algorithm 1. To establish whether the distribution of the integers sampled with SUSS is truly approximating a uniform distribution, we sampled $n = 10.000.000$ integers over $[0, \ldots, 10.000]$, by using both SUSS and by drawing from a uniform distribution in $[0, \ldots, 10.000]$. We then used the Wilcoxon signed-rank test to compare the frequencies of the obtained indices and we obtained a p-value of 0.9428, meaning that there is not a statistically significant difference between the two distributions. Therefore, by using a time complexity $\Theta(k)$ and a spatial complexity $\Theta(k)$ SUSS produces reliable approximations of a uniform distribution.

Algorithm 1 Sorted Unique Sub-Sampling (SUSS)

```
Require: Minimum value of range min_val
Require: Maximum value of range max_val
Require: Quantity of values to sample $k$
Require: Seed to reproduce the sampling $s$
  extracted ← []
  delta ← max_val − min_val
  step ← $\lfloor \text{delta}/k \rfloor$
  for $i \leftarrow 0; i < k - 1; i \leftarrow i + 1$ do
      extracted.push(min_val + step ∗ $i$ + $s$ % step)
      $s \leftarrow \text{xorshift}(s)$
  extracted.push(max_value − $s$ % (delta − step ∗ ($k$ − 1)) − 1)
  return extracted
```

Efficient sampling for Node2Vec random walks

Sampling from a discrete probability distribution is a fundamental step for computing a random walk and can be a significant bottleneck. Many graph libraries implementing the Node2Vec algorithm speed up sampling by using the Alias method (see Appendix C), which allows sampling in constant time from a discrete probability distribution with support of cardinality n, with a pre-processing phase that scales linearly with n.

The use of the Alias Method for Node2Vec incurs the "memory explosion problem" since the preprocessing phase for a second-order random walk on a graph with $|E|$ edges has a support whose cardinality is $\mathcal{O}\left(\sum_{e_{ij} \in E} \deg(j)\right)$, where $\deg(j)$ is the degree of the destination node of the edge $e_{ij} \in E$.

Therefore, the time and memory complexities needed for preprocessing make the Alias method impractical even on relatively small graphs. For instance, on the unfiltered Human STRING PPI graph (19.354 nodes and 5.879.727 edges) it would require 777 GB of RAM.

To avoid this problem, we compute the distributions on the fly. For a given source node v, our sampling algorithm applies the following steps:

1. computation of the un-normalized transition probabilities to each neighbour of v according to the provided *in-out* and *return* biases;
2. computation of the un-normalized cumulative distribution, which is equivalent to a cumulative sum;
3. uniform sampling of a random value between 0 and the maximum value in the un-normalized cumulative distribution;
4. identification of the corresponding index through either a linear scan or a binary search, according to the degree of the node v.

In step 2 the cumulative sum is computed by a SIMD routine that processes at once in CPU batches of 24 values (see Appendix A for more details). Moreover, when the length of the vector is smaller than 128, we apply a linear scan instead of a binary search because it is faster thanks to lower branching and better cache locality.

Specialized Random-Walks

Node2Vec is a second-order random-walk sampling method [52], whose peculiarity relies in the fact that the probability of stepping from one node v to its neighbours considers the preceding step of the walk More precisely, Node2Vec defines the un-normalized transition probability π_{vx} of moving from v to any direct neighbor x, starting at a previous step from node t, as a function of the weight w_{vx} on the edge connecting v and x (v, x), and a search bias $\alpha_{pq}(t, x)$:

$$\pi_{vx} = \alpha_{pq}\ (t, x)\ w_{vx}$$

The search bias $\alpha_{pq}(t, x)$ is defined as a function of the distance $d(t, x)$ between t and x, and two parameters p and q, called, respectively, the *return* and *in-out* parameters:

$$\alpha_{pq}(t, x) = \begin{cases} \frac{1}{p} & \text{if } d(t, x) = 0 \\ 1 & \text{if } d(t, x) = 1 \\ \frac{1}{q} & \text{if } d(t, x) = 2 \end{cases} \tag{2.1}$$

In figure 2.9 the node2vec schema for defining the search bias is schematized.

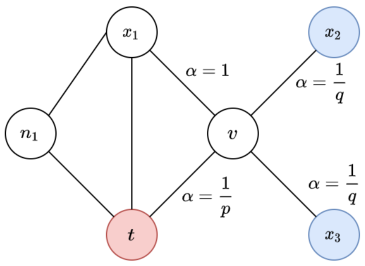


Figure 2.9: Illustration of the random walk procedure in node2vec. The walk just transitioned from t to v and is now evaluating on which node to step next. Edge labels indicate search biases α. The nodes in blue are at distance 2 from t, so that the edge connecting them to v has α equal to the explore (in-out) bias; blue nodes, at distance 1 from t, are connected to v by and edge with α equal to the return bias.

If the return parameter p is small, the walk will be enforced to return to the preceding node; if p is large, the walk will otherwise be encouraged to visit new nodes. The in-out parameter q allows to vary smoothly between Breadth First Search (BFS) and Depth First Search (DFS) behaviour. Indeed, when q is small the walk will prefer outward nodes, thus mimicking DFS; it will otherwise prefer inward nodes emulating in this case BFS. Since α must be recomputed at each step of the walk, the algorithm to compute it must be carefully designed to guarantee scalability.

The in-out bias can be re-formulated to allow an efficient implementation: starting from an edge (t, v) we need to compute $\beta_q(t, x)$ for each $x \in N(v)$, where $N(v)$ is the set of nodes adjacent to v including node v itself.

$$\beta_q(t, x) = \begin{cases} 1 & \text{if } d(t, x) \leq 1 \\ \frac{1}{q} & \text{otherwise} \end{cases} \qquad \Rightarrow \qquad \beta_q(t, x) = \begin{cases} 1 & \text{if } x \in N(t) \\ \frac{1}{q} & \text{otherwise} \end{cases}$$

This formulation (figure 2.10c) allows us to compute in batch the set of nodes X_β affected by the in-out parameter q:

$$X_\beta = \left\{ x \mid \beta_q(t,x) = \frac{1}{q}, q \neq 1 \right\} = N(v) \setminus N(t)$$

where $N(v)$ are the direct neighbors of node v. In this way, the selection of the nodes X_β affected by β_q simply requires computing the difference of the two sets $N(v) \setminus N(t)$.

X_β is efficiently computed by using a SIMD algorithm implemented in assembly, leveraging AVX2 instructions that work on node-set representations as sorted vectors of the indices of the nodes (see appendix B and figures B.3 and B.4). The algorithm is adapted from Lemire's et al. [68] SIMD algorithm for set intersection, which similarly works on sets represented as sorted arrays.

The return bias γ_p can be simplified as:

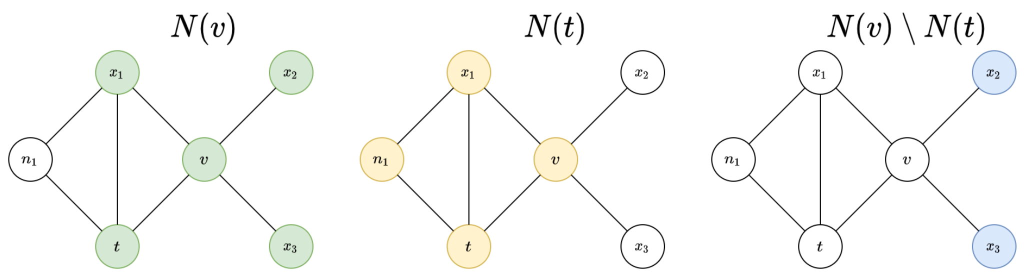
$$\gamma_p(t,x) = \begin{cases} \frac{1}{p} & \text{if } d(t,x) = 0 \\ 1 & \text{otherwise} \end{cases} \qquad \Rightarrow \qquad \gamma_p(t,x) = \begin{cases} \frac{1}{p} & \text{if } t = x \\ 1 & \text{otherwise} \end{cases}$$

It can be efficiently computed using a binary search for the node t in the sorted vector of neighbours. Summarizing, we re-formulated the transition probability π_{vx} of a second-order random walk in the following way:

$$\pi_{vx} = \beta_q(t,x)\gamma_p(t,v,x)w_{vx} \qquad \beta_q(t,x) = \begin{cases} 1 & \text{if } x \in N(t) \\ \frac{1}{q} & \text{otherwise} \end{cases} \qquad \gamma_p(t,v,x) = \begin{cases} \frac{1}{p} & \text{if } t = x \\ 1 & \text{otherwise} \end{cases}$$

If p, q are equal to one, the biases can be simplified, so that we can avoid computing them.

In practice, depending on the values of p, q and on the type of the graph (weighted or unweighted), *Ensmallen* provides eight different specialized implementation of the Node2Vec algorithm, detailed in table 2.2. This trick allows to significantly speed-up the computation, as showed by the empirical computational times reported in table 2.3, which were obtained by each of the specialized algorithms when computing 1000 random walks of length 100 on the KGCovid19 Graph, using an AMD Ryzen 9 3900x processor. For instance, in the base case ($p = q = 1$ and an unweighted graph) the specialized algorithm runs more than 100 times faster than the most complex one ($p \neq 1, q \neq 1$, weighted graph). Moreover, as expected, we observe that the major bottleneck is the computation of the in-out bias (see table 2.3).



(a) Neighbours of v **(b)** Neighbours of t **(c)** The left difference of Neighbours

Figure 2.10: Illustration of the equivalence with Leskovec formulation. The walk just transitioned from t to v and is now evaluating its next step out of node v. The green nodes are the neighbours of v, the yellow nodes are the neighbours of t, already computed in the previous step of the walk. The blue nodes are those effected by the in-out bias (also in figure 2.9), which may be computed as the difference of $N(v)$ and $N(t)$.

Table 2.2: Specialized first and second-order random walks algorithms: the 8 different algorithms, dynamically dispatched by the library according to the use case.

	$q = 1$		$q \neq 1$	
	$p = 1$	$p \neq 1$	$p = 1$	$p \neq 1$
Unweighted Graph	Unweighted first-order random walk	Unweighted second-order return-weight-only random walk	Unweighted second-order explore-weight-only random walk	Unweighted second-order random walk
Weighted Graph	Weighted first-order random walk	Weighted second-order return-weight-only random walk	Weighted second-order explore-weight-only random walk	Weighted second-order random walk

Table 2.3: Empirical Computational time required by the different optimized implementations for computing the transition probability π_{vx} listed in table 2.2. The computational time was measured when computing 1000 random walks of length 100 on the KGCovid19 Graph, using an AMD Ryzen 9 3900x processor.

q	p	Graph	π_{vx}	Time (ms)
$q = 1$	$p = 1$	Unweighted	1	0.46 ($\pm$0.01)
$q = 1$	$p = 1$	Weighted	w_{vx}	0.50 ($\pm$0.01)
$q = 1$	$p \neq 1$	Unweighted	$\gamma_p(t,x)$	13.8 ($\pm$0.08)
$q = 1$	$p \neq 1$	Weighted	$\gamma_p(t,x)w_{vx}$	14.2 ($\pm$0.07)
$q \neq 1$	$p = 1$	Unweighted	$\beta_q(t,x)$	45.8 ($\pm$0.5)
$q \neq 1$	$p = 1$	Weighted	$\beta_q(t,x)w_{vx}$	47.3 ($\pm$1)
$q \neq 1$	$p \neq 1$	Unweighted	$\beta_q(t,x)\gamma_p(t,x)$	47.7 ($\pm$0.2)
$q \neq 1$	$p \neq 1$	Weighted	$\beta_q(t,x)\gamma_p(t,x)w_{vx}$	49.0 ($\pm$0.3)

Approximated random walks

Since the computational time complexity of the sampling algorithm for either weighted or second-order random walks scales linearly with the degree of the considered source node, computing an exact random walk on graphs with high degree nodes (where "high" refers to nodes having an outbound degree larger than 10000) would be impractical, also considering that such nodes have a higher probability to be visited.

To cope with this problem, we designed an approximated random walk algorithm, where each step of the walk considers only a sub-sampled set of k neighbors, where the the degree threshold parameter k is set to a value significantly lower than the maximum node degree.

Figure 2.11 sketches the approximated random walk algorithm. **a** The random walk starts at a node, *src*, whose degree is 15 (the 15 neighbours of *src* are highlighted in cyan). **b.** If the degree threshold parameter is set to $k = 5$, the approximated random walk exploits the Sorted Unique Sub-Sampling algorithm (SUSS, described in section 2.3.2) to uniformly sample 5 distinct nodes in the neighborhood of *src*. The sub-sampled neighborhood is then used to randomly select the successor node where the random walk steps (edge highlighted with an arrow). **c.** The successor node becomes the novel source node, *src*, and the points **a** and **b** are repeated to choose a new successor. The process is iterated until the end of the walk.

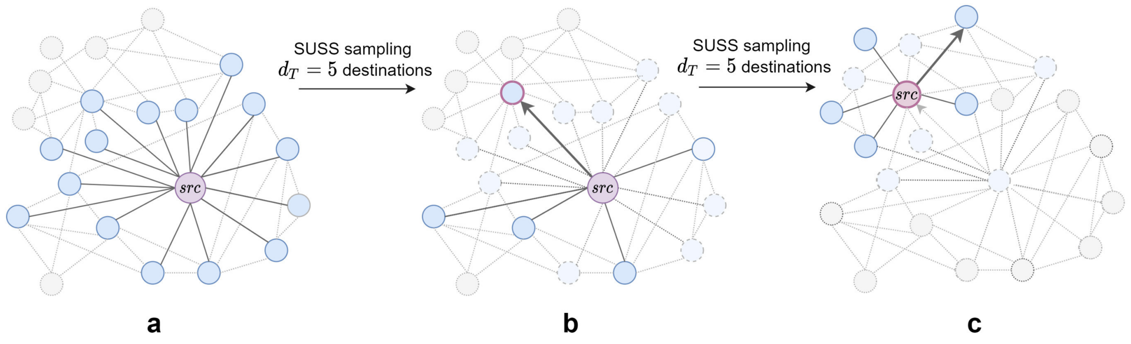


Figure 2.11: Approximated random walk sketch.

2.4 Edge-sampling and corrupted triple sampling methods.

GRAPE includes efficient Rust implementations of edge sampling methods such as LINE and TransE.

Moreover, a large set of corrupted-triple sampling models is integrated from the PyKeen library. The PyKeen library models are implemented in Pytorch and are generally implemented as approximations, with a loss of the form (note the addition of ℓ' and the lack of $ReLU$):

$$\mathcal{L}_{PyKeen} = \sum_{(v,\ell,s)} (v + \ell - s)^2 - \sum_{(v',\ell',s')} (v' + \ell' - s')^2$$

The integrated models include TransH, DistMult, HolE, AutoSF, TransF, TorusE, DistMA, ProjE, ConvE, CP, RESCAL, QuatE, TransD, ERMLP, CrossE, TuckER, TransR, PairRE, RotatE, ComplEx, and BoxE [3]. Since the variations between these models and their performance are often minimal and mostly relative to the choice of their hyper-parameters [62], it is particularly hard to clearly summarize their effective differences. Therefore, we refer to each of the original papers for the extensive explanation. The parameters used for the evaluation of node embedding models in *GRAPE* pipelines are available in the Supplementary Information S4.1.

2.5 Edge embedding methods

The *GRAPE* library offers a variety of methods for computing edge embeddings from node embeddings. Edge embeddings are vector representations of edges that combine the embeddings of the source and destination nodes. These embeddings can then be fed into a machine learning model for tasks such as edge prediction and edge or node classification.

In the previous sections, we have illustrated the many methods available in the *GRAPE* library to compute node embeddings, which we recall are low-dimensional vector representations of nodes in a graph and can capture the structure and properties of the graph. Edge embeddings are similar, but they represent the relationships between nodes, rather than the nodes themselves.

The available edge embeddings include concatenation, average, cosine distance, L1, L2, and Hadamard operators. The specific method used can be chosen by the user, who can set it through a parameter. We stress that different node embedding methods, tasks and classifier models may benefit for distinct edge embedding methods, and the selection of the optimal approach should be explored through adequate hyper-parameter section techniques for discrete options, such as Parzen Trees [12]

Concatenation is a method for combining the source and destination node embeddings into a single vector by concatenating their values. For example, if the source node embedding is [1, 2, 3] and the destination node embedding is [4, 5, 6], the concatenated edge embedding would be [1, 2, 3, 4, 5, 6].

Average is a method for computing the edge embedding by taking the average of the source and destination node embeddings. For example, if the source node embedding is [1, 2, 3] and the destination node embedding is [4, 5, 6], the average edge embedding would be [2.5, 3.5, 4.5].

Cosine distance is a method for computing the similarity between two vectors using the cosine of the angle between them. It is commonly used as a measure of similarity between node embeddings. This particular edge embedding method produces a scalar value representing the similarity between the two vectors.

L1 and L2 are methods for computing the distance between two vectors. The L1 distance, also known as the Manhattan distance, is the sum of the absolute differences between the elements of the two vectors. The L2 distance, also known as the Euclidean distance, is the square root of the sum of the squares of the differences between the elements of the two vectors. For example, if the source node embedding is [1, 2, 3] and the destination node embedding is [4, 5, 6], the L1 distance between them would be the sum of the absolute differences between the elements, which would be 9. The L2 distance between them would be the square root of the sum of the squares of the differences, which would be ≈ 3.6.

Hadamard is a method for computing the element-wise product of two vectors. For example, if the source node embedding is [1, 2, 3] and the destination node embedding is [4, 5, 6], the Hadamard product of the two vectors would be [4, 10, 18].

GRAPE provides three different implementations of the edge embedding procedure. The first implementation uses Keras and TensorFlow, and can be used in any Keras model. The second implementation uses NumPy, and the third implementation uses Rust. Rust is a programming language that is designed for performance and safety, and is often used for systems programming. It can be more efficient than other languages, such as a NumPy-based Python implementation, for certain operations.

In order to avoid running out of memory, the computation of edge embeddings is executed lazily for a subset of the edges at a time. This means that the edge embeddings are generated on the fly, rather than being stored in memory all at once. Lazy generation of edge embeddings is supported during training for some, but not all, of the supported edge and edge-label prediction models. It is supported for all models during inference.

Overall, *GRAPE* comes with a rich set of tools for computing edge embeddings from node embeddings. It offers a range of methods for computing edge embeddings, and provides efficient implementations using Keras, NumPy, and Rust, so to cover a wide variety of use cases. It also supports lazy generation of edge embeddings, which can help avoid running out of memory. This can be particularly useful for large graphs, where storing all of the edge embeddings in memory at once might not be feasible. By generating the edge embeddings on the fly, *GRAPE* allows for more efficient and scalable computation of edge embeddings.

2.6 Node-label, edge-label, and edge prediction models

GRAPE provides implementations to perform node-label prediction, edge-label prediction and edge prediction tasks.

All the models devoted to any of the three prediction tasks share the following implementation similarities. Firstly, they all implement the *abstract classifier interface* and therefore provide straightforward methods for training (*fit*) and inference (*predict* and *predict_proba*).

Secondly, all models are multi-modal, that is, they not only can receive the (user-defined) node/edge embedded representation, but also other embeddings computed in multiple ways and therefore carrying different semantics (e.g., topological node/edge embeddings or BERT embeddings). For edge prediction and edge-label prediction models, this also generalizes to multiple node-type features, which, if available, are concatenated to the considered node features, and to the possibility of computing traditional edge metrics (e.g. Jaccard, Adamic-Adar, and so on).

For each task, we make available at least eight models from the literature, adapted to the considered task: 5 are Scikit-learn-based models, namely Random Forest, Extra Trees, Decision Tree, Multi-Layer Perceptron (MLP), and Gradient Boosting. The remaining 3 are TensorFlow-based models, namely GraphSAGE [57], Kipf GCN [129] and a baseline GNN.

As per the node embedding models, custom and third party models can be integrated through task-specific Python abstract classes (Section 2.7).

Scikit-learn-based models make available all the parameters that are available in the Scikit version. TensorFlow-based models make available parameters to set the number of layers in each provided feature's sub-module and head module.

Visualizations of the Kipf GCN model for node-label, edge-label and edge prediction tasks are also available (see Appendix D).

All edge prediction models can be trained by sampling the graph negative edges by either following a uniform or a degree distribution; by default we set a degree distribution because it generally produces more informative negative-training sets, characterized by a smaller covariate-shift with respect to the positive-set. This approach still guarantees a negligible number of false negatives edges. The unbalance between positive and negative edges is also a free parameter which may be arbitrarily set: by default the models are trained using a balanced approach, that is we sample a number of negative edges equal to the number of positive edges.

In addition to the eight models presented in section 2.6, we also make available a multi-modal perceptron model implemented in Rust. This model, analogously to all other models, supports lazy computation of edge embedding and edge features, but does this in an extensively parallel manner with no additional memory requirement over the model weights. The model optimizer is Nadam. The Perceptron is a great baseline for comparison, given its rapid convergence, minimal hardware requirements (no GPUs nor significant RAM requirement), and competitive performance in many considered tasks. Such a model is essential to put into perspective the improvements achieved by more complex and often significantly more expensive models.

Parameters used for the evaluation of edge prediction models in *GRAPE* pipelines are available in the Supplementary Information S4.2.

All of the provided edge-label prediction models support binary and multi-class classification tasks. We currently lack support for multi-label classification tasks, which is being addressed.

All of the provided node-label prediction models support binary, multi-class and multi-label classification tasks. Parameters used for the evaluation of node-label prediction models in *GRAPE* pipelines are available in the Supplementary Information S4.3.

2.7 Pipelines for the comparative evaluation of different models across graph-prediction tasks

To provide actionable and reliable results, the fair and objective comparative evaluation of datasets, graph embedding, and prediction models is crucial and not only requires specifically designed and real-world benchmark datasets [58], but also pipelines that could allow non-expert users to easily test and compare graphs and inference algorithms on the desired graphs.

GRAPE provides pipelines for evaluating node-label, edge-label and edge prediction experiments trained on user-defined embedding features and by using task-specific evaluation schemas (Section 4.2.2).

All the implemented pipelines have integrated support for differential caching, storing the results of every step of the specific experiment, and for “smoke tests”, i.e. for running a lightweight version of the experimental setup with minimal requirements to ensure execution until completion before running the full experiment.

The pipelines can use any model implementing a standard interface we developed. The interface requires the model to implement methods for training (*fit* or *fit_ transform*), inference (*predict* and *predict_ proba*) plus additional metadata methods (e.g., whether to use node types, edge types, and others) which are used to identify experimental flaws and biases. As an example, in an edge-label prediction task using node embeddings, *GRAPE* will use the provided metadata to check whether the selected node embedding method also uses edge labels. If so, the node embedding will be recomputed during each holdout. Conversely, if the edge labels are not used in the node embedding method, it may be computed only once. The choice to recompute the node embedding for each holdout, which may be helpful to gauge how much different random seeds change the performance, is left to the user in this latter case.

To configure one of the comparative pipelines, users have to import the desired pipeline from the *GRAPE* library and specify the following modular elements:

Graphs The graphs to evaluate, which can be either graph objects or strings matching the names from graphs retrieval.

Graph normalization callback For some graphs it is necessary to execute normalization steps and filtering, such as the STRING weighted protein-protein interaction graphs which can e.g. to be filtered at 700 minimum edge weight, which is the threshold suggested by its authors. For this reason, users can provide this optional normalization callback.

Classifier models The classifier models to evaluate, which can either be a model implemented in *GRAPE* or custom models implementing the proper interface.

Node, node type, and edge features The features to be used to train the provided classifier models. These features can be node embedding models, either implemented in *GRAPE* or custom embedding models implementing the node embedding interface.

Evaluation schema The evaluation schema to follow for the evaluation. It can be, e.g., a Monte Carlo connected holdout, or a K-fold cross validation.

Given any input graph, each pipeline starts by retrieving it (if the name of the graph was provided) and validating the provided features (checking for NaNs, constant columns, compatibility with the provided graphs); next, and if requested by the user, it computes all the node-embeddings to be used as additional features for the prediction task. Once this preliminary phase is completed, the pipeline starts to iterate and generate holdouts following the provided evaluation schema.

For each holdout, *GRAPE* then computes the node embeddings required to perform the prediction task (such as topological node embeddings for a node-label prediction task, or topological node embeddings followed by their combination through a user-defined edge embedding operator - see Section 2.5 - to obtain the edge embedding in an edge-prediction task), so that a new instance of the provided classifier models can be fitted and evaluated (by using both the required embedding and, eventually, the additional, label-independent, features computed in the preliminary phase). The classifier evaluation is finally performed by computing an exhaustive set of metrics including AUROC, AUPRC, Balanced Accuracy, Miss-rate, Diagnostic odds ratio, Markedness, Matthews correlation coefficient and many others.

More details and ecamples about the usage of the evaluation pipelines are reported in chapter 4, section 4.2.2.

Chapter 3

Open graph processing libraries and datasets used in the experiments

The node embedding problem has been approached multiple times in the literature, and several packages and datasets are available to support a fair comparison between different methods.

This chapter provides a schematized picture of the available resources, focusing on those we used as benchmarks for comparison with *GRAPE*.

In section 3.1, we sketch the renowned packages mostly related to *GRAPE*. Next, we summarize all the datasets available through the *GRAPE* resource (section 3.2 and section 3.3), and we particularly focus on three real-world datasets used to (section 3.4) assess the edge and node-label prediction experiments of models implemented in *GRAPE*.

3.1 Related graph processing libraries

None of the libraries identified in the literature provide an implementation of either a Graph-based CBOW or SkipGram but relies on Gensim [102] CBOW and SkipGram Word2Vec model for the embedding procedure. The most performing libraries considered use Numba [66] just-in-time compilation to achieve better run-time executions. However, such a dependency and related ecosystem (e.g. llvmlite) can be very complex to properly install e prone to significant breaking changes between versions (e.g. between versions '0.4' and '0.5').

3.1.1 NetworkX

NetworkX [56] is a highly renowned Python language package for the exploration and analysis of networks. Even though this library does not provide fast first/second-order random walks, we include it in our comparisons since it is used to handle the graph structure in the GraphEmbedding and Node2Vec packages mentioned below.

3.1.2 GraphEmbedding

GraphEmbedding [25] is a Python package for embedding networks via random-walk-based methods such as Node2Vec. The graph is loaded using NetworkX. Within this package, the random walks are executed using the alias method [65] (see section C.1), whose complexity during the preprocessing phase hampers the computation of random walks to large graphs. To our knowledge, GraphEmbedding handles only undirected homogeneous graphs. The library relies on Gensim [102] for the embedding procedure.

3.1.3 Node2Vec

Node2Vec is a Python language package for embedding networks via random-walk-based methods such as Node2Vec [79]. The graph is loaded using NetworkX. As mentioned for GraphEmbedding, also Node2Vec makes is limited by the usage of the alias method to execute the random walks. To our knowledge, Node2Vec handles only undirected homogeneous graphs. The library relies on Gensim [102] for the embedding procedure.

3.1.4 iGraph

iGraph [32] is a library collection for creating and manipulating graphs and analyzing networks. While it was originally written in C, some implementations are presently available as Python and R packages. We did not use this library in our comparisons because it does not currently implement fast parallel first or second-order random walks, nor does it support their easy implementation. The iGraph library is not equipped with node embedding methods except for two baseline spectral Laplacian methods.

3.1.5 CSRGraph

CSRGraph[100] is a library to execute fast first and second-order random walks using Numba [66]. Unlike the previously mentioned libraries (e.g., GraphEmbedding or Node2Vec), CSRGraph does not store the graph into a NetworkX object but exploits a compressed sparse row (CSR) matrix, which significantly reduces the memory requirements with respect to methods exploiting NetworkX objects. Additionally, instead of using the alias method, it computes the node probabilities and samples them lazily, as required. To our knowledge, this library only handles undirected homogeneous graphs. The library relies on Gensim [102] for the embedding procedure.

3.1.6 PecanPy

PecanPy [72] is a library to execute fast first and second-order random walks based on a set of different solutions depending on the graph densities and node number. For graphs with less than 10000 nodes, PecanPy uses the alias method, while for larger graphs, it uses a strategy similar to the CSR library, where node probabilities and sampling are lazily computed, and the CSR data structure is used to store the edges. Finally, it switches the graph data structure to a dense adjacency matrix for graphs with edge densities larger than 10%. Analogously to CSRGraph, also PecanPy makes extensive use of Numba [66]. The library relies on Gensim [102] for the embedding procedure.

3.1.7 FastNode2Vec

FastNode2Vec [1] is a library to *lazily* execute fast first and second-order random walks. Analogously to CSRGraph and PecanPy, also FastNode2CVec makes extensive use of Numba [66] and relies on Gensim [102] for the embedding procedure. The significant difference between FastNode2Vec and the previously mentioned libraries, namely PecanPy and CSRGraph, is that the random walks are computed lazily and directly fed into the Gensim model with a small amount of overhead. Therefore it avoids the memory peak related to the rasterization of the random walks. This solution is very much analogous to what is done in GraPE, where the random walks are also computed lazily and fed into either the SkipGram or CBOW Rust models.

3.2 Datasets/Graphs directly accessible from *GRAPE*

Other than simply loading an arbitrary graph from tabular documents with the node and edge lists, *GRAPE* allows the retrieval of all the graphs included into the STRING repository [117] (56691 graphs), KGHub and KGOBO [101], Monarch Initiative [83], Linqs [48], PheKnowLator [22], and over 1000 graphs from Network Repository [105]. Once a graph is loaded, the library can compute an extensive HTML human readable report simply by displaying the object, which results in a well-formatted report within the context of a Jupyter Notebook. Additional tooling is made available to convert the report in LaTeX. The automatically generated reports include general statistics about the graph and data about singletons, node tuples, possible topological oddities, isomorphic node groups, trees, dendritic trees, stars and dendritic stars and tendrils eventually present in the graph.

3.3 Graphs used for *Ensmallen* assessment

This section presents the main features of the graphs used in previous publications to test *Ensmallen* vs state-of-the-art graph libraries. 44 graphs from Network Repository [105], having a considerably different number of nodes and edges have been collected as a benchmark for comparison; the other graphs are GiantTN provided by Zenodo, Homo sapiens being provided by STRING and KGCOVID19 provided by KGHub. Their main features are summarized in Table 3.1.

Table 3.1: Main features of the graphs considered as benchmark for comparison with *Ensmallen.*

Name	Nodes	Edges	Min degree	Max degree	Weights
HomoSapiens [117]	19566	11938498	0	7507	true
SocFlickr [105, 82]	513969	6380904	1	4369	false
SocFriendster [105]	65608366	3612134270	1	5214	false
SocBlogcatalog [105]	88784	4186390	1	9444	false
KGCOVID19 [101]	574215	36500884	0	122238	false
IMDB [105, 34]	896305	7564901	1	1590	false
BNHumanJung [105, 4]	975930	292218600	1	8009	false
BNFlyDrosophilaMedulla [105, 4]	1781	17927	1	927	false
BNMouseRetina [105, 4]	1076	181622	1	744	false
BNMacaqueRhesusBrain [105, 4]	242	6108	1	111	false
BioCeCx [105, 26]	15229	491904	1	375	true
BioCeGn [105, 26]	2220	107366	1	242	true
BioCeGt [105, 26]	924	6478	1	151	true
BioCeHt [105, 26]	2617	5970	1	44	true
BioCeLc [105, 26]	1387	3296	1	131	true
BioCePg [105, 26]	1871	95508	1	913	true
BioDmCx [105, 26]	4040	153434	1	362	true
BioDmHt [105, 26]	2989	9320	1	37	true
BioDmLc [105, 26]	658	2258	1	50	true
BioDmela [105, 26]	7393	51138	1	190	false
BioDrCx [105, 26]	3289	169880	1	497	true
BioHsCx [105, 26]	4413	217636	1	473	true
BioHsHt [105, 26]	2570	27382	1	149	true
BioHsLc [105, 26]	4227	78968	1	397	true
BioScCc [105, 26]	2223	69758	1	571	true
BioScGt [105, 26]	1716	67974	1	549	true
BioScHt [105, 26]	2084	126054	1	472	true
BioScLc [105, 26]	2004	40904	1	167	true
BioScTs [105, 26]	636	7918	1	66	true
BioCelegansDir [105, 37]	453	4065	1	238	false
BioCelegans [105, 37]	453	4050	1	237	false
BioDiseasome [105, 49]	516	2376	1	50	false
BioDmela [105, 112]	7393	51138	1	190	false
BioGridFissionYeast [105, 115]	2026	25274	1	439	false
BioGridFruitfly [105, 115]	7274	49788	1	176	false
BioGridHuman [105, 115]	9436	62364	1	308	false
BioGridMouse [105, 115]	1450	3272	1	111	false
BioGridPlant [105, 115]	1717	6196	1	71	false
BioGridWorm [105, 115]	3507	13062	1	523	false
BioGridYeast [105, 115]	6008	313890	1	2557	false
BioHumanGene1 [105, 10]	22283	24669643	1	7939	true
BioHumanGene2 [105, 10]	14340	18068388	1	7229	true
BioMouseGene [105, 10]	45101	28967291	1	8032	true
BioYeastProteinInter [105, 61]	1870	4480	1	56	false
BioYeast [105, 61]	1458	3896	1	56	false
GiantTN [72, 73]	25689	77809858	1	12384	true
WebWikipedia2009 [105]	1864433	9014630	1	2624	false

3.4 Graphs used for edge & node-label prediction experiments

In this section, we summarize the graphs/datasets used to assess the performance of the edge-prediction models (subsection 3.4.1, and appendix C.1) and the node-label prediction models (subsection 3.4.2, and appendix C.2). The results obtained by our comparative evaluation are reported in chapter 4.

3.4.1 Graphs used for the edge-prediction experiments

For the edge prediction experiments, the STRING's *Homo sapiens* and *Mus musculus* [118], and *Human Phenotype Ontology* (HPO) [64] graphs have been used. To describe them in Appendix C.1, we copied the graph reports computed by *Ensmallen*.

Homo sapiens and Mus musculus are two protein-protein interaction graphs representing the protein interactions within the two species. These interactions include direct (physical) and indirect (functional) associations; they stem from computational prediction, knowledge transfer between organisms, and interactions aggregated from other (primary) databases. Human Phenotype Ontology (HPO) [64] provides a standardized vocabulary of phenotypic abnormalities encountered in human disease. We note that the topology of interaction graphs, such as STRING graphs and curated ontologies, are vastly different.

3.4.2 Graphs used for node-label prediction experiments

For the node-label prediction experiments CiteSeer [48, 110], Cora [48, 110], and Pubmed [84] graphs have been used. To describe them in Appendix C.2 the following, we copied the graph reports automatically computed by *Ensmallen*.

In citation graphs such as these, the nodes represent papers while the edges represent citations between the various publications. The papers are labelled according to their category.

Note that while additional node features exist relative to the three graphs, these have been omitted as we have focused strictly on node embedding involving the graph topology.

Cora [110] consists of 2708 scientific publications classified into one of seven classes. The citation network consists of 5429 edges. CiteSeer [104] consists of 3312 scientific publications classified into one of six classes. The citation network consists of 4732 edges. Pubmed Diabetes [84] consists of 19717 scientific publications from the PubMed database about diabetes classified into one of three classes. The citation network consists of 44338 edges.

3.5 Large Graphs used for assessing GRAPE against complex, real-world problems

To empirically show that *GRAPE* allows obtaining translational results in different fields, we built three large graphs using one dataset used in the context of, e.g., social-network analysis (English Wikipedia graph), one biological dataset used in the context of, e.g., drug repurposing (CTD dataset), and the PheKnowLator software resource, which has been developed to semi-automatically build knowledge graphs in the context of, e.g., disease predictions. In the following, we report a more detailed description of the three resources.

English Wikipedia. Wikipedia graphs are web graphs with nodes representing either Wiki sites pages or related websites; edges represent the links between the pages. In the experiments, we used the English Wikipedia graph having 17 million nodes and 130 million (undirected) edges (2021-11-01 version). The task for the English Wikipedia graph is a whole-graph edge prediction, that is, predicting whether an edge connects two given nodes in the entire graph. The set of positive edges is defined as the undirected edges present in the entire graph (about 130 million). The set of negative edges is instead defined as the edges that are not present in the graph, around 150 trillion undirected edges.

Comparative Toxicogenomic Database (CTD). CTD is a publicly available database that aims to advance understanding of how environmental exposures affect human health. It provides manually curated information about chemical–gene/protein interactions, chemical–disease, and gene-disease relationships. Also, it includes information about phenotypes, pathways, ontologies, and their relations with genes, chemicals, and diseases, including about 45 million edges and more than $100K$ nodes. The CTD edge-prediction task consists in predicting *gene-disease* associations. The set of positive edges is defined as the set of all the existing (undirected) relationships (edges) between gene and disease nodes, which amounts to about 29 million. Negative edges were defined by pairs of gene-disease being unrelated in the CTD dataset (about 362 million "negative" edges).

PheKnowLator biomedical data. PheKnowLator is a software resource designed to construct large-scale biomedical knowledge graphs using several knowledge models (instance-based and subclass-based). PheKnowLator currently integrates 12 Open Biomedical Ontologies and 31 linked open-data sources. In our experiments, we used a 2022-04-11 build including about 7 millions of (undirected) edges and about $800K$ nodes. The PheKnowLator task consists in the prediction of *genetic variant-disease* associations, using $44K$ known "positive" associations and a set of "negative" edges, including about 3 billions of variant-disease pairs having no known associations. Detailed information about the source data and scripts used to generate the above three big real-world graphs are available in Supplementary Section ✓S6.

Chapter 4

Results

Designed to leverage succinct data structures [38] (chapter 2), *GRAPE* loads real-world graphs composed of millions of nodes and billions of edges by requiring only a fraction of the memory required by other libraries (section 4.1), and guarantees average constant-time rank and select operations (sections 2.2.3, 2.2.5 and table 2.1). This makes it possible to execute many graph processing tasks, e.g. accessing node neighbours and running first- and second-order random walks, with memory usage close to the theoretical minimum. Among the many high-performance algorithms it provides, the library implements efficient *approximated* weighted DeepWalk & Node2Vec embedding models (sections 2.3.2). They can process graphs containing high-degree nodes (degree $> 10^6$), an otherwise unmanageable task when using the analogous exact algorithms, and allow one to obtain edge-prediction performance comparable to those achieved by using the *exact* version (Section 4.1.1).

In this section we report the results obtained by the experiments we run to assess the efficiency and effectiveness of *GRAPE* and to compare it with state of the art libraries. In more detail, we used the evaluation pipelines (Section 2.7) to compare the edge prediction and node-label prediction performance of 16 node embedding models, 12 reimplemented in *GRAPE* and 4 integrated from the KarateClub library [106]. Moreover, we compared *GRAPE* with state-of-the-art graph-processing libraries across several types of graphs having different size and characteristics, including big real-world graphs such as *Wikipedia*, the *CTD*, Comparative Toxicogenomic Database [33] and biomedical Knowledge Graphs generated through *PheKnowLator* [23], showing that *GRAPE* achieves state-of-the-art results in processing big real-world graphs both in terms of empirical time and space complexity and prediction performance.

4.1 Fast error-resilient graph loading

The challenge addressed in this section is the efficient loading of large graphs while simultaneously checking for common format errors. Previous state-of-the-art graph processing libraries have struggled with this task, often requiring large amounts of memory and time to load graphs.

GRAPE can process many graph formats and simultaneously check for common format errors. Fig. 4.1 shows the empirical space (a) and time complexity (b) required by *GRAPE* and by state-of-the-art graph processing libraries, including NetworkX [56], iGraph [32], CSRGraph, PecanPy [72], when loading 44 real-world graphs.

Results show that *GRAPE* is faster and requires less memory as compared to state-of-the-art libraries. For instance, *GRAPE* loads the *ClueWeb09* graph (1.7B nodes and 8B undirected edges) in less than 10 minutes and requires about 60GB of memory, whereas the other libraries were not able to load this graph. All graphs and libraries used in these experiments are directly available from *GRAPE*'s and are detailed in chapter 3, sections 3.1 and 3.3.

4.1.1 *GRAPE* outperforms state-of-the-art libraries on random walk generation

This section addresses the issue of efficiently computing random walks in large graphs. Previous state-of-the-art libraries have struggled to compute random walks in a timely and space-efficient manner, often requiring significant computational resources and time.

Through extensive use of thread and SIMD parallelism and specialized quasi-succinct data structures, *GRAPE* outperforms state-of-the-art libraries by one to four orders of magnitude in the computation of random walks, both in terms of empirical computational time and space requirements (Figure 4.1-c, d, e, f and Section 4.2).

Further speed-up of second-order random walk computation is obtained by dispatching one of the 8 optimized implementations of *Node2Vec* sampling [52] (Section 2.3.2). The dispatching is based on the values of the *return* and *in-out* parameters and on the type of the graph (weighted or unweighted). *GRAPE* automatically provides the version best suited to the requested task, with minimal code redundancy (Section 2.3.2). The time performance difference between the least and the most computationally expensive implementations is around two orders of magnitude (tables 2.2 and 2.3).

4.2 Experimental comparison of graph processing libraries.

The challenge addressed in this section is to evaluate the efficiency and effectiveness of *GRAPE* and compare it with state-of-the-art libraries on a variety of tasks. Previous state-of-the-art libraries often struggle to perform graph processing and node embedding on large graphs due to the high time and space complexity of these algorithms. *GRAPE* introduces a novel approach to approximated random walks, which allows it to achieve significantly faster performance on large graphs while maintaining comparable accuracy to the exact methods.

The major improvement of *GRAPE* is its ability to efficiently compute random walks on large graphs that would be infeasible to compute using previous state-of-the-art libraries. This is achieved through the use of specialized quasi-succinct data structures and approximated random walk methods. The results of our experiments show that *GRAPE* outperforms state-of-the-art libraries by one to four orders of magnitude in terms of both time and space complexity.

We compared *GRAPE* with a set of state-of-the-art libraries including GraphEmbedding, Node2Vec, CSRGraph and PecanPy [72], on a large set of first and second-order random walk tasks. The random walk procedures in the GraphEmbedding and Node2Vec libraries use the alias method (Appendix C). The PecanPy library also employs the alias method for small graphs use-cases (less than $10,000$ nodes). CSRGraph, on the other hand, computes the random walks lazily using Numba [66]. Similarly, PecanPy leverages Numba lazy generation for graphs having more than 10,000 nodes. All libraries are further detailed in chapter 3, section 3.1.

Figure 4.1 shows the experimental results of a complete iteration of one-hundred step random walks on all the nodes across 44 graphs having a number of edges ranging from some thousands to several billions (Section 4.1).

For properly measuring the peak memory usage and the time requirements of both *Ensmallen* and the other state-of-the-art libraries used as benchmark for comparison, we created an additional thread for logging purposes. We have measured the used memory by reading `/proc/meminfo`, which makes five different metrics available:

MemTotal RAM installed on the system

MemFree RAM not used

Buffers RAM used for I/O buffers

Cached RAM used for dirty pages and ramdisks

Slab RAM used by the kernel

We define the memory in use as: MemInUse = MemTotal − MemFree − Buffers − Cached − Slab.

We executed *Ensmallen* and all the benchmarks on a dedicated server with no significant running process, except the sshd service we use to connect to it. To obtain a truthful evaluation of the memory usage required to execute a specific task, we have logged the average memory usage before the task starts, and we have subtracted such value from the memory usage we measure during the task execution. Though more accurate methods exist, e.g. jemalloc [42], Valgrind [86], hooking malloc using `__malloc_hook` or using `LD_PRELOAD` to hook the malloc function, the method we implemented is precise enough to detect significant differences.

We designed the tracker to have a linearly increasing delay between measurements because we have to measure tasks that might take from few microseconds to hours: long-running tasks would otherwise log too much data and start using Gigabytes of RAM. To further reduce this problem, we log the values in a constant size buffer; when the task finishes or the delay between two consecutive measurements is significantly longer than the time necessary to write the log to disk, we dump the log to a file.

The results of our experimental analysis showed that *GRAPE* greatly outperforms all the compared graph libraries on both first and second-order random walks in terms of both space and time complexity. Note that *GRAPE* scales well with the biggest graphs considered in the experiments, while the other libraries either crash when exceeding 200GB of memory, or take more than 4 hours to execute the task (Figure 4.1 c, d, e, f).

Approximated random walks to process graphs with high-degree nodes

The challenge addressed in this section is the computation of weighted and/or second-order random walks on graphs containing high-degree nodes, which can be computationally intensive. Previous state-of-the-art methods, which have primarily relied on the alias method to sample node neighbors, do not scale well to graphs with nodes with high-degree. *GRAPE* introduces an approximated implementation of weighted and/or second-order random walks that undersamples the node neighbors to allow for the efficient computation of random walks on graphs with high-degree nodes. This approach has been shown to be both efficient and effective, as demonstrated by experimental results on real-world graphs.

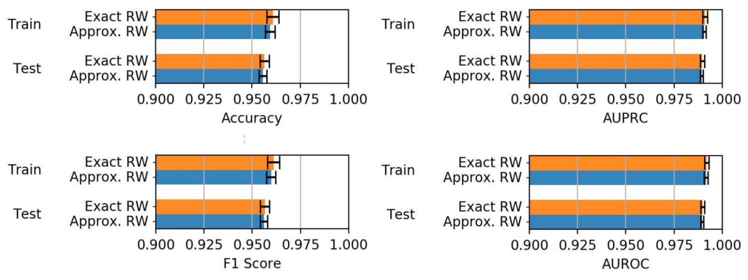
Random walks on graphs containing high-degree nodes is challenging, since multiple paths from the same node need to be processed. To overcome this computational burden, *GRAPE* provides an approximated implementation of weighted random walks that undersamples the neighbors to scale with graphs containing nodes with high-degree, e.g. with millions of neighbors (Figure 2.11 a, b, c, Section 2.3.2).

Since neighborhood undersampling may decrease the informativeness of the obtained embeddings, therefore having a negative impact on the following graph-predictions tasks, we firstly assessed the robustness of the proposed approximation by the (unfiltered) *H. sapiens* STRING PPI network [117] graph to compute Node2Vec and SkipGram-based embeddings that exploited both exact and approximated random walk samples, where the maximum degree threshold for the approximated random walks was set to a deliberately low threshold (10). The computed (approximated and exact) embedded samples where then used to train an MLPclassifier for an edge prediction task.

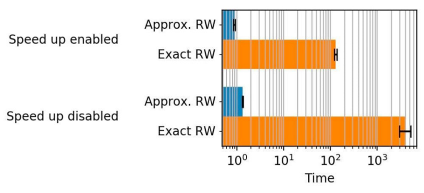
The edge prediction performance on the train and test sets were evaluated by computing the average and standard deviations of the accuracy, AUPRC, F1, and AUROC scores over 30 holdouts. Figure 4.2-a shows the obtained results, which were comparable, as per the Wilcoxon rank-sum test (p-value > 0.2).

Note that the maximum node-degree in the training set ranged between 3325 and 4184 across the holdouts, which is more than $300 - 400$ times higher than the degree threshold (10)) set for the approximated random walks.

For testing the efficiency of the approximated random walks, we used the *sk-2005* [15] graph and we computed random walks with 100 hops from 100 randomly chosen nodes. This procedure was repeated 10 times; figure 4.2-b shows the mean and the standard deviation of the elapsed time.



a



b

Figure 4.2: Approximated random walk. a. Edge prediction performance comparison using random walk samples obtained with exact and approximated methods. The mean and standard deviations of the achieved performance across the 30 holdouts are shown both for the training and the test set. Bar plots are zoomed-in at 0.9 to 1.0. **b.** Empirical time comparison (in msec) of the approximated and exact second-order random walk algorithm on the *sk-2005* [15] graph: Time is on a logarithmic scale.

4.2.1 Node and edge embedding models

GRAPE provides GRL methods (sections 2.3.1, 2.3.2, and 2.4) and prediction models (section 2.6) with high-performance implementations for kernel preprocessing as well as generation of random-walk and triple mini-batches.

GRAPE provides both its own implementations and Keras-based implementations for all shallow neural network models (e.g. CBOW, SkipGram, TransE). Nevertheless, since shallow models allow for particularly efficient data-race aware and synchronization-free implementations [132], the "by-scratch" *GRAPE* implementations significantly outperform the Keras-based ones, as TensorFlow APIs are too coarse and high-level for such fine-grained optimizations. While GPU training is available for the TensorFlow models, their overhead with shallow models tends to be so relevant that "by-scratch" CPU implementations outperform those based on GPU. Moreover, the embedding of large graphs (such as Wikipedia) do not fit in most GPU hardware memory. Still, Keras-based models allow users to experiment with the open-software available in the literature for Keras, including, e.g., advanced optimizer and learning rate scheduling methods.

The provided spectral and matrix-factorization-based models, including HOPE [90], NetMF [99] and their variations (GLEE [122], SocioDim [120]), compute the (generally sparse) weighted adjacency matrix in parallel and then rely on LAPACK [5] routines for the singular values decomposition and eigenvectors computations. Lastly, *GRAPE* also provide Keras/TensorFlow-based first-order and second-order LINE models, as they generally suffer from noisy gradients and therefore require higher-order optimisers such as Adam. Notable third parties libraries integrated are Sciki-Learn [91], PyKeen [3] and KarateClub [106].

Furthermore, *GRAPE* provides many methods to compute edge embeddings given some computed node embedding, ranging from concatenation, Hadamard (element-wise multiplication), the element-wise difference in L_1 or L_2 norm, and element-wise mean, subtraction or sum between the node embedding vectors. The library also comes equipped with tools to visualize the computed node and edge embedding and their properties, including edge weights, node degrees, connected components, node types and edge types. For example, in figure 2.1 **c** we display the node (left) and edge types (center) of the KG-COVID19 graph and whether sampled edges exist (right) by using the first two components of the *t-SNE* decomposition of the node/edge embeddings [124].

4.2.2 GRAPE enables a fair and reproducible comparison of graph embedding and graph-based prediction methods

GRAPE provides both a large set of ready-to-use graphs that can be used to run comparative evaluation experiments, and standardized pipelines (section 2.7) to fairly compare different models and graph libraries ensuring reproducibility of the results (Fig. 2.1 b). Graph embedding are efficiently implemented in Rust by scratch (with a Python interface) or are integrated from other libraries by implementing the interface methods of an abstract *GRAPE* class (section 2.7). *GRAPE* users can compare different embedding methods and prediction models and can also add their own methods to the standardized pipelines. Our experiments show how to use the standardized pipelines to fairly compare a large set of methods and different implementations by using only a few lines of Python code.

FAIR graph retrieval

GRAPE facilitates *FAIR* access to an extensive set of graphs and related datasets, including both commonly used benchmark datasets and graphs actively used in biomedical research. Any of the available graphs can be retrieved and loaded with a single line of Python code (Fig. 2.1 **b.**), and their list is constantly expanding, thanks to the generous contributions of *GRAPE* users. The list of resources currently supported can be found at Supplementary Information S3.1.

Findability and Accessibility. Datasets may change locations, versions may appear in more than one location, and file formats may change. Using an ensemble of custom web scrapers, we collect, curate and normalize the most up-to-date datasets from an extensive resources list (currently over $80,000$ graphs). The collected metadata is shipped with each *GRAPE* release, ensuring end-users can always find and immediately access any available versions of the provided datasets.

Interoperability. The graph retrieval phase contains steps that robustly convert data from (even malformed) datasets into general-use TSV documents that, while primarily used as graph data, can be used for any desired application case.

Reusability. Once loaded, the graphs can be arbitrarily processed and combined, used with any of the many embedding and classifier models from either the *GRAPE* library or any third-party model integrated in *GRAPE* by implementing the interface described in section 4.2.2.

FAIR evaluation pipelines

The fair and objective comparative evaluation of datasets, graph embedding and prediction models is fundamental for scientific research. To provide actionable results, this comparison requires specifically designed and real-world benchmark datasets [58], as well as pipelines that could allow non-expert users to easily test and compare graphs and inference algorithms on the desired graphs.

Beside FAIR graphs, *GRAPE* allows even users with minimal Python language experience to implement experimental designs through pipelines for running node-label, edge-label and edge prediction experiments with task-specific evaluation schemas.

More precisely, each pipeline allows users to tailor the experiment by choosing: (a) the set of graphs to be used in the experiments, (b) the functions to be called for graph filtering (if needed), (c) the set of (embedding and prediction) algorithms to be applied, and (d) the evaluation schema (Section 2.7).

In particular, the evaluation schema for edge prediction models are K-fold cross-validations, Monte Carlo, and Connected Monte Carlo (Monte Carlo designed to avoid the introduction of new connected components in the training graph) holdouts. All of the edge prediction evaluation schemas may sample the edges in a uniform or stratified way, with respect to a provided list of edge-types. Sampling of negative (non-existing) edges may be executed by either following a uniform or a scale-free distribution. Furthermore, the edge-prediction evaluation may be performed by using varying unbalance ratios (between existent and non-existent edges) to better gauge the true-negative rate (specificity) and false-positive rate (fall-out). Stratified Kfold and stratified Monte Carlo holdouts are also provided for node and edge-label prediction models.

For all tasks, an exhaustive set of evaluation metrics are computed, including AUROC, AUPRC, Balanced Accuracy, Miss-rate, Diagnostic odds ratio, Markedness, Matthews correlation coefficient and many others.

Each pipeline can receive as input any model that implements the specific task's interface (a Python abstract class). All interfaces follow the familiar scikit-learn style, with embedding models required to implement the *fit_transform* method, and prediction models required to implement the *fit*, *predict*, *predict_proba* methods, plus some additional metadata necessary to check for biases in the considered task. Interfaces are made available for embedding models, node-label prediction, edge-label prediction, and edge prediction. All models available in *GRAPE* implement these interfaces, and they can be used as starting points for custom integrations. Many usage examples are available in the library tutorials: https://github.com/AnacletoLAB/grape/tree/main/tutorials.

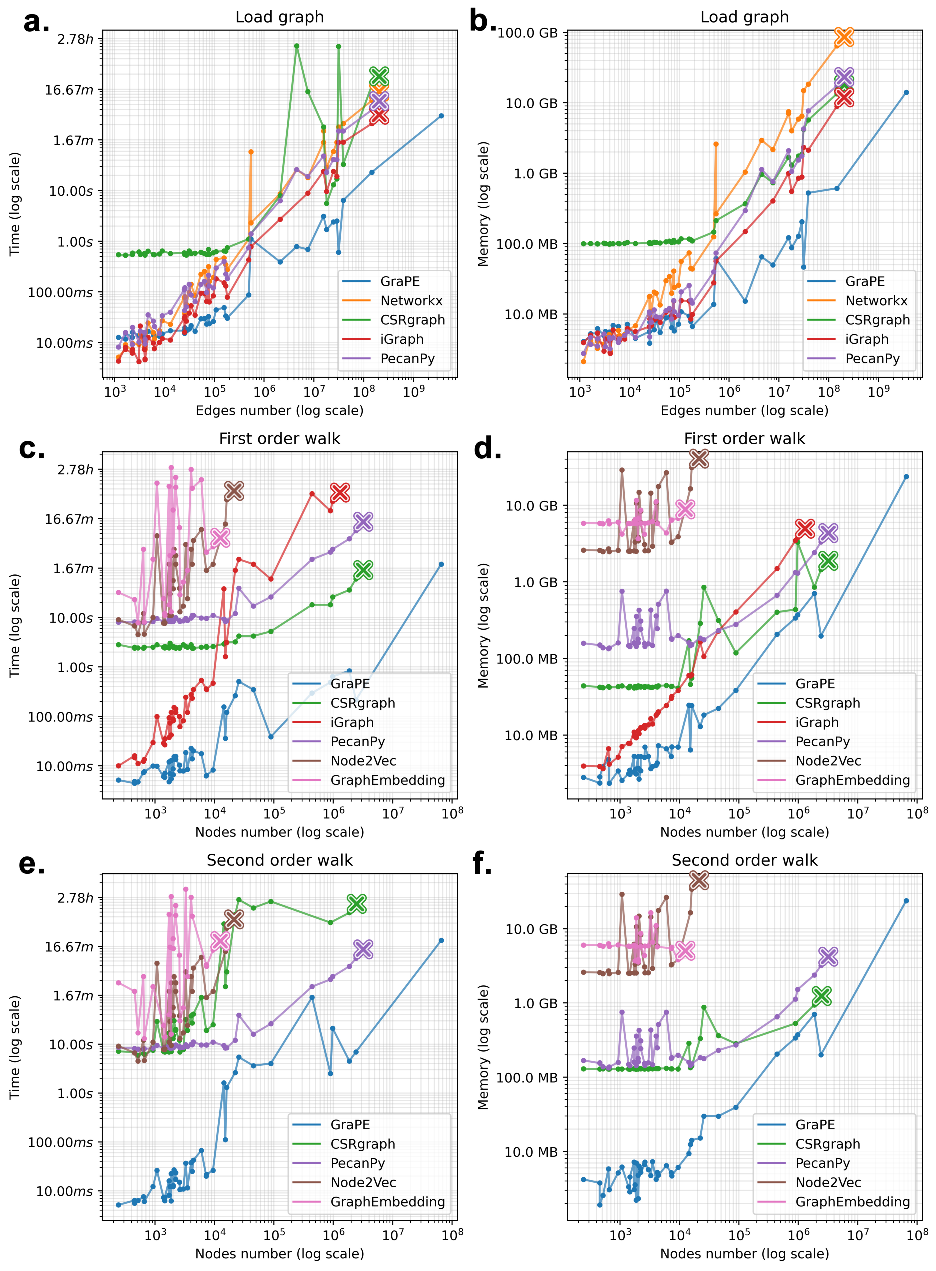


Figure 4.1: Experimental comparison of *GRAPE* with state-of-the-art graph processing libraries across 44 graphs. Top row - graph loading: **a.** Empirical execution time. **b.** Peak memory usage. The horizontal axis shows the number of edges, vertical axis peak memory usage. **Middle row - first-order random walk**: **c.** Empirical execution time. **d.** Peak memory usage. **Bottom row - second order random walk**: **e.** Empirical execution time. **f.** Peak memory usage. The horizontal axis shows the number of nodes, and the vertical axis respectively execution time (c,e) and memory usage (d,f). All axes are in logarithmic scale. The $\times$ represent when either a library crashes, exceeds 200GB of memory or takes more than 4 hours to execute the task. Each line corresponds to a graph resource/library, and points on the lines refer to the 44 graphs used in the experimental comparison. Note that the blue line representing *GRAPE* is always below all the other lines.

Experimental comparison of node embedding methods

We selected 16 among the 61 node embedding methods available in *GRAPE*, and we used the edge prediction and node-label standardized prediction pipelines to compare the prediction results obtained by Perceptrons, Decision Trees, and Random Forests classifiers (Fig. 4.3). For the edge prediction tasks we used the Hadamard product to construct edge embeddings from node embeddings. We applied a "connected Monte Carlo" evaluation schema for edge prediction and a stratified Monte Carlo evaluation schema for node-label prediction (Appendix H.2).

The models have been tested on 3 graphs for edge prediction (Fig. 4.3-a,b) and 3 graphs for node-label prediction (Fig. 4.3-c,d). The graph reports, describing the characteristics of the analyzed graphs, automatically generated with *GRAPE*, are available in Appendix C.1 and C.2. Since they are homogeneous graphs[1] we considered only homogeneous node embedding methods. Moreover, we discarded non-scalable models, e.g. models based on the factorization of dense adjacency matrices.

Among the 16 methods, 12 are implemented in *GRAPE* (purple in Fig. 4.3) and 4 have been integrated from the Karate Club library [106] (cyan in Fig. 4.3). They can be grouped into four broad classes:

a. Spectral and matrix factorization methods: Geometric Laplacian Eigenmap Embedding (GLEE) [122], Alternating Direction Method of Multipliers for Non-Negative Matrix Factorization (NMFADMM) [116], High-Order Proximity preserved Embedding (HOPE) [90], Iterative Random Projection Network Embedding (RandNE) [133], Network Matrix Factorization (NetMF) [99], and Graph Representations (GraRep) [24].

b. First-order random-walk methods: DeepWalk-based GloVe, CBOW, and SkipGram, Walklets SkipGram [96, 92, 80], and Role2Vec with Weisfeiler-Lehman Hashing [2, 111, 106].

c. Second-order random-walk methods: Node2Vec-based GloVe, CBOW, and SkipGram [92, 80, 52].

d. Triple-sampling methods: first and second order LINE [119].

All the embedding methods and classifiers are described in more detail in sections 2.3.1, 2.3.2, 2.4, and 2.6.

Results show that no model is consistently better with respect to the others across the types of task and the data sets used in the experiments (Figure 4.3). These results are analogous to those obtained by Kadlec et al. [62] for TransE model family, and those obtained by Errica et al [41] for GNN models, highlighting the need for objective pipelines to systematically compare a wide array of possible methods for a desired task. The standardized pipelines implementing the experiments are available from the online *GRAPE* tutorials and allow the full reproducibility of the results summarized in Fig. 4.3. Full results relative to other evaluation metrics are available in Appendix F.

[1]Hereafter, graph homogeneity/heterogeneity refers to the homogeneity/heterogeneity of node and edge types.

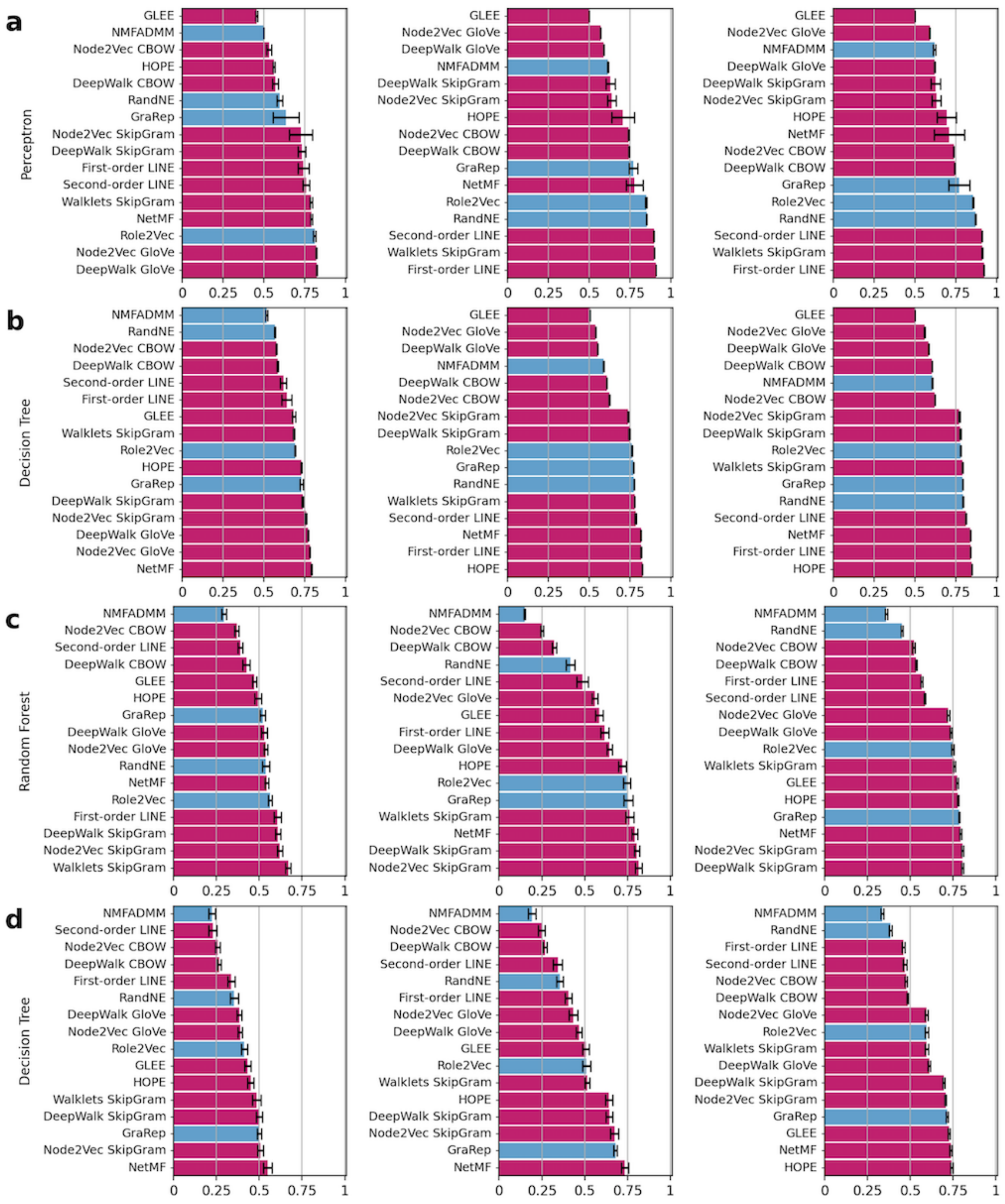


Figure 4.3: Comparison of embedding methods through the *GRAPE* pipelines: edge and node label prediction results. Results represent the balanced accuracy averaged across ten holdouts (results relative to other evaluation metrics are available in the Supplementary Information S5.). We sorted the embedding models by performance for each task; methods directly implemented in *GRAPE* are in purple, while integrated methods are in cyan. **(a, b)**: Edge prediction results obtained through a Perceptron (a) and a Decision tree (b). Barplots from left to right, show the balanced accuracy results obtained with the *Human Phenotype Ontology* (left), STRING *Homo sapiens* (center) and STRING *Mus musculus* (right). **(c, d)**: Node-label prediction results obtained through a Random Forest (c) and a Decision Tree (d). Barplots from left to right show the balanced accuracy respectively achieved with *CiteSeer* (left), *Cora* (center) and *PubMed Diabetes* (right) datasets.

4.2.3 Scaling with big real-world graphs

To show that *GRAPE* can scale and boost edge prediction in big real-world graphs, we compared its Node2Vec-based models with state-of-the-art implementations on three big graphs: 1) English Wikipedia; 2) Comparative Toxicogenomic Database (CTD [33]); 3) A biomedical graph generated through PheKnowLator [23]. Details about the three graphs are reported in chapter 3, section 3.5 and in Appendix G.

Experimental setup

Graph libraries compared in the experiments. In the experiments we used two *GRAPE* implementations of embedding algorithms: CBOW and SkipGram. We compared them with the following state-of-the-art embedding libraries, widely used by the scientific community:

- *PecanPy* [72] is a Python library implementing a Numba-based version of *node2vec*, leveraging Numba's just-in-time Python compilation [66] to generate the random walks on the input graph, and forwarding them to an embedding model provided by Gensim natural language processing library [102].
- *NodeVectors*[2] is a Python package that enables fast and scalable node embedding algorithms. It leverages CSR matrix storage for graphs, but it also support NetworkX [56] graph loading. Besides *node2vec*, the library also implements several kinds of first and second-order random walks.
- *SNAP* [70], Stanford Network Analysis Platform, is a general-purpose system for the manipulation and analysis of large networks, written in C++. Once compiled, it becomes an executable to analyze and compute different statistics about the graphs; it also implements different kinds of graph-processing algorithms and allows computing node embeddings, by using a pre-processing phase for pre-computation of transition probabilities through the Alias method [65].
- *Node2Vec*[3] is a Python package for embedding networks through random walk-based algorithms like *node2vec*. Similar to SNAP, it employs the Alias method [65] to pre-compute transition probabilities. It also handles the graph loading through the NetworkX library [56].
- *GraphEmbedding*[4] is a Python package that handles network embeddings with random walk-based methods. Again, the transition probability is pre-computed via the Alias method and employs NetworkX library to handle the loading of a graph.
- *FastNode2Vec*[5] implements the *node2vec* algorithm, leveraging both Numba and Gensim. This implementation scales linearly, in time and memory, with respect to the dimension of the input graph.

Evaluation of the results For all of the considered tasks, we firstly computed the embedded graphs using graph libraries and then the resulting embeddings have been used to train machine learning methods for an edge prediction problem. To evaluate the ML models we adopted a connected Monte Carlo (Appendix H) repeated ten times,

[2] `https://github.com/VHRanger/nodevectors`
[3] `https://github.com/eliorc/node2vec`
[4] `https://github.com/shenweichen/GraphEmbedding`
[5] `https://github.com/louisabraham/fastnode2vec`

with a train:test ratio equal to 80% : 20% of the data. As evaluation metrics we applied precision, recall, accuracy, balanced accuracy, F1, AUROC, and AUPRC. In the experimental set-up we imposed the following memory and time constraints, using a Google Cloud VM with 64 cores[6]:

- A maximum time of 48 hours for each holdout to produce the embedding;
- The maximum memory usage allowed during the embedding phase is 64GB.
- The maximum memory usage allowed during the prediction phase is 256GB.

To keep track of memory and time requirements and of possible stops for exceptions and system-related errors (out of memory, core dumps), the Python library `memory_time_tracker` was used[7].

[6]N1 Cpus with Intel Haswell micro-architecture

[7]`https://github.com/LucaCappelletti94/memory_time_tracker`

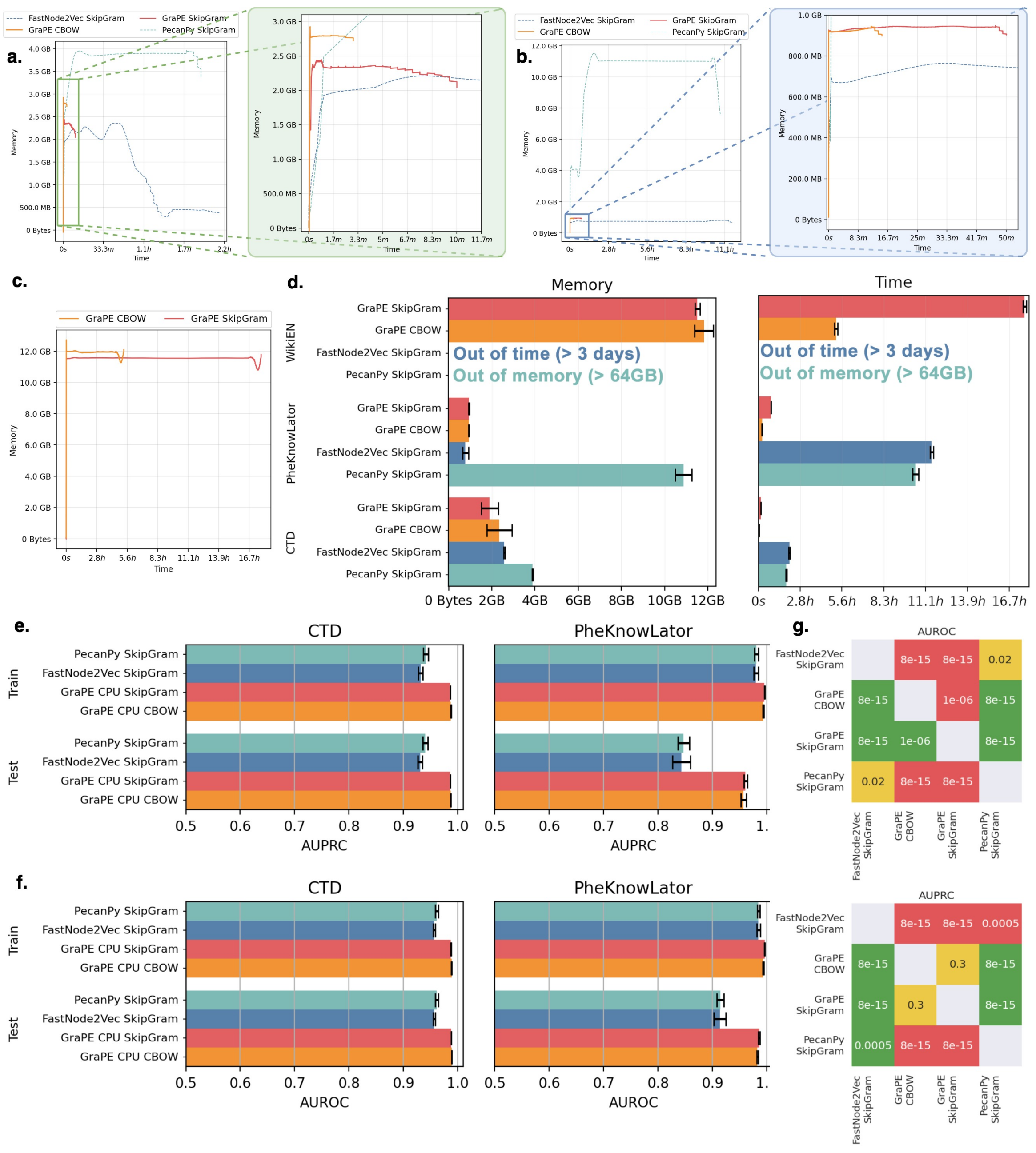


Figure 4.4: Performance comparison between *GRAPE* and state-of-the-art implementations of Node2Vec on real-world big graphs. *GRAPE* implementations achieve significantly better empirical time complexity: (**a.**), (**b.**) and (**c.**) show the worst performance (maximum time and memory, denoised using a Savitzky–Golay filter) over 10 holdouts on *CTD*, *PheKnowLator* and *Wikipedia*, respectively. In textbfa. and textbfb. the rectangles in the left figure are magnified in the right figure to highlight *GRAPE* performances. In the *Wikipedia* plot (**c.**) only *GRAPE* results are available as the others either go out-of-time or out-of-memory. (**d.**) Average memory and computational time across the holdouts; error bars represent standard deviation. (**e.**) AUPRC and (**f.**) AUROC results of Decision Trees trained with different graph embedding libraries: *GRAPE* embedding achieve better edge prediction performance than those obtained by the other libraries. (**g.**) Wilcoxon signed-rank tests results (p-values) between *GRAPE* and the other state-of-the-art libraries, where the win of a row against a columns is in green, the tie in yellow, and the loss in red). Top: AUROC, bottom: AUPRC.

***GRAPE* is able to scale with big graphs when the other competing libraries fail.** Most of the competing libraries were not able to complete the embedding and prediction tasks on big real-world graphs. Indeed *NodeVectors* exceeded the time computation limit, while *SNAP*, *Node2Vec*, and *GraphEmbedding* went out of memory in the embedding phase due to the high memory complexity required by the Alias method they use for pre-computing the transition probabilities (Appendix C)[8]. *FastNode2Vec* and *PecanPy* went out of time (more than 48h of computation) on the biggest Wikipedia graph. In practice only *GRAPE* was able to successfully terminate the embedding and prediction tasks with all the three big real-world graphs considered here.

***GRAPE* improves the empirical time complexity of state-of-the-art libraries.** Fig. 4.4 a, b and c show the memory and time requirements of *GRAPE*, *FastNode2Vec* and *PecanPy* (note that the other state-of-the-art libraries ran out of time or memory on these real-world graph prediction tasks. With *CTD* and *PheKnowLator* biomedical graphs we can observe a speed-up of about one order of magnitude (Fig. 4.4 a, b) of *GRAPE* with respect to both *FastNode2Vec* and *PecanPy* with also a significant gain in memory usage with respect to *PecanPy* and a comparable memory footprint with *FastNode2Vec*. These results are confirmed by the average memory and time requirements across ten holdouts (Fig. 4.4 d). Note that both *FastNode2Vec* and *PecanPy* fail with the Wikipedia task, while *GRAPE* was able to terminate the computation in a few hours using a reasonable amount of memory (Fig. 4.4 c and d).

***GRAPE* boosts edge prediction performance.** *GRAPE* not only enables big graph embedding and speed-up computation, but can boost prediction performance on big real world graphs. Fig. 4.4-e and f show that *GRAPE* achieves better results on edge prediction tasks with both *CTD* and *PheKnowLator* biomedical graphs. *GRAPE* outperforms the other competing libraries at 0.001 significance level, according to the Wilcoxon rank sum test (Fig. 4.4 g). The edge embeddings have been used to train a decision tree to allow a safe comparison between the embedding libraries.

Appendix G reports AUROC, accuracy, and F1-score performances and other more detailed results about the experimental comparison of *GRAPE* with state of the art libraries.

[8]the Alias method has quadratic complexity with respect to the number of nodes in the graph, therefore becoming quickly too expensive on big graphs.

Chapter 5

ALPINE

In the previous chapters of this thesis, we introduced several methods for graph representation learning and explained how the *GRAPE* library implements these methods in a highly efficient and scalable manner. We compared *GRAPE* with other state-of-the-art libraries and showed that it greatly improves over their performance. However, these methods still face many challenges that hinder their scalability to real-world graphs, such as the internet, especially when using limited hardware.

In this chapter, we introduce the ALPINE framework, which allows for the execution of node embedding tasks on graphs with billions of nodes on commodity hardware in a rather limited amount of time.

5.1 Scalability limitations of SOTA methods

One common approach in the context of GRL is to train shallow graph neural networks (GNN) to predict the desired property given a set of nodes [57]. Notable examples of shallow GNNs include CBOW, SkipGram, GloVe [92] trained on either DeepWalk [94], Node2Vec [52] or Walklets [95] samples: these are models that learn whether nodes occur together within random walks. Other NN-based models are first, and second-order LINE [119], or TransE-like models [16], which learn whether two nodes are connected (LINE) or whether an edge connects two nodes with a given label (TransE).

A different approach embeds the graph elements using matrix factorization (MF) based techniques, including spectral and singular value decomposition methods. Notable examples include HOPE [89], which uses properties such as the cardinality of two nodes' neighbourhood overlap, or MatMF [99], which uses the node log-normalized co-occurrence of two nodes within a given window size.

In the last decade, literature has reported several examples of applications where embedding algorithms have shown promise in several fields. However, most of the embedding techniques presented so far suffer from the following limitations, which hamper their applicability to real-world (knowledge) graphs characterized by many nodes and edges.

First, Gradient descent-based models (e.g., GNN) are often trained on mini-batches of the property of interest, which are "lazily" computed to avoid intractable memory requirements.

The "lazy" evaluation (as opposed to "eager") computes on-demand the properties requested for each training batch, achieving memory requirements that generally scale linearly with the size of the training batch. The drawback of lazy computation is the need to recompute the property of interest multiple times during each training epoch.

Second, all the models mentioned above compute the embedding values of a specific node based on all other nodes and features. The computational dependency makes it hard to distribute the algorithms on High-Performance Computing (HPC) resources. It generally involves iterative graph partitioning and synchronization steps to merge the computed weights through the network.

Analogous considerations also apply for distributing the computation across GPUs. Indeed, most methods require random access to the node embedding values; however, the memory hierarchy of modern CPUs is optimal for local and sequential accesses, while random accesses, especially when the matrix is MMAPed from disk, are orders of magnitude slower. Therefore, random access to the nodes while computing the embeddings makes it unfeasible to distribute the memory requirements between the main memory and disk through Memory Mapping (MMAP) [71].

Third, often these methods employ real-valued data types, which are generally represented as either 32-bits (full precision) or 16-bits (half-precision) floating point values. Unfortunately, all commercially available CPUs support half-precision only through emulation. On the other hand, GPUs widely support half-precision operations [75], but their dedicated memory (VRAM) is generally significantly smaller than the standard RAM, which constrains the size of the feasible tasks. Mini floats (i.e. 8 bits floating point values) are gaining interest in literature [128]. Until recently, Mini floats were supported only by application-specific integrated circuits [45], and, since October of 2022, Nvidia has introduced support for two types of Mini floats in the new "Ada Lovelace" architecture [88].

Fourth, embedding methods often start from random values and maintain significant noise from initialization, even at convergence. The noise considerably limits the compression ratio of libraries such as gzip [46], thus, making it impractical to share the embeddings of large graphs over the internet, constraining the FAIRness of these results.

Fifth, the properties used to compute the embeddings often implicitly depend on the node degrees. In particular, high-degree nodes often have a disproportionate impact on the embeddings compared to low-degree nodes. In particular, high-degree nodes are contextual to more nodes, are the endpoints in more edges, and often have shared neighbours with any sampled vertex. Topological sampling methods extract high-degree nodes more often, leading to embeddings biased towards this type of node. The design of node embedding algorithms should consider this essential property of the obtained embedding.

Lastly, the embedding methods presented so far lack interpretability and explainability of the computed embeddings. Indeed, while it is generally true that a correlation exists between distances defined between the computed embeddings, the stand-alone meaning of any given element of the embedded vector is usually unknown, resulting in a significant lack of interpretability.

Interpretability and explainability are issues of particular importance that should be addressed to allow computing human-understandable embeddings. This would produce more informative GRL results, particularly appealing in medical and clinical informatics contexts.

Finally, as witnessed by the increasing usage of the currently ever-mentioned "human-in-the-loop" term, it is crucial to develop applications where human knowledge can be easily integrated. For most embedding algorithms presented so far, straightforward knowledge integration is complex.

5.2 Overcoming the limitations with ALPINE

In this thesis, we present Abstract Landmark Properties-Inferred Node Embedding, henceforth **ALPINE**, an algorithmic framework for interpretable node embedding based on computationally independent integer properties (Sections 5.4 and 5.5) defined on abstract landmarks, i.e. sets of nodes sharing a common task-specific characteristic (Section 5.6). ALPINE-based algorithms scale on graphs with billions of nodes and edges on commodity hardware and HPC.

After detailing the notation used in the remaining part of this chapter, in section 5.4, we describe the impact of computational dependency and why computational independence is fundamental when scalability is of paramount importance. Then, we discuss the limited support for operations on small floating point values in commercially available hardware and arguments for preferring small integer values (Section 5.5).

Next, we introduce the **novel** concept of *abstract landmarks*, which are groups of nodes in a graph that share a distinctive characteristic. These groups of nodes can be thought of as a single, abstract node that represents one of the graph's key ideas or themes. We use this concept to ensure feature interpretability by design in our ALPINE algorithm. The specific landmarks used in the algorithm can be chosen based on the characteristics of the graph, or they can be designed for a specific task by an expert. The landmarks do not have to be a partition of the graph, and a single node can be included in multiple landmarks. In addition to defining the landmarks, we also introduce the concept of landmark-based properties, which are functions that can be used to compute feature columns of a node embedding independently from other feature columns. These properties can be based on existing graph properties or they can be original and specific to the task and graph at hand (Section 5.6). Landmarks function as a tool to allow feature engineering and the integration of experts' knowledge into embedding algorithms.

Next, we describe two possible landmark computation schema and their potential applications (Sections 5.7.2 and 5.7.2). In section 5.7, we define the elements composing the ALPINE algorithmic framework, alongside considerations on the scalability of distributed systems. Finally, we describe two concrete ALPINE algorithms: SPINE (shortest paths inferred node embedding) and WINE (windows inferred node embedding). SPINE and WINE implementations are part of the *GRAPE* library, presented in chapter 2. We will use undirected WikiData (1.2G nodes and 12.4G edges) as a real-world example driving the need for scalable algorithms. Alongside each algorithm, we provide the theorems detailing their time and memory complexities.

5.3 Notation

A graph $G = (V, E)$ is composed of a set of nodes V and edges $E \subseteq V \times V$, respectively representing entities and relationships.We denote the neighbours of a given node v_i as

$\mathcal{N}(v_i) = \{v_j \mid (v_i, v_j) \in E, \quad v_i, v_j \in V\}$. The outbound degree of a node $d(v)$ is the cardinality of the set of neighbours $d(v) = |\mathcal{N}(v)|$.

For a given graph, node embedding methods compute an embedding matrix Φ with shape $(|V|, f)$, which contains an f-dimensional vector representation for each node $v \in V$. We will denote Φ_v as the f-dimensional node embedding vector associated with the node v. Moreover, we denote $\boldsymbol{\varphi}_i$ as the i-th column of the matrix Φ, which is the $|V|$-dimensional feature vector in the embedding space.

In the remaining part of this work, we adopt the mathematical notation of Cormen et al. [31]. All algorithms involve parallel computation. Therefore, their formal computational and space complexity analysis assume a theoretical concurrent-read concurrent-write shared memory parallel random access machine (CRCW-PRAM) [44] with t threads and word sizes $\geq \lceil \log_2 |E| \rceil$ bits. We use $m_V \geq \lceil \log_2 |V| \rceil$ bits as the memory needed to store a single node index, usually 32-bits for graphs with less than $|V| < 2^{32} \approx 4.3G$ nodes. The dimension of the embedding space (i.e. the number of elements/features of the embedding vectors) will be denoted as f, while m_f will refer to the number of bits needed to store one feature value.

5.4 Computational independence

The computational dependency between features obliges most of the state-of-the embeddings methods to either load the entire embedding matrix into memory or distribute its memory requirements between main memory and disk, which leads to intractable memory or time requirements when processing large graphs. This problem mainly affects the scalability of many SOTA methods involving distances (MSE) or similarities (dot product) between node embeddings. For instance, WikiData's node embedding would require $1.2 \cdot f \cdot 4$GB using a single precision float (4 bytes), which, for a number of features such as $f = 100$, tallies to ≈ 480GB.

Hence, the computational independence of embedding features is a fundamental requirement for designing efficient and scalable GRL algorithms. If individual embedding features were computed independently of each other, the memory requirements could be reduced by a factor of f, which translates to a 100-fold improvement in WikiData's example, hence reaching a computational memory requirement of ≈ 4.8GB. Computational independence makes it trivial to distribute the computation across machines, allowing perfect horizontal scalability on HPCs.

5.5 Representing features with small integers

While literature often employs iterative optimization strategies on float values, integer values have better hardware support, especially for single or double bytes values. All CPUs natively implement integer operations. In contrast, half-precision floating point operations require emulation with multiple integer operations leading to lower performance. Therefore, small-integer valued features are preferable. For instance, using single-byte features reduces the memory requirements by 4. In many real-world graphs, the small world hypothesis holds true [7], so that shortest-path distances may be expressed via small integers that follow a scale-free distribution. In WikiData, the longest of the shortest paths, i.e. the diameter, is only 7. By choosing small integer features over floats, we enjoy a small memory footprint, hardware acceleration, and often a good compression ratio because of their scale-free distribution. Achieving a small compressed file size is fundamental for the FAIRness of the resulting embedding.

5.6 Interpretable abstract landmarks properties

In this section, we introduce the novel concept of abstract landmarks and their properties, and how we use this concept to ensure feature interpretability **by design** in ALPINE. A landmark is a group of nodes in a graph that share a distinctive trait or characteristic, such as similar node degree or common labels. This group of nodes can be thought of as a single, abstract node that represents one of the graph's key ideas or central themes.

We stress that the interpretability present in ALPINE embedding is not inherent in ALPINE by itself nor an emergent property, but derives from the selection of meaningful landmarks and features with a clear interpretable meaning, i.e. to ensure an interpretable ALPINE embedding there must be maintained a clear and unbroken interpretability chain. This section will focus on the description of the landmarks, and we will describe interpretable features defined upon landmarks in sections 5.7.3 and 5.7.4.

The ALPINE algorithm uses a set of landmarks to compute node embeddings. The specific landmarks used by the algorithm can be chosen based on the characteristics of the graph, or designed for a specific task by an expert. For instance, a landmark might be defined based on node degrees or labels. It is important to note that the set of landmarks does not have to be a partition of the graph, and a single node can be included in multiple landmarks.

In addition to defining the landmarks themselves, we also introduce the concept of landmark-based properties. These are functions that, given a landmark, can compute a *feature column* of a node embedding independently from all other feature columns. These functions can be based on existing graph properties, or they can be original and specific to the considered task and graph. In the following sections, we provide concrete examples of how abstract landmarks and their properties can be defined and used in practice.

5.6.1 Abstract landmarks

A landmark $L \in \mathcal{P}(V)$, where $\mathcal{P}(V)$ is the power-set of the nodes, is a set of nodes characterized by some shared distinctive trait, such as similar node degree or common labels. Conceptually, a landmark is an abstract source node equivalent to merging all the constituent nodes into a single one. It represents the graph's semantical origin or zenith of a designated notion. We define the outbound neighbours of a landmark L as $\mathcal{N}(L) = \bigcup_{s \in L} \mathcal{N}(s)$. A landmark does not have inbound edges. Most commonly, the cardinality of a landmark is $|L| \ll |V|$.

An ALPINE algorithm relies on a set of landmarks $\mathcal{L} \subset \mathcal{P}(V)$ to compute a node embedding with $f = |\mathcal{L}|$ features. The set of landmarks $\mathcal{L}$ is not necessarily a graph partition, i.e. $V \supseteq \bigcup_{L \in \mathcal{L}} L$, nor its constituent sets have to be disjointed, i.e. a node $v \in V$ may appear in multiple landmarks. A feature engineering schema is employed to compute the set of landmarks $\mathcal{L}$ and can be defined based on the graph's geometric characteristics or task-specific and designed by a field expert. We provide two concrete examples of such schema, based on node degrees (section 5.7.2) and node labels (section 5.7.2).

5.6.2 Abstract landmark properties

A key component of an ALPINE algorithm is the associated landmark-based property $\boldsymbol{\lambda}(L) : \mathcal{L} \to \mathbb{N}^{|V|}$, i.e. a function that, given a landmark $L \in \mathcal{L}$, computes the associated feature column $\boldsymbol{\varphi}_L \in \mathbb{N}^{|V|}$ of a node embedding $\Phi \in \mathbb{N}^{|V| \times |\mathcal{L}|}$, independently from all other feature columns. Note that $\boldsymbol{\lambda}$ is defined on a specific graph $G = (V, E)$ and on a specific landmark set $\mathcal{L}$.

While the definition of $\boldsymbol{\lambda}(L)$ may also be completely original and specific for the considered task and graph, a straightforward approach to obtain a landmark-based property $\boldsymbol{\lambda}(L) : \mathcal{L} \to \mathbb{N}^{|V|}$ is to generalize one of the many graph properties $\boldsymbol{\rho}(s) : V \to \mathbb{N}^{|V|}$ available in the literature, defined between a source node $s \in V$ and all nodes V in the graph. Many approaches are possible, including direct generalization such as the minimum ($\boldsymbol{\varphi}_L(v) = \boldsymbol{e}_v^T \boldsymbol{\lambda}(L) = \min_{s \in L} \boldsymbol{e}_v^T \boldsymbol{\rho}(s)$), maximum ($\boldsymbol{\varphi}_L(v) = \boldsymbol{e}_v^T \boldsymbol{\lambda}(L) = \max_{s \in L} \boldsymbol{e}_v^T \boldsymbol{\rho}(s)$), sum ($\boldsymbol{\varphi}_L(v) = \boldsymbol{e}_v^T \boldsymbol{\lambda}(L) = \sum_{s \in L} \boldsymbol{e}_v^T \boldsymbol{\rho}(s)$) or mean ($\boldsymbol{\varphi}_L(v) = \boldsymbol{e}_v^T \boldsymbol{\lambda}(L) = \left\lceil {}^{1}/_{|L|} \sum_{s \in L} \boldsymbol{e}_v^T \boldsymbol{\rho}(s) \right\rceil$) of the node-based property. Any statistic of $\boldsymbol{\rho}(s)$ may be employed, given an efficient method to directly compute it.

We list the following desirable soft requirements to guide the selection of a scalable landmark-based property $\boldsymbol{\lambda}(L)$:

1. For a property $\boldsymbol{\lambda}(L)$ to be considered for the ALPINE framework, its asymptotic worst-case time $T_{\boldsymbol{\lambda}}$ and space $S_{\boldsymbol{\lambda}}$ complexities should be at most linear with regards to the number of nodes and edges, i.e. $T_{\boldsymbol{\lambda}}, S_{\boldsymbol{\lambda}} \in O(|V| + |E|)$.

2. On non-pathological cases, the time complexity of $\boldsymbol{\lambda}(L)$ should be inversely proportional to the number of available threads t, i.e. $T_{\boldsymbol{\lambda}} \in \Theta({}^{1}/_{t})$, and its space complexity should be sublinear w.r.t t, i.e. $S_{\boldsymbol{\lambda}} \in o(t)$. In short, the computation of $\boldsymbol{\lambda}(L)$ should employ all available cores, and its associated data structures should not be replicated for all cores.

3. Finally, the asymptotic worst-case time $T_{\boldsymbol{\lambda}}$ and space $S_{\boldsymbol{\lambda}}$ complexities should have *sublinear scaling* on the landmarks' cardinality $|L|$, i.e. the the property computation should, ideally, not rely on computing a node-based property for all the nodes $s \in L$:
$$T_{\boldsymbol{\lambda}} \in o(|L|) \qquad S_{\boldsymbol{\lambda}} \in o(|L|)$$

We present two algorithms to compute properties $\boldsymbol{\lambda}$ fulfilling these requirements, namely SPINE (section 5.7.3) and WINE (section 5.7.4).

5.7 The ALPINE algorithmic framework

Given a graph $G = (V, E)$, a concrete ALPINE implementation requires the definition of two procedures:

1. Firstly, a schema ℓ_G, defined for the provided graph G, to create a set of landmarks $\mathcal{L} \subset \mathcal{P}(V)$. A schema ℓ_G can be specific for the considered task and graph G, but generic approaches are also possible. The cardinality of $\mathcal{L}$ determines the dimensionality of the embedding Φ, i.e. $f = |\mathcal{L}|$.

2. Secondly, a property $\boldsymbol{\lambda}(L) : \mathcal{L} \to \mathbb{N}^{|V|}$ which, given a landmark $L \in \mathcal{L}$ and a graph G, computes the associated feature column $\boldsymbol{\varphi}_L \in \mathbb{N}^{|V|}$. The property $\boldsymbol{\lambda}$ must be defined for the provided graph G and the set of landmarks $\mathcal{L}$ obtained using ℓ_G. Most commonly, the property $\boldsymbol{\lambda}$ represents either some notion of similarity or distance between each node $v \in V$ and the provided landmark $L \in \mathcal{L}$.

Most of the interpretability of ALPINE relies on the selected landmark generation schema ℓ_G. Indeed, the embedded features are constructed on the basis of a specific concept/landmark $L \in \mathcal{L}$ that expresses either a well-defined topological or semantic property of the graph. This, in turn, allows us to embed apriori knowledge on the algorithmic framework, using, e.g. set of landmarks $\mathcal{L}$ belonging to the apriori known category: for instance, if nodes represent genes, we can construct landmarks by grouping genes/nodes associated with a given disease, or with a specific biological function.

We employ the computational independence of $\boldsymbol{\lambda}$ to compute independently the feature columns $\boldsymbol{\lambda}(L) = \boldsymbol{\varphi}_L \; \forall L \in \mathcal{L}$ of the node embedding $\Phi \in \mathbb{N}^{|V| \times |\mathcal{L}|}$.

Depending on the dimension of the graph G and on the available computational resources, the embedded features $\boldsymbol{\varphi}_L$ can be computed sequentially (e.g. on a desktop or laptop computer) or in parallel using both multi-core architectures or a distributed or an HPC environment (Algorithm 2). In the first line of algorithm 2, we iterate over the landmarks $L \in \mathcal{L}$. Given the complete independence of the feature computation, we can distribute each iteration both horizontally, using upwards to $c \leq f$ computing nodes, and, when $f > c$, such as on a single desktop $c = 1$, we can distribute the computation of the features temporally, i.e. we compute the features sequentially. While computing multiple features $f' \leq f$ concurrently on the same node is possible, the memory requirements would increase by f' times, making this often unfeasible. However, the design of algorithms computing $\boldsymbol{\lambda}(L)$ should heavily exploit parallelism. Hence the computation of a single feature should use all available cores. For instance, the algorithms we present to compute $\boldsymbol{\lambda}(L)$ can employ all available cores except on pathological graphs such as chains. The number of computing nodes c available in modern HPC is often in the thousands. Generally, $c \gg f$, for instance, the Fugaku system has $c \approx 159k$ compute nodes [36]. These vast computational resources may be employed to explore multiple landmark generation schemas simultaneously. While we executed all the experiments reported on a commodity desktop computer, where embedding WikiData requires about one hour, our implementation supports SLURM-based distribution [130] to scale on significantly larger graphs.

In the second line algorithm 2, given the assigned landmark L, each computing node executes the provided function $\boldsymbol{\lambda}(L)$ to compute the relative feature $\boldsymbol{\varphi}_L$, ideally employing all available cores.

In the third line, we explicit the non-trivial practical step $\sigma(\boldsymbol{\varphi}_L)$ of storing the obtained feature $\boldsymbol{\varphi}_L$, primarily dependent on the available hardware. Writing to permanent storage is an IO task, and the design of σ should consider the available hardware to achieve good performance. Lacklustre implementations of σ might cause the time required to store the feature to overtake the time necessary to compute it, making the task IO-bound. Network disks, commonly used in HPCs, may be particularly detrimental to performance when multiple nodes try to write to the same file. In such cases, each feature should be written on a different file and later merged if needed. When the overhead of working on a single file is negligible, such as when $c = 1$, σ should store the feature in Fortran order, i.e. column-wise, which guarantees the best cache-locality and operations such as left dot products. Procedures such as right-dot products require the transposition of the embedding to be in C order, i.e. row-wise. In the implementation, we provide routines to transpose the embedding on disk, with constant RAM requirement.

Algorithm 2 ALPINE Framework

Input the graph $G = (V, E)$, a set of landmarks $\mathcal{L} \subset \mathcal{P}(V)$ and two functions: $\boldsymbol{\lambda}(L) : \mathcal{L} \to \mathbb{N}^{|V|}$ to compute a single computationally independent feature $\boldsymbol{\varphi}_L \in \mathbb{N}^{|V|}$ from a landmark $L \in \mathcal{L}$, i.e. an embedding column with $|V|$ elements, from landmarks, and $\sigma(\boldsymbol{\varphi}_L)$ to store the embedding column $\boldsymbol{\varphi}_L$.

```
1: for $L \in \mathcal{L}$ distributedly do
2:     $\boldsymbol{\varphi}_L \leftarrow \boldsymbol{\lambda}(L)$
3:     $\sigma(\boldsymbol{\varphi}_L)$
```

The proofs relative to the theorems regarding the complexities of the proposed algorithms are provided in section 5.9.

5.7.1 Graph data structure

For our time complexity analysis we assume the graph data structure to be a compressed-sparse row (CSR) matrix [19], which is composed of the cumulative outbound node degree vector $C \in \mathbb{N}^{|V|}$ and the destination nodes vector $D \in \mathbb{N}^{|E|}$. WikiData requires $4B$ integers for each destination node and $8B$ integers for each element of the comulative outbound degree vector, for a total of $12.4G \cdot 4B + 1.2G \cdot 8B = 59.2GB$. In a CSR setting, computing the outbound node degree $d(v)$ and iterating a node's destinations $\mathcal{N}(v)$ require constant and $d(v)$ time, respectively. Both operations have constant memory.

5.7.2 Computing abstract landmarks

In this section, we present schemas for the computation of a set of landmarks $\mathcal{L}$ based on node degrees and node labels, detailed in sections 5.7.2 and 5.7.2, respectively. We define $T_{\text{sort}}(k), S_{\text{sort}}(k)$ as the time and memory worst-case to sort a vector of length k using t threads. For the sorting operations we use Rust Rayon [21] Parallel Quick-sort which ensures $T_{\text{sort}}(k) = O(k \log_2 k)$ and $S_{\text{sort}}(k) = O(t)$.

Degree-based landmarks

This section describes a schema for calculating landmarks based on outbound node degrees. High-degree nodes represent in many networks the most important elements. They represent the graph centers, while low-degree nodes represent the periphery. Often, nodes with a similar degree share a similar immediate topological structure. Thus landmarks built on nodes with similar node degrees represent a specific topological structure across the graph (even across otherwise disconnected components). In real-world graphs, low-degree nodes are more common than high-degree ones, and node degrees often follow a scale-free distribution. This phenomenon implies that the higher the node degree is, the rarer nodes with a similar degree will be, and vice-versa for low-degree nodes. Using this insight to balance the cardinality of the different landmarks, we will compute them to assign roughly the same number of edges to all landmarks, i.e. the sum of the degree of the nodes $s \in L$ should be roughly the equal across all landmarks ${}^{|E|}/_{f} \approx \sum_{s \in L} d(s) \quad \forall L \in \mathcal{L}$, where $d(v)$ is the degree of a given node $v \in V$.

In algorithm 3 we start by creating a vector with the graph nodes. We sort the nodes by *decreasing* node degree (line 2). Subsequently, we iterate on the sorted nodes and populate the current landmark L. Each time we add a node to L, we increase the number of edges contained therein. Every time the number of edges exceeds ${}^{|E|}/_{f}$ edges, we push a new landmark L to $\mathcal{L}$ and reset both L and the number of edges (lines 6-12). In practice, we implement row 8 through lazy generators (e.g. the Python-like yield operator [108]) to avoid unnecessary memory allocation. Note that we will analyze the lazy and more efficient version in all the theorems.

We observe that it is possible to use outbound node degrees, inbound node degrees or a combination of the two in the context of directed graphs. While node degrees are an efficient and interpretable metric, this landmark computation schema works with any scoring mechanism.

A possible application of this landmark selection schema could be the prediction of real-estate value using a graph with nodes representing buildings and edges representing roads. By using algorithm 3, we obtain landmarks sorted from less central nodes, which are the ones closer to the periphery, to very central nodes, which are closer to city centers. Real-estates closer to the city centers and, therefore, having access to more services may obtain a higher valuation than those in the periphery.

Theorem 5.7.1. *The worst-case time and memory upper bounds for algorithm 3 are* $\Theta(|V|) + T_{sort}(|V|) + O(t)$ *and* $2m_V \cdot |V| + S_{sort}(|V|) + O(t)$ *bits* $= \Theta(|V| \log|V|) + O(t)$, *respectively.*

Algorithm 3 Degree-based Landmarks

Input the number of features $f \in \mathbb{N}$, a graph $G = (V, E)$, a routine *degree_sort* to sort in-place nodes by decreasing degrees

Output a landmark set $\mathcal{L} = \{L_1, L_2, \dots L_f\}$

```
 1: nodes ← $V$                                   ▷ $\Theta(|V|)$
 2: degree_sort(nodes)                            ▷ $T_{\text{sort}}(|V|)$
 3: edges_count ← 0
 4: $L$ ← empty vector
 5: $\mathcal{L} \leftarrow \emptyset$
 6: for $v \in$ nodes do                          ▷ $\Theta(|V|)$
 7:     if edges_count > ${}^{|E|}/_{f}$ then
 8:         $\mathcal{L}$.push($L$)
 9:         edges_count ← 0
10:         $L$ ← empty vector
11:     $L$.push($V$)                             ▷ $\Theta(1)$
12:     edges_count+= $d(v)$                      ▷ $\Theta(1)$
13: return $\mathcal{L}$
```

Categorical landmarks

In algorithm 4 we associate to each landmark all the nodes belonging to a given category (nodes may belong to multiple categories).

Suppose the categories are whether a given building are hospitals, houses, schools, railway stations, or incinerators. Proximities (or distances) to the landmarks associated with the services may be valuable in evaluating real estate.

It is possible to combine the categorical landmarks with the node degree landmarks to obtain potentially more expressive features.

Theorem 5.7.2. *The worst-case time and memory upper bounds of algorithm 4 are* $\Theta(|C|\cdot|V|/t)$ *and* $m_V \cdot \max_{k\in C}|\{v \mid c(v) = k\}|\mathit{bits} = \Theta(|V|\log|V|)$, *respectively.*

Algorithm 4 Categorical landmarks

```
    Input set of labels $C$, a graph $G = (V, E)$, a function $c(v)$ to get the category of a node $V$
    Output a landmark set $\mathcal{L} = \{L_1, L_2, \dots L_{|C|}\}$
1: $\mathcal{L} \leftarrow \emptyset$
2: for $k \in C$ do                                         ▷ $\Theta(|C||V|/t)$
3:     $L \leftarrow$ empty vector
4:     for $v \in V$ do in parallel                           ▷ $\Theta(|V|/t)$
5:         if $c(v) = k$ then
6:             $L$.push($V$)                                  ▷ $\Theta(1)$
7:     $\mathcal{L}$.push($L$)
8: return $\mathcal{L}$
```

5.7.3 SPINE

In this section, we present SPINE, a parallel algorithm $\boldsymbol{\lambda}(L) : \mathcal{L} \to \mathbb{N}^{|V|}$ to compute an embedding column $\boldsymbol{\varphi}_L$ based on shortest path distances from a landmark $L \in \mathcal{L}$. Starting from the definition of the shortest path distances $\boldsymbol{\rho}(s)$ from a source node $s \in V$ to all the V nodes in the graph, we generalize $\boldsymbol{\rho}(s)$ to $\boldsymbol{\lambda}(L)$ as the minimum shortest path distances between any node in the landmark L and all the V nodes in the graph: $\boldsymbol{\varphi}_L = \boldsymbol{e}_v^T \boldsymbol{\lambda}(L) = \min_{s\in L} \boldsymbol{e}_v^T \boldsymbol{\rho}(s)$.

Providing the distance from the closest $s \in L$ makes intuitive sense: consider an application case where L represents hospitals, and the other nodes represent rock climbing sites. When determining the latters' insurance risk, the distance from the closest hospital to each rock climbing is more relevant than knowing, for instance, the average distance to all hospitals in a country.

The property $\boldsymbol{\lambda}(L)$ respects the requirement of being a small integer value, as in many real-world graphs, the longest shortest path in a graph, the diameter, is indeed a tiny integer value [7].

Since we intend to compute the shortest path distance from any $s \in L$ to all nodes V, without being interested in which node is the source of the shortest path, in the algorithm 5 we use an approach analogous to a parallel breadth-first search [18].

Algorithm 5 uses *parallel frontiers* within the context of an iterative graph exploration to store the nodes to visit at the next step. A parallel frontier ξ is an unsorted vector supporting concurrent iteration and wait-free constant-time push. Given t threads, our concrete implementation of the frontier contains t pointers to node vectors. Each i-th thread owns the respective vector ξ_i and can push nodes to ξ_i without synchronization steps. We will represent this as ξ.push(V) with an abuse of notation. While duplicated values are allowed in a frontier, our algorithms guarantee the absence of duplicates through atomic operations [109]. Thus, we can define the set of nodes in a frontier ξ as the union of the nodes contained in its constituent vectors, i.e. $\xi = \bigcup_{i\in[0,t)} \xi_i$. It follows that the frontier cardinality as $|\xi| = \sum_{i\in[0,t)} |\xi_i|$. A frontier is empty when $|\xi| = 0$. ξ is an immutable object during the iteration procedure.

We initialize all values of the distance vector $\boldsymbol{\varphi}_L$ of length $|V|$ with the maximum feature value MAX, which is 256 for an unsigned single-byte integer (line 1). We proceed to set the distances from L of all $s \in L$ to zero, marking them as visited (lines 2 and 3). We initialize the depth $d = 1$(line 4) and assign the landmark L to the nodes frontier $\xi = \{v \mid \boldsymbol{e}_v^T \boldsymbol{\lambda}(L) = d-1 \quad v \in V\}$. We iterate until we do not encounter any new node, i.e. ξ is empty (line 6), or the maximum depth MAX is reached (line 7). In the latter case, the computed distances $\boldsymbol{\varphi}_L$ (line 8) are ready and can be returned. In line 9 we create the temporary node frontier $\xi_{\text{TMP}} = \{v \mid \boldsymbol{e}_v^T \boldsymbol{\lambda}(L) = d \quad v \in V\}$. We start the internal for loop (lines 10-14) by iterating over the $v \in \xi$ nodes and their neighbours $w \in \mathcal{N}(v)$ (lines 10 and 11). We check whether each neighbour w has a distance equal to MAX and if so, we set its distance to d and add it to the temporary frontier ξ_{TMP}. Concretely, to avoid pushing multiple times w to ξ_{TMP}, both the if-condition and the distance assignment are implemented by using a single-word compare-and-swap (CAS) instruction [109]. CAS is a standard CPU instruction to transactionally exchange the value of a register and a word of memory if, and only if, the word of memory is equal to an expected value. CAS allows efficient, sequentially consistent read-modify-write operations without expensive synchronization mechanisms like locks and semaphores. Once the loop is complete, we proceed to increase the current depth d (line 15) and to reassign the temporary frontier ξ_{TMP} to ξ (line 16). We observe that the number of nodes in either frontier is always $|\xi| + |\xi_{\text{TMP}}| \leq |V|$.

Once the while-loop completes, we clip all distances that have not yet been visited to d to avoid characterizing unreachable nodes such as nodes in distinct components with respect to all $s \in L$ or singletons with a potentially excessively high value (lines 17, 18 and 19). The algorithm concludes by returning the computed distances $\boldsymbol{\varphi}_L$ (line 20).

We observe that SPINE may be generalized to graphs characterized by small non-negative integer edge weights [78].

A first approximation stems from observing that average distances are much less than the maximum depth, the diameter d. For instance, in the Facebook relationships graph, the average distance is approximately four, while the diameter of the largest component is $d = 41$ [7]. An approximation is to cap the maximum depth to some $\delta < d$ to get the correct distance for most nodes and a clipped value for the pathological cases. This leads both to smaller time and memory requirements, as it requires less iterations of the main loop and the features can be stored in $\lceil \log_2 \delta \rceil$ bits instead of $m_f = \lceil \log_2 d \rceil$ bits. A second approximation adopts the approximated random-walk approach described in (section 2.3.2), by employing the Sorted Unique Sub-Sampling (SUSS) algorithm sub-sample the node neighbours of very high-degree nodes. This approximation can be interpreted as regularization and may reduce the topology overfitting. Neither of these approximations was necessary to embed WikiData on a desktop computer. Still, they may find application in tasks relative to larger graphs.

Theorem 5.7.3. *The worst-case time and memory upper bounds of algorithm 5 are* $T_{\boldsymbol{\lambda}} = O(|V| + |E|)$ *and* $S_{\boldsymbol{\lambda}} = (m_V + m_f)|V|$ *bits, respectively.*

Note that, in practice $|\xi| + |\xi_{\text{TMP}}| \ll |V|$. $|\xi| + |\xi_{\text{TMP}}| \approx |V|$ exclusively in degenerate topologies such as stars.

Algorithm 5 SPINE

```
    Input landmark $L$, $MAX$ feature value, method to iterate neighbours $\mathcal{N}(v)$
    Output shortest path distance vector $\boldsymbol{\varphi}_L$ from $L$
 1: $\boldsymbol{\varphi}_L \leftarrow$ vector filled with $|V|$ $MAX$ values        ▷ $\Theta(|V|)$
 2: for $s \in L$ do in parallel                                    ▷ $\Theta(|L|/t)$
 3:     $\boldsymbol{\varphi}_L[s] \leftarrow 0$
 4: $d \leftarrow 1$
 5: $\xi \leftarrow L$
 6: while $|\xi| > 0$ do                                            ▷ $O(|V| + |E|)$
 7:     if $d = MAX$ then
 8:         return $\boldsymbol{\varphi}_L$
 9:     $\xi_{\text{TMP}} \leftarrow$ empty frontier
10:     for $v \in \xi$ do in parallel
11:         for $w \in \mathcal{N}(v)$ do
12:             if $\boldsymbol{\varphi}_L$[w] $=$ $MAX$ then
13:                 $\boldsymbol{\varphi}_L[w] \leftarrow d$
14:                 $\xi_{\text{TMP}}$.push($w$)
15:     $d$ += 1
16:     $\xi \leftarrow \xi_{\text{TMP}}$
17: for $v \in V$ do in parallel                                    ▷ $\Theta(|V|/t)$
18:     if $\boldsymbol{\varphi}_L[v] = MAX$ then
19:         $\boldsymbol{\varphi}_L[v] \leftarrow d$
20: return $\boldsymbol{\varphi}_L$
```

5.7.4 WINE

In this section, we present WINE, a parallel algorithm $\boldsymbol{\lambda}(L) : \boldsymbol{\mathcal{L}} \to \mathbb{N}^{|V|}$ to compute the co-occurrence count vector $\boldsymbol{\varphi}_L \in \mathbb{N}^{|V|}$ between each node $v \in V$ and a given landmark $L \in \boldsymbol{\mathcal{L}}$ within a window of size ω, i.e. an embedding column with $|V|$ elements. More precisely, we define the co-occurrence count $\boldsymbol{\rho}_\omega(s)$ between a source node s and all the nodes V within a window size ω as the number of paths of length $\leq \omega$ from the source node s to each node $v \in V$. We then generalize $\boldsymbol{\rho}_\omega$ to $\boldsymbol{\lambda}_\omega(L)$ as the sum of the number of paths of length $\leq \omega$ from each source node $s \in L$ and each node V in the graph destination: $\boldsymbol{\lambda}_\omega(L) = \sum_{s \in L} \boldsymbol{\rho}_\omega(s)$.

When $\omega = 2$, as in two-hops WINE (Section 5.7.4), each value is equal to a non-normalized Jaccard index [60], i.e. $\boldsymbol{\varphi}_L(v) = \boldsymbol{e}_v^T \boldsymbol{\rho}_2(s) = |\mathcal{N}(v) \cap \mathcal{N}(s)|$, where $\boldsymbol{e}_v$ is a vector of length $|V|$ with a one in the position corresponding to the node v and zeros otherwise.

The same consideration applies also to landmarks generalization, where: $\boldsymbol{\varphi}_L(v) = \boldsymbol{e}_v^T \boldsymbol{\lambda}_2(L) = |\mathcal{N}(L) \cap \mathcal{N}(v)|$, with $\mathcal{N}(L) = \bigcup_{s \in L} \mathcal{N}(s)$.

Analogies exist between WINE and power method-based centralities such as personalized PageRank [9]. To make these analogies plain, we redefine $\boldsymbol{\rho}_\omega(s) = \boldsymbol{e}_s^T \left(\sum_i^\omega A^i\right)$, where A is the adjacency matrix of the graph.

By using the definition, it follows that $\boldsymbol{\lambda}_\omega(L) = \left(\sum_{s \in L} \boldsymbol{e}_s^T\right) \left(\sum_i^\omega A^i\right)$. In recommender systems, the elements of interest for a given user are represented as a binary vector $\boldsymbol{u}$[69], for our purposes $\boldsymbol{u} \in \{0, 1\}^{|V|}$. In the context of WINE, the field expert is the user selecting the elements of interest, i.e. the nodes composing the landmark. It follows that $\boldsymbol{u}_L = \sum_{s \in L} \boldsymbol{e}_s$ and therefore we can express this as $\boldsymbol{\lambda}_\omega(L) = \boldsymbol{u}_L \sum_{i=0}^{\omega} A^i$.

Providing the co-occurrence count between a node and a landmark makes intuitive sense: consider, for instance, an application case of a graph whose nodes are ecosystems, and the edges represent routes that species may use to move between different ecosystems. Using the categorical landmarks schema (section 5.7.2), we can design landmarks containing ecosystems where an invasive species, e.g. different mosquito species, has taken root. WINE features may be employed to predict the risk of range shifting [121], i.e. the probability of a deleterious invasive species spreading across ecosystems [14], and inform mitigation measures.

The two-hops algorithm 7 differs significantly from the general WINE algorithm 6 as $\omega = 2$ has many exploitable implications. We will describe the two-hops algorithm in section 5.7.4.

ω-Hops WINE

We now proceed to describe Algorithm 6. We initialize the co-occurence counts vector $\boldsymbol{\varphi}_L$ and the co-occurrences variations vector $\boldsymbol{\delta}$ with the sum of the standard basis vectors $\boldsymbol{u}_L$ representing the landmark L, i.e. $\boldsymbol{u}_L = \sum_{s \in L} \boldsymbol{e}_s$ (line 1). Note that at any given iteration $\gamma \in [0, \omega)$ it holds that $\boldsymbol{\varphi}_L = \boldsymbol{u}_L \sum_{i=0}^{\gamma} A^i$ and $\boldsymbol{\delta} = \boldsymbol{u}_L A^\gamma$. We assign the landmark L to the nodes frontier $\xi = \{v \mid \boldsymbol{\delta}[v] > 0 \quad v \in V\}$, i.e. the destinations of all the paths starting from $s \in L$ of length γ at each iteration γ (line 2). In line 3 we iterate up to ω where we initialize the temporary co-occurrences variation vector $\boldsymbol{\delta}_{\text{TMP}}$, which will contains the occurrences from this iteration, with $|V|$ zeros and create a new empty frontier ξ_{TMP}. In the two inner loops we iterate on the nodes $v \in \xi$ and their neighbours $w \in \mathcal{N}(v)$. If the procedure has not yet visited w during the γ-th iteration, we push it to the new frontier ξ_{TMP}, thus assuring that ξ_{TMP} collects only newly visited nodes. Since we use m_f bits to represent an unsigned integer feature value, the maximum representable value is the finite number $2^{m_f} - 1$. The $\boldsymbol{\delta}_{\text{TMP}}$ and $\boldsymbol{\varphi}_L$ additions (lines 10 and 11) may cause numerical overflow when using the wrapping addition from modular arithmetic $a +_W b = (a + b) \mod 2^{m_f}$, i.e. $(2^{m_f} - 1) +_W 1 = 0$, which is the default addition on all CPUs. Therefore, we use saturating additions $a +_S b = \min\{a + b, 2^{m_f} - 1\}$, i.e. $(2^{m_f} - 1) +_S 1 = 2^{m_f} - 1$ [30]. Saturating arithmetic is slower than modular arithmetic, but its behaviour in numeric overflow is preferable as it approximates better the exact features of the embedding. Analogously to SPINE, the if-condition, and the $\boldsymbol{\delta}_{\text{TMP}}$ and $\boldsymbol{\varphi}_L$ saturating additions are implemented by using two single-word compare-and-swap (CAS) (lines 6 through 11). Once the two inner loops are completed, we proceed to reassign the temporary frontier ξ_{TMP} to ξ and the iteration count difference $\boldsymbol{\delta}_{\text{TMP}}$ to $\boldsymbol{\delta}$ (lines 12 and 13). The algorithm concludes by returning the computed co-occurrence counts $\boldsymbol{\varphi}_L$.

Theorem 5.7.4. *The worst-case time and memory upper bounds of algorithm 6 are* $T_{\boldsymbol{\lambda}} = O(\omega(|V| + |E|))$ *and* $S_{\boldsymbol{\lambda}} = (2 \cdot m_V + 3 \cdot m_f)|V| = \Theta(|V| \log|V|)$ *bits, respectively, if* $m_f = O(m_V)$

We observe that the values of the features increase exponentially w.r.t ω, quickly maxing out the features of nodes reachable starting from the nodes $s \in L$. As such, practical values of ω are very low $\omega \leq 5$ and therefore the asymptotic worst case time complexity $T_{\boldsymbol{\lambda}}$ is closer to $T_{\boldsymbol{\lambda}} = O(|V| + |E|)$.

Algorithm 6 ω-Hops WINE

```
    Input landmark $L$, window size $\omega$, method to iterate neighbours $\mathcal{N}(v)$
    Output co-occurrence counts vector $\boldsymbol{\varphi}_L$ from $L$
 1: $\boldsymbol{\varphi}_L \leftarrow \boldsymbol{\delta} \leftarrow \boldsymbol{u}_L$            ▷ $\Theta(|V| + |L|/t)$
 2: $\xi \leftarrow L$
 3: for $\omega$ times do                                          ▷ $O(\omega(|V| + |E|)$
 4:     $\boldsymbol{\delta}_{\text{TMP}} \leftarrow$ vector filled with $|V|$ zeros                ▷ $\Theta(|V|)$
 5:     $\xi_{\text{TMP}} \leftarrow$ empty frontier
 6:     for $v \in \xi$ do in parallel                                ▷ $O(|V| + |E|)$
 7:         for $w \in \mathcal{N}(v)$ do
 8:             if $\boldsymbol{\delta}_{\text{TMP}}[w] = 0$ then
 9:                 $\xi_{\text{TMP}}$.push($w$)
10:             $\boldsymbol{\delta}_{\text{TMP}}[w] \mathrel{+}=_S \boldsymbol{\delta}[v]$
11:             $\boldsymbol{\varphi}_L[w] \mathrel{+}=_S \boldsymbol{\delta}[v]$
12:     $\boldsymbol{\delta} \leftarrow \boldsymbol{\delta}_{\text{TMP}}$
13:     $\xi \leftarrow \xi_{\text{TMP}}$
14: return $\boldsymbol{\varphi}_L$
```

Two-Hops WINE

We observe that when the first round of co-occurrence counts completes from L, the vector $\boldsymbol{\varphi}_L$ contains, for each node, the number of predecessors that are in L, i.e. $\boldsymbol{\varphi}_L = \boldsymbol{u}_L A$. Therefore, when $\omega = 2$, it holds that $\boldsymbol{\varphi}_L$ is always equal to the co-occurrences variation vector $\boldsymbol{\delta}$, and so we can reduce the memory requirements by avoiding it altogether. We will use instead a frontier δ_{TMP} containing the temporary co-occurrence variations, which we populate with the counts of nodes $v \in \xi$.

We define the *zip*(ξ_1, ξ_2) operation as the set of tuples of values of ξ_1 and ξ_2, frontiers that by construction share the same index set I: *zip*$(\xi_1, \xi_2) = \{(a_i, b_i) \mid a_i \in \xi_1, b_i \in \xi_2 \ \forall i \in I\}$. We iterate on the zipped frontiers $(v, c) \in$ *zip*$(\xi, \delta_{\text{TMP}})$, and directly increase the counts of the neighbours $\mathcal{N}(v)$ by c (lines 12 through 14). We terminate by returning the computed co-occurrence counts $\boldsymbol{\varphi}_L$. We observe that unreached nodes and, more specifically, nodes in distinct connected components for the nodes in L and singletons will have co-occurrence counts equal to zero. As per algorithm 5.7.4, all additions are saturating atomic additions to avoid numeric overflows, and CAS is employed to emulate these operations.

The SUSS-based approximation described for SPINE (section 5.7.3) also applies to Two-hops WINE.

Theorem 5.7.5. *The worst-case time and memory upper bounds of algorithm 7 are* $O(|V| + |E|)$ *and* $(m_V + 2 \cdot m_f)|V|$ *bits, respectively.*

Analogously to SPINE, in practice $|\xi| = |\delta_{\text{TMP}}| \ll |V|$. $|\xi| = |\delta_{\text{TMP}}| \approx |V|$ exclusively in degenerate topologies such as stars. Since the two-hops WINE has lower complexities than WINE, the complexity constraints are respected.

Algorithm 7 Two-Hops WINE

```
    Input landmark $L$, method to iterate neighbours $\mathcal{N}(v)$
    Output co-occurrence counts vector $\varphi_L$ from $L$
 1: $\varphi_L \leftarrow$ vector filled with $|V|$ zeros                         $\triangleright \Theta(|V|)$
 2: $\xi \leftarrow$ empty frontier
 3: for $s \in L$ do in parallel                                                  $\triangleright O(|L| + |E|)$
 4:     for $w \in \mathcal{N}(s)$ do
 5:         if $\varphi_L[\mathrm{w}] = 0$ then
 6:             $\xi$.push($w$)
 7:         $\varphi_L[w] \mathrel{+}=_S 1$
 8:     $\varphi_L[s] \mathrel{+}=_S 1$
 9: $\delta_{\mathrm{TMP}} \leftarrow$ empty frontier
10: for $v \in \xi$ do in parallel                                                 $\triangleright O(|V|/t)$
11:     $\delta_{\mathrm{TMP}}$.push($\varphi_L[v]$)
12: for $(v, c) \in zip(\xi, \delta_{\mathrm{TMP}})$ do in parallel                $\triangleright O(|V| + |E|)$
13:     for $w \in \mathcal{N}(v)$ do
14:         $\varphi_L[w] \mathrel{+}=_S c$
15: return $\varphi_L$
```

5.8 Experiments

To show the scalability properties of ALPINE, we run the SPINE and WINE algorithms on the Wikipedia graph having about 1.2G nodes and 12.4G edges, using a commodity desktop computer. Then we show that ALPINE algorithms can significantly speed up the empirical time computation with respect to state-of-the-art embedding algorithms without losing too much in prediction performance and sometimes also being competitive with significantly more complex embedding algorithms.

We experimentally evaluated four concrete ALPINE-based algorithms on two tasks: node-label prediction (section 5.8.3), i.e. predicting the category of a given node, and edge prediction (section 5.8.4), i.e. predicting whether an edge exists between two given nodes.

The considered ALPINE-based algorithms are 2-Hops WINE and SPINE, with abstract landmarks obtained through either the node degrees or, when available, node categories schemas. We stress that these landmark schemas are most likely not optimal for the considered task, and the relative performance only constitutes a baseline.

We compare the computational time requirements and quality of the embedding obtained with the ALPINE-based methods and a wide variety of methods from the literature. The primary goal is to illustrate the significant computational time and memory gains obtained with ALPINE methods maintaining nevertheless reasonably competitive performance.

We include first and second-order LINE [119], DeepWalk [94] and WalkLets [95]-based GloVe [92], CBOW, and SkipGram [81], and two matrix factorization methods, namely HOPE [89] and NetMF [99].

We note that the considered methods from the literature fall prey to many of the scalability issues discussed within the introduction and addressed in detail in the presentation of the ALPINE framework (section 5.7). Thus, to provide a complete comparison within a reasonable amount of memory and time, we considered graphs commonly used in the literature with a relatively small size in the experiments. Nevertheless, we provide TSNE bi-disimensional dimensionality reduction of 100-dimensional node embedding of the Wikidata graph, obtained using Degree-based SPINE and WINE.

We have executed 10 holdouts for all tasks and graphs, considering multiple training and test splits to evaluate the performance variation relative to the number of known samples. While within this section, we report only bar plots with the F1 score over the test set, we provide tables with multiple other performance metrics such as balanced accuracy and AUROC in the appendix I.

For all of these methods, we employ our heavily optimized Rust implementations with Python bindings available from the *GRAPE* library described in chapter 2. All experiments were run on a machine with 12 cores (24 threads) with CPU AMD Ryzen 3900x @ 4.0 GHz and 128 GB of RAM on Linux with kernel version 5.15.25-1.

5.8.1 ALPINE embeds big graphs on commodity desktops

Embedding large graphs such as Wikidata, which we recall has 1.2G nodes and 12.4G edges, using any of the considered models from the literature, has memory requirements that surpass the capabilities of most available hardware. More specifically, to obtain a 100-dimensional node embedding using 4B floats, even when using optimization strategies that do not require the allocation of additional data structures, e.g. using SGD instead of Adam, it would require $100 \cdot four \cdot 1.2 = 480GB$ in first-order models such as first-order LINE and $960GB$ in second-order models such as CBOW, SkipGram or GloVe. Conversely, the memory requirements in methods such as SPINE are just $4.8GB + 1.2GB = 6GB$. High-end devices with at least $960GB$ are uncommon and come with a premium cost when rented on services such as Google Cloud. Furthermore, in most HPCs, while the total system RAM may be considerable, the RAM available in each computing node is often rather limited: in the Fukagu, for instance, it is 32GB, while in CINECA's Marconi100, it is 256GB.

While state-of-the-art embedding methods are unable to process the Wikidata graph also on well-equipped desktops, degree-based SPINE and WINE can compute a 100-dimensional node embedding of Wikidata in respectively 42 minutes and 12 minutes on a commodity desktop. Embeddings with a higher number of features are possible. For instance, a 1000-dimensional SPINE node embedding took ≈ 7 hours on the same desktop, showing strictly linear scaling with the increased number of features. Evaluating the obtained embedding on such a large graph is a non-trivial endeavour, as training simple models such as a Perceptron would still require significant computational time. Therefore, to provide proof of the quality of the node embedding obtained with SPINE and WINE, in this section, we present the TSNE decompositions of the Degree-based SPINE, and WINE node embedding of $40K$ edges uniformly sampled from the Wikidata graph, illustrating the edge labels and the clusters that can be identified (figures 5.1 and 5.2). The edge labels were never provided during training, as exclusively the graph topology was employed.

Wikidata is a scale-free graph, and we observe that topological properties such as shortest-path distance and unnormalized Jaccard Index also follow a scale-free distribution. Such a phenomenon leads to repeated feature patterns, which are optimal for compression algorithms such as gzip. We can employ gzip to compress the complete $1.2\cdot$

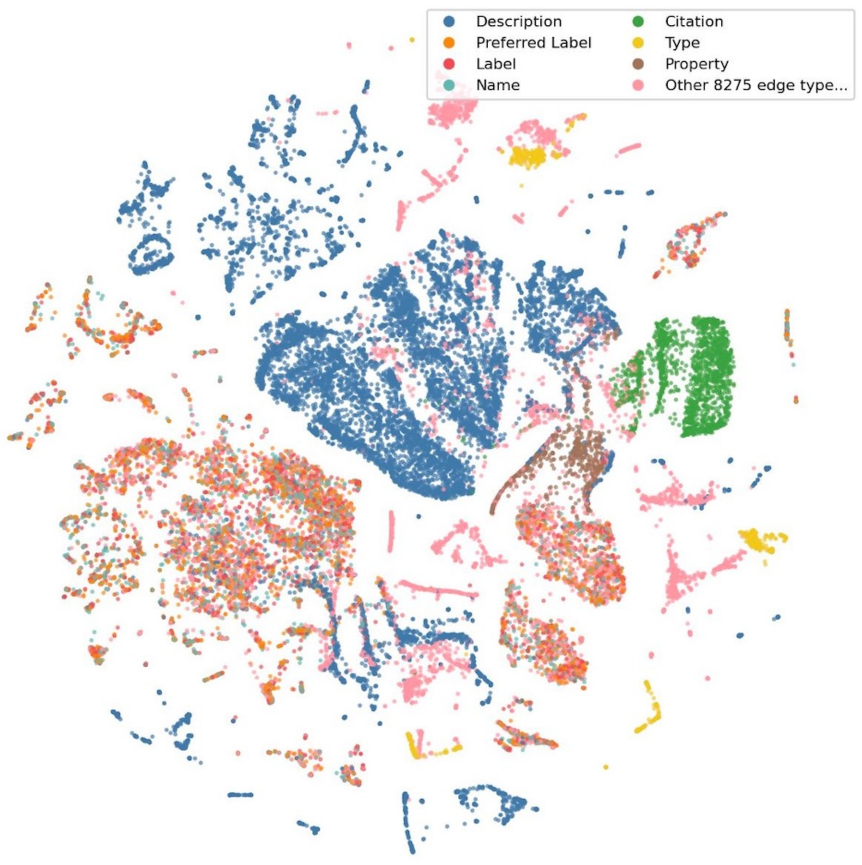


Figure 5.1: Edge embedding of Wikidata using Degree SPINE TSNE decompositions of 100 dimensional node embedding. Edge embeddings are obtained by concatenating the corresponding pairs of node embeddings.

100 = 120GB embedding obtained with SPINE and WINE down to 4.4GB and 4.9GB, respectively. Such a small file size allows us to share on Zenodo the two node embedding, which may be downloaded from `https://doi.org/10.5281/zenodo.7117686`.

5.8.2 ALPINE boosts empirical time complexity

In figure 5.3, we display the computation time necessary to compute the node embeddings on different graphs on a logarithmic scale.

Having premised that the implementations of the SOTA methods used are generally at least an order of magnitude faster than any other we could benchmark, we observe that the Walklets-based models, CBOW, SkipGram (SG) and GloVe, are the slowest in all graphs, followed by their analogous trained on DeepWalk (DW) samples. Next, we find that the methods based on matrix factorization NetMF are generally an order of magnitude lower than HOPE, based on shared neighbours counts. The fastest methods are those based on the ALPINE framework, and only LINE achieves a comparable time speed-up (Figure 5.3).

While the node-label-based ALPINE models appear to be the fastest, we must stress that they computed a much smaller embedding with cardinality equal to the number of labels in the respective graph, while all other methods produced a 100-dimensional node embedding.

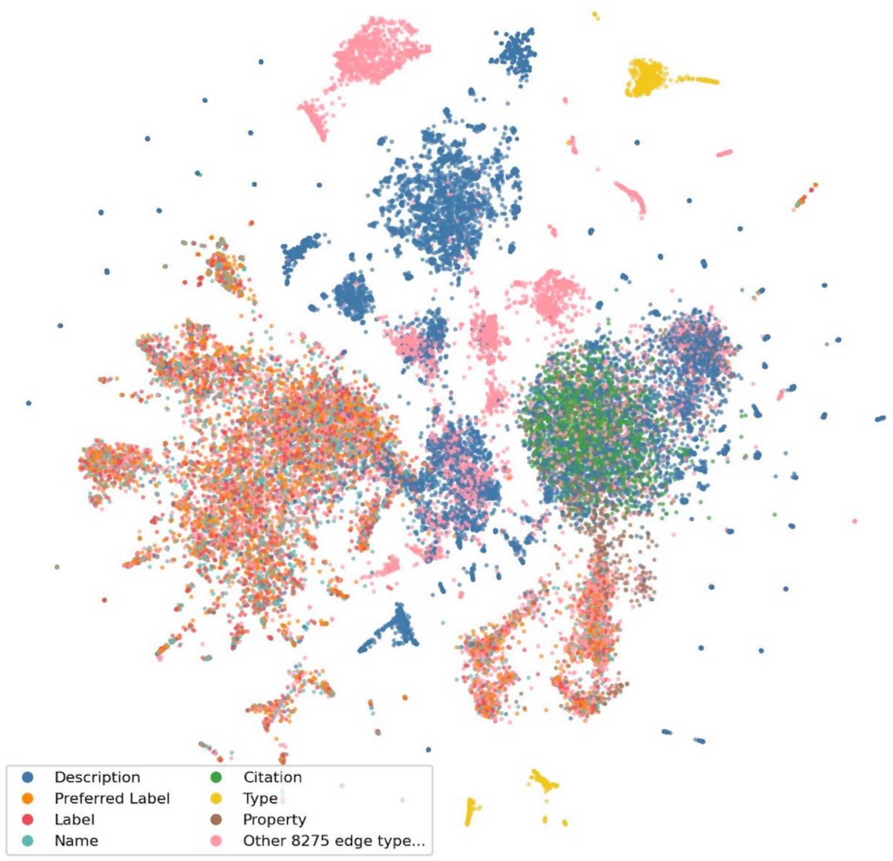


Figure 5.2: Edge embedding of Wikidata using Degree WINE TSNE decompositions of 100 dimensional node embedding. Edge embeddings are obtained by concatenating the corresponding pairs of node embeddings.

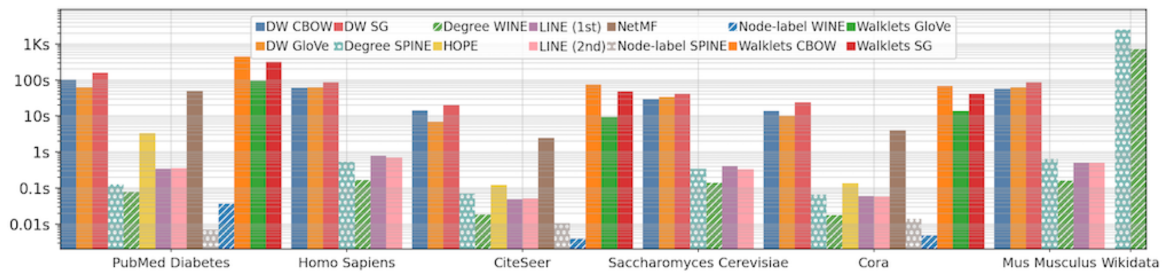


Figure 5.3: Computation time required by embedding: bar plot representing in logarithmic scale the seconds necessary to run a node embedding on different graphs and methods. A gap in the plot separates the bar plots relative to different graphs. Note that not all node embedding methods are present for all graphs. Except for the Node-label SPINE and WINE, all node embeddings are 100 dimensional. Considering the relatively high number of evaluated methods throughout the bar plots visualisations, we employ different patterns to help distinguish the ALPINE-based methods. More specifically, we will use dots for performance relative to SPINE and parallel inclined parallel lines for WINE. We denote that the time requirements of first and second-order LINE are generally comparable with the time requirements of ALPINE methods. Nevertheless, the former methods are neural networks and, as such, fall prey to many of the scalability limitations detailed in the introduction, such as the computational dependency of features. Conversely, 2-hops WINE and SPINE are extremely scalable and can complete a 100 dimensional node embedding of Wikidata in 12 minutes and 42 minutes.

The first and second-order LINE methods achieve computation times comparable with the ALPINE-based methods. However, they are neural networks and, as such, fall prey to many of the scalability limitations detailed in the introduction, such as the computational dependency of features. More specifically, ALPINE-based methods such as Degree SPINE require f and $2f$ less memory than first and second-order LINE, respectively.

On average, Degree-based SPINE and WINE are 152 times and 485 times faster than our optimized version of DeepWalk SkipGram, while requiring ≈ 800 times less memory.

5.8.3 Node-label prediction experiments

For the node-label prediction task, we have considered three citation graphs commonly used in the literature: Cora, CiteSeer and Pubmed Diabetes.

In citation graphs such as these, the nodes represent papers while the edges represent citations between the various publications. The papers are labelled according to their category.

Note that while additional node features exist relative to the three graphs, these have been omitted as we have focused strictly on node embedding involving the graph topology.

Cora [110] consists of 2708 scientific publications classified into one of seven classes. The citation network consists of 5429 edges. CiteSeer [104] consists of 3312 scientific publications classified into one of six classes. The citation network consists of 4732 edges. Pubmed Diabetes [84] consists of 19717 scientific publications from the PubMed database about diabetes classified into one of three classes. The citation network consists of 44338 edges.

The node embedding models from the literature we consider for this comparison are first and second-order LINE, DeepWalk (DW) GloVe, SkipGram (SG) and CBOW, Walklets GloVe, SG and CBOW, NetMF and HOPE. The ALPINE-based models we considered are Two-hops WINE and SPINE with landmarks based on degree and node labels. Since this is a node-label classification task, using the training labels may lead to some overfitting. Nevertheless, we employ categorical landmarks to provide a baseline.

All methods will compute 100-dimensional node embedding, except for WINE and SPINE, using categorical landmarks which will compute node embedding with a number of features f equal to the number of classes in the given graph, therefore $f \ll 100$.

We trained a Random Forest classifier model on the node embedding obtained by the abovementioned methods, characterized by 1000 estimator trees and a maximum depth of 10. All the models' parameters are reported in the appendix sections I.1 and I.2.1.

We considered a 40/60 and 80/20 split of the training and test set to evaluate the change, if any, in the performance with different percentages of available samples. We have executed 10 stratified Monte Carlo holdouts, i.e. the proportions of node labels are maintained between training and test set to avoid biases. As may be expected, the performance generally improves with a larger percentage of training, except for the ALPINE models with categorical landmarks, i.e. Label WINE and SPINE, which experience overfitting. Label SPINE suffers more strongly from overfitting as by using the training labels as landmark categories, the node embedding has a value of zero when the node is of a given category, making this a forced application.

The two worst-performing models are SPINE using categorical landmarks, which overfit for the reasons mentioned above, and second-order LINE. The performance of Degree-based SPINE generally falls in between the considered SOTA models. Even though the degree-based landmarks are a generic approach that is not mainly related to the considered task, it provides a highly scalable baseline with consistently decent performance.

5.8.4 Edge prediction

For the edge prediction task, we have considered three protein-protein interactions graphs from the STRING dataset [118], namely *Mus musculus*, *Homo sapiens* and *Saccharomyces cerevisiae*.

In protein-protein interaction graphs, the nodes represent proteins, while the edges represent interactions between the various proteins. In STRING graphs, the edges are weighted, representing a score between 100 and 1000, representing the confidence that the edge exists. The authors of the STRING dataset suggest considering the edges with a score of 700 or more.

The *Mus musculus* graph has 16633 nodes and 467530 edges, the *Homo sapiens* graph has 16814 nodes and 505968 edges, and finally, the smaller *Saccharomyces cerevisiae* graph, with 5966 nodes and 240772 edges.

Since the topology of the graphs changes in each holdout, it is necessary to recompute the node embeddings for each different holdout. Thus, edge prediction tasks are much more computationally intensive to evaluate, so we have restricted the node embed-

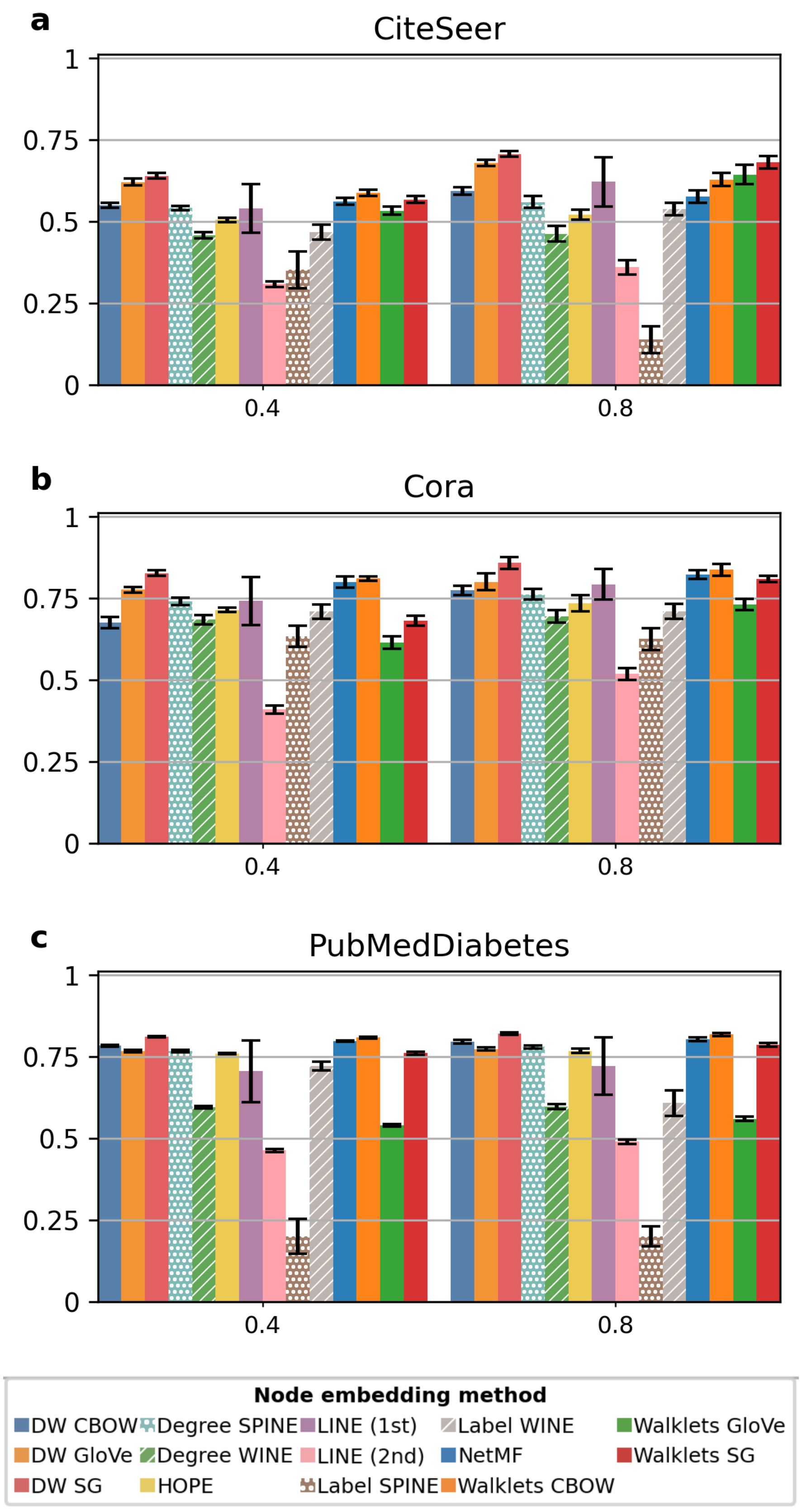


Figure 5.4: Node label prediction: bar plots showing the average F1 score across 10 stratified holdouts, performance on the test set. The error bars represent the standard deviation. The bars on the left are relative to a training-test split of 40/60, while 80/20 for those on the right. **(a.)** Average F1 scores relative to CiteSeer, **(b.)** Average F1 scores relative to Cora, **(c.)** Average F1 scores relative to Pubmed Diabetes.

ding models considered in this task compared to those employed within the node-label prediction task. Namely, we have dropped the models based on Walklets and matrix factorization, HOPE and NetMF. All methods computed 100-dimensional node embedding.

To represent a given edge, the relative node embedding have to be converted into a vector representing the edge. Many possible methods exist to convert node embedding into edge embedding, such as element-wise Hadamard product, average or euclidean distance, and it is often unclear which one may be better, especially when applying them to new node embedding algorithms. For this reason, we experimentally evaluate a wide array of edge embedding methods.

More specifically, we evaluated both method producing a scalar value, i.e. cosine similarity and euclidean distance, and the method producing an edge embedding vector, i.e. node embedding concatenation (creating a $2f$-dimensional edge embedding) and element-wise Hadamard product, L1, L2, addition, subtraction, maximum and minimum (creating f-dimensional edge embedding.)

While in the bar plot in figure 5.5, we only display for each node embedding method and training percentage the best performing edge embedding method, we report all of the performance obtained in the appendix I.3. The Hadamard product is employed when no edge embedding method is specified in the legend.

We have employed a Perceptron with Sigmoid activation as an edge prediction model. All the models' parameters are reported in the appendix.

We considered a 25/75, 50/50 and 75/25 split of the training and test set edges to evaluate the change, if any, in the performance with different percentages of available samples. We have executed 10 connected Monte Carlo holdouts, i.e. the training graph will always maintain the same connected components of the original graph to avoid biases. Moreover, the negative edges for the evaluation have been sampled following a scale-free distribution to avoid biases related to the degrees. They must be within the same connected component, as edges between two different connected components result in trivial predictions using node embedding methods. The number of non-existing edges, i.e. the negative class, sampled for the evaluation procedure has the same cardinality as the existing edges, i.e. the positive class, in the test partition.

The performance of the ALPINE node embedding models using the degree-based landmarks does not generally outperform the best state-of-the-art models in any of the considered edge prediction tasks but falls in the middle ground across all tasks and training percentages. The best performing ALPINE model appears to be the 2-hops WINE model, with cosine similarity-based edge embedding. While landmark schemas should be task-specific and not generic such as the one considered in this context, the achieved performance and especially the minimum time and memory requirements suggest it could be used as a baseline to evaluate newly proposed embedding methods.

We note that across all training splits and graphs, there appears to exist variation in the optimal edge embedding methods identified with all non-ALPINE methods. We identify as optimal edge embedding method for node embedding computed using SPINE, which we recall computes shortest path distances, the element-wise L2 distance, while for node embedding computed using 2-hops WINE, which we recall computes unnormalized Jaccard indices, the cosine similarity. In both features, the optimal edge embedding fits with the intuitive meaning of the feature.

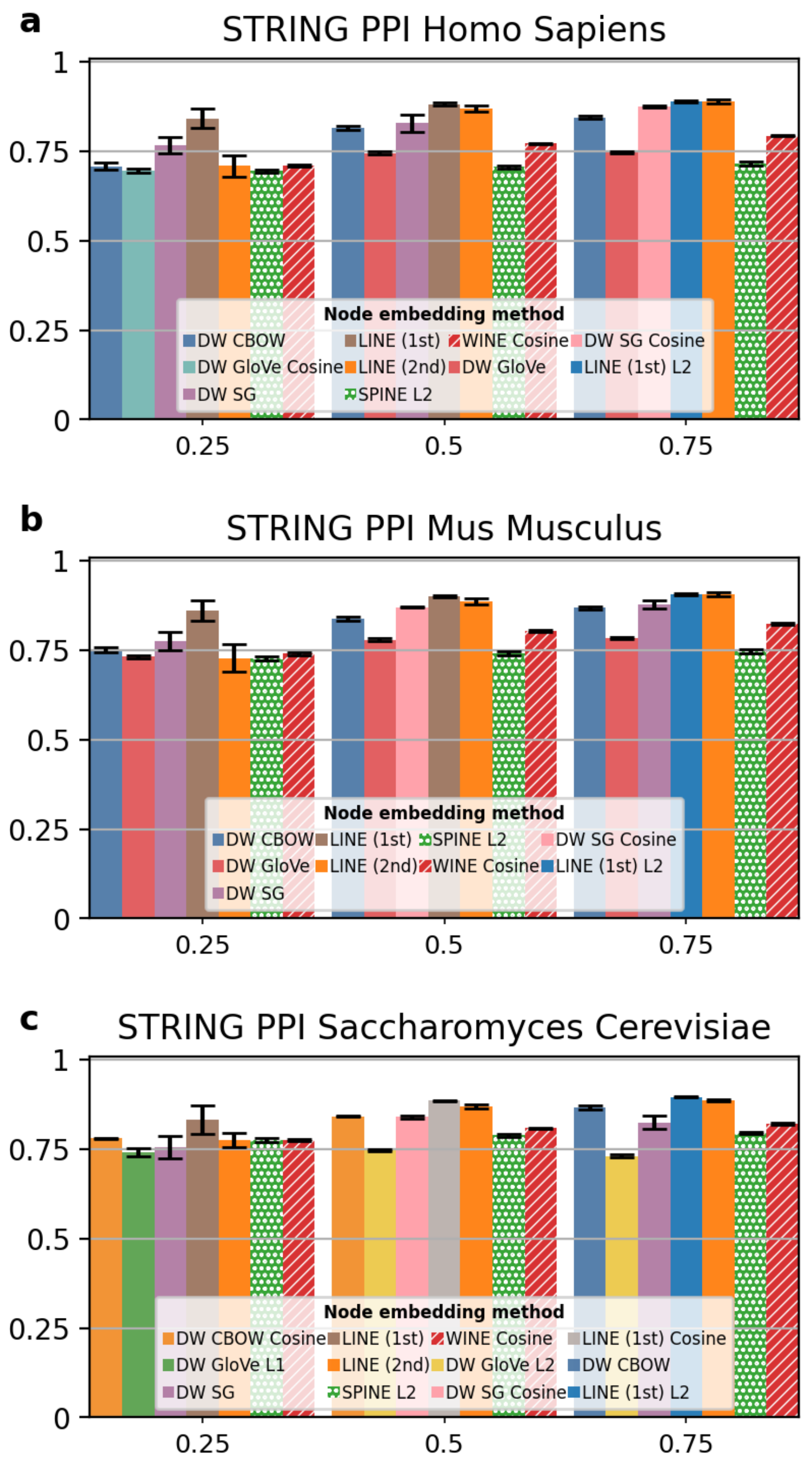


Figure 5.5: Edge prediction: bar plots showing the average F1 score across 10 connected holdouts, performance on the test set. The error bars represent the standard deviation. In all plots, the bars on the left are relative to a training-test split of 25/75, in the center 50/50 and those on the right 75/25. We report the average F1 scores obtained with the best-performing edge embedding method for each node embedding method and training percentage. Where no method is explicitly specified, Hadamard is employed. **(a.)** Average F1 scores relative to the *Homo sapiens* graph, **(b.)** Average F1 scores relative to *Mus musculus*, and **(c.)** Average F1 scores relative to *Saccharomyces cerevisiae*.

The current implementation of ALPINE and its components do show clear computational advantages over existing methods, but it does not show a clear improvement in task-related performance. This is because the landmark schemas used in the current implementation of ALPINE are not optimal for the tasks being considered, and only serve as a baseline for comparison.

To improve the task-related performance of ALPINE, the notion of landmarks can be defined in a number of ways. One approach would be to use automated learning algorithms to automatically identify the most appropriate node properties to use as landmarks. This could involve using techniques such as reinforcement learning or evolutionary algorithms to search for the optimal set of landmarks for a given task.

This could provide more information to the embedding algorithm and potentially improve classifiers performance on tasks such as node-label prediction and edge prediction.

Overall, there are many ways in which landmarks may be created to improve the task-related performance of ALPINE, and further research is needed to explore these possibilities.

5.9 Theorems' proofs

5.9.1 Constant amortized time push of a vector

Lemma 5.9.1. *Pushing an element ε onto a vector $\boldsymbol{a}$ takes amortized constant time.*

$$\boldsymbol{a}.push(\varepsilon) \in \Theta(1)$$

Proof. If the vector $\boldsymbol{a}$ is not full, appending a value takes constant time, otherwise, it has linear worst-case complexity $\Theta(|\boldsymbol{a}|)$ w.r.t. the size of the vector $|\boldsymbol{a}|$ due to the need for reallocation, but this happens only once every $|\boldsymbol{a}|$ appends. Therefore, the amortized time complexity is ${}^{1}/_{\Theta(|\boldsymbol{a}|)}(\Theta(|\boldsymbol{a}|) - 1)\Theta(1) + \Theta(|\boldsymbol{a}|) = \Theta(1)$. □

Lemma 5.9.2. *Pushing an element ε onto a vector $\boldsymbol{a}$ takes approximately constant time on modern memory allocators.*

$$\boldsymbol{a}.push(\varepsilon) \in \Theta(1)$$

Proof. While the initial allocation of a vector $\boldsymbol{a}$ takes $\Theta(|\boldsymbol{A}|)$, the allocation for new vectors can and often do reuse the allocations of previously freed vectors making reallocations events less frequent as the computation proceeds. Thus, in practice, we can approximate its worst-case complexity as constant time. □

Lemma 5.9.3. *Pushing an element ε onto a parallel frontier ξ takes approximately constant time on modern memory allocators.*

$$\xi.push(\varepsilon) \in \Theta(1)$$

Proof. The frontier is composed of t vectors, one for each thread, and as such, there is no synchronization overhead, and the lemma 5.9.2 directly applies. □

Lemma 5.9.4.

$$\frac{n^n}{e^{n-1}} \leq n! \leq \frac{n^{n+1}}{e^{n-1}} \qquad \forall n \in \mathbb{N}$$

Proof. **Upperbound:** Let's first state this simple fact that holds for all $k \geq 0$:

$$1 + k \leq e^k \implies \frac{k}{k+1} \leq e^{-\frac{1}{k+1}}$$

Since both $1 + k$ is non decreasing, we can construct two products that will eventually yield our result:

$$\prod_{k=1}^{n-1} \frac{k}{(k+1)} \leq \prod_{k=1}^{n-1} e^{-\frac{1}{k+1}}$$

Thanks to the distributive property of multiplication, we can elevate to $k+1$ products and thus the internal factors:

$$\prod_{k=1}^{n-1} \left(\frac{k}{(k+1)}\right)^{k+1} \leq \prod_{k=1}^{n-1} \left(e^{-\frac{1}{k+1}}\right)^{k+1}$$

The right-hand side can be trivially simplified:

$$\prod_{k=1}^{n-1} \left(\frac{k}{(k+1)}\right)^{k+1} \leq \prod_{k=1}^{n-1} \frac{1}{e} = \frac{1}{e^{n-1}}$$

Let's simplify the left-hand side. We can define:

$$a_k = \left(\frac{k}{(k+1)}\right)^{k+1} = k\frac{k^k}{(k+1)^{k+1}}$$

So we can expand the product as:

$$\begin{aligned} a_k a_{k+1} &= \left(k\frac{k^k}{\cancel{(k+1)^{k+1}}}\right)\left((k+1)\frac{\cancel{(k+1)^{k+1}}}{(k+2)^{k+2}}\right) \\ &= k(k+1)\frac{k^k}{(k+2)^{k+2}} \end{aligned}$$

So by induction:

$$\prod_{i=k}^{\gamma} a_i = \frac{\gamma!}{(k-1)!}\frac{k^k}{(k+\gamma+1)^{k+\gamma+1}}$$

Which when evalued with $k = 0$ and $\gamma = n - 1$ yields:

$$\prod_{k=1}^{n-1} \left(\frac{k}{(k+1)}\right)^{k+1} = \frac{(n-1)!}{n^n} = \frac{n!}{n^{n+1}}$$

Back to the disequation:

$$\frac{n!}{n^{n+1}} \leq \frac{1}{e^{n-1}} \implies n! \leq \frac{n^{n+1}}{e^{n-1}}$$

Lowerbound: Let's first state this simple fact that holds for all $k \geq 0$:

$$1 + k \leq e^k \implies \frac{k+1}{k} \leq e^{\frac{1}{k}}$$

Since both $1+k$ is non decreasing, we can construct two products that will eventually yield our result:

$$\prod_{k=1}^{n-1} \frac{k+1}{k} \leq \prod_{k=1}^{n-1} e^{\frac{1}{k}}$$

Thanks to the distributive property of multiplication, we can elevate to k both internal factors:

$$\prod_{k=1}^{n-1} \left(\frac{k+1}{k}\right)^k \leq \prod_{k=1}^{n-1} \left(e^{\frac{1}{k}}\right)^k$$

The right-hand side can be trivially simplified:

$$\prod_{k=1}^{n-1} \left(\frac{k+1}{k}\right)^k \leq \prod_{k=1}^{n-1} e = e^{n-1}$$

Let's simplify the left-hand side. We can define:

$$a_k = \left(\frac{k+1}{k}\right)^k = \frac{1}{k}\frac{(k+1)^k}{k^k}$$

So we can expand the product as:

$$\begin{aligned} a_k a_{k+1} &= \left(\frac{1}{k}\frac{\cancel{(k+1)^k}}{k^k}\right)\left(\frac{1}{k+1}\frac{(k+2)^{k+1}}{\cancel{(k+1)^{k+1}}}\right) \\ &= \frac{1}{k(k+1)}\frac{(k+2)^{k+1}}{k^k} \end{aligned}$$

So by induction:

$$\prod_{i=k}^{\gamma} a_i = \frac{(k-1)!}{\gamma!}\frac{(k+\gamma+1)^{k+\gamma}}{k^k}$$

Which when evalued with $k=0$ and $\gamma = n-1$ yields:

$$\prod_{k=1}^{n-1} \left(\frac{k}{(k+1)}\right)^{k+1} = \frac{n^{n-1}}{(n-1)!} = \frac{n}{n}\frac{n^{n-1}}{(n-1)!} = \frac{n^n}{n!}$$

Back to the inequality:

$$\frac{n^n}{n!} \leq e^{n-1} \implies n! \geq \frac{n^n}{e^{n-1}}$$

□

Lemma 5.9.5. *For any pair of integer $n, k \in \mathbb{N}$ such that $n \geq k$ it holds that:*

$$n^{-k}\frac{n!}{(n-k)!} = \frac{n!}{(n-k)!n^k} \leq ne^{-k}$$

Proof. we apply upper-bound from lemma 5.9.4 to the numerator factorial and the lower-bound from lemma 5.9.4 to the factorial in the denominator to obtain an upper-bound of $\binom{n}{k}$:

$$\begin{aligned}
\frac{n!}{(n-k)!n^k} &\leq n^{-k}\frac{\frac{n^{n+1}}{e^{n-1}}}{\frac{(n-k)^{n-k}}{e^{n-k-1}}} \\
&= n^{-k}\frac{n^{n+1}}{e^{n-1}}\frac{e^{n-k-1}}{(n-k)^{n-k}} \\
&= n^{-k}\frac{e^{n-k-1}}{e^{n-1}}\frac{n^{n+1}}{(n-k)^{n-k}} \\
&= n^{-k}\frac{1}{e^k}\frac{n^{n+1}}{(n-k)^{n-k}} \\
&\leq n^{-k}\frac{1}{e^k}\frac{n^{n+1}}{n^{n-k}} \\
&= n^{-k}\frac{n^{k+1}}{e^k} \\
&= ne^{-k}
\end{aligned}$$

Therefore:

$$\frac{n!}{(n-k)!n^k} \leq ne^{-k}$$

□

Lemma 5.9.6. *The probability p that t threads have c collision writing to uniformly sampled elements from a vector of size $|V|$, where each values uses m_V bits is:*

$$p = \frac{l!}{(l-t+c)!l^{t-c}} \leq Le^{c-t}$$

where $l = {}^{|V|}/_{\lceil {}^{64bytes}/_{m_V\, bits}\rceil}$ is the number of cache-lines needed to to store contiguosly the $|V|$ values on x86_64 CPUS. This theorem holds if, and only if, the number of cache-lines L is bigger than the number of threads, i.e. $\lceil {}^{|V|}/_k \rceil \geq t - c$, otherwise $P = 1$ due to the pigeon-hole principle.

Proof. An atomic operation might fail if, and only if, two or more threads write to the same cache-line, which on x86_64 is 64 bytes. Thus on the same cache-line we will have the bits of $k = \lceil {}^{64\text{bytes}}/_{m_V\text{bits}} \rceil$ different features. Therefore we can simplify the problem to the probability that each thread chooses a different cache-line, and the $|V|$ features will be stored contiguously using ${}^{|V|}/_k$ cache-lines.

$$p = \frac{l!}{(l-t+c)!l^{t-c}}$$

e.g. the server used for all the experiment has $t = 24$ threads, WikiData has $1.2G$ nodes, therefore $m_V = \lceil \log_2(1.2G) \rceil = 31\text{bits}$. So on a cache-line there are bits of $k = \lceil {}^{64\text{bytes}}/_{31\text{bits}} \rceil = 3$ features.

$$P = \frac{\frac{(1.2G/3)!}{(1.2G/3-24)!}}{(1.2G/3)^{24}} \approx \frac{4.537 \times 10^{182}}{2.815 \times 10^{206}} \approx 1.611 \times 10^{-24}$$

Therefore, using lemma 5.9.5 we obtain:

□

Lemma 5.9.7. *The expected number of iterations k a thread will have to execute the CAS operation to win a tie is:*

$$p(k,t) \geq \left(1 - \frac{1}{l}\right)^k$$

Proof. The CAS operation is the instruction LOCK CMPXCHG on x86_64 CPUs.

The hardware usually does not provide any mechanism to avoid lock-starvation [1], Therefore, if two or more threads try to execute a CAS on the same memory cell, only one will succeed, and all the others will have to try again. Thus, a thread will retry to execute CAS until it succeeds. Once a competitor thread wins the tie, it will have to "sample" a new cache line to write to, so it might or might not be the same. Therefore the probability of winning in the beginning is:

$$p(0,t) = \frac{l!}{(l-t)!l^t} \qquad p(k,1) = 1$$

Each time we fail, the probability follows this recurrence relation:

$$p(k,t) = \frac{1}{l}p(k-1,t) + \frac{l-1}{l}p(k-1,t-1)$$

Therefore asymptotically, we know that:

$$p(k,t) \geq \frac{l-1}{l}p(k-1,t-1) = \left(\frac{l-1}{l}\right)^k = \left(1 - \frac{1}{l}\right)^k$$

□

5.9.2 Degree landmarks complexities proof

Theorem 5.9.8. *6.1 The worst-case time and memory upper bounds for algorithm 3 are $\Theta(|V|)+T_{sort}(|V|)+O(t)$ and $2m_V \cdot |V| + S_{sort}(|V|) + O(t)$ bits $= \Theta(|V|\log|V|)+O(t)$, respectively, if $m_f = O(m_V)$.*

Proof. Algorithm 3 begins with the allocation of the *nodes* vector, which takes $\Theta(|V|)$ time and $m_V|V| + O(1)$ (line 1). Then we sort it in-place, which by definition (section 5.7.2) takes $T_{\text{sort}}(|V|)$ time and $S_{\text{sort}}(|V|)$ bits (line 2). The main loop takes $\Theta(|V|)$ time as all its constituent operations take constant time (line 6). More specifically, the computation of the degree is guaranteed to be constant-time thanks to the CSR assumption, and the push on the vector L takes amortized constant time (lemma 5.9.2).

[1]Intel Manual 3A Section 8.1 LOCKED ATOMIC OPERATIONS

The landmark L is a subset of nodes $L \subseteq V \implies |L| \leq |V|$ and thus its memory requirements are $m_V|L| \leq m_V|V|$.

All other lines take constant time and memory due to the PRAM assumption.

Therefore the total time complexity is:

$$\underbrace{\Theta(|V|)}_{\text{nodes allocation}} + \underbrace{T_{\text{sort}}(|V|)}_{\text{sort}} + \underbrace{\Theta(|V|)}_{\text{loop}} + O\left(t\right) = O(|V|\log|V|) + O(t)$$

The total memory complexity is:

$$\underbrace{m_V|V|}_{\text{nodes}} + \underbrace{m_V|V|}_{\text{landmark}} + \underbrace{S_{\text{sort}}(|V|)}_{\text{sort}} + O\left(t\right) = \Theta(|V|\log|V|) + O(t)$$

The $O(t)$ terms cover the cost of time to communicate the processors and the stack and ancillary data structures needed for each processor. □

5.9.3 Categorical landmarks complexities proof

Theorem 5.9.9. *6.2 The worst-case time and memory upper bounds of algorithm 4 are $\Theta(^{|C|\cdot|V|}/_t)$ and $m_V \cdot \max_{k\in C}|\{v \mid c(v) = k\}|bits = \Theta(|V|\log|V|)$, respectively, if $m_f = O(m_V)$.*

Proof. The push operation in the core of the inner loop of Algorithm 4 is executed in amortized constant time. The inner loop is executed $|V|$ times. We assume that the lookup of the class of each node takes constant time and that the vector L is practically implemented as a parallel frontier which has a constant-time push (lemma 5.9.3). Since each iteration of the inner loop is independent of one another, it can be parallelized across all t threads. Therefore, the time complexity of the inner loop is $\Theta(^{|V|}/_t)$.

The outer loop is executed $|C|$ times, and, since in the inner loop each iteration is independent, the total time complexity is:

$$\frac{1}{t} \cdot \underbrace{|C|}_{\text{outer}} \cdot \underbrace{|V|}_{\text{inner}} \cdot \Theta(1) + O(t) = \Theta(^{|C||V|}/_t)$$

We can reuse the same frontier for all the $|C|$ landmarks, therefore its maximum size is $\max_{k\in C}\{v : c(v) = k\} \leq |V|$. Therefore, its memory requirement in bits is:

$$m_V \max_{k\in C}\{v : c(v) = k\} \leq m_V|V| = \Theta(|V|\log|V|)$$

□

5.9.4 SPINE complexities proof

Theorem 5.9.10. *6.3 The worst-case time and memory upper bounds of algorithm 5 are $T_{\boldsymbol{\lambda}} = O(|V| + |E|)$ and $S_{\boldsymbol{\lambda}} = (m_V + m_f)|V|$ bits, respectively.*

Proof. The algorithm 5 starts with the allocation of the distances vector D initialized with $MAX = 2^{m_f} - 1$, which requires $\Theta(|V|)$ time and $m_f|V|$ bits (line 1).

Then we initialize the distances of the nodes v included in the landmark L, which can be concurrently executed by upwards to $t \leq |L|$ threads, which requires $\Theta(^{|L|}/_t)$ time (lines 2 and 3).

Pushing an element onto the frontier ξ_{TMP} requires constant time (lemma 5.9.3), and the atomic saturating additions have amortized constant time (lemma 5.9.7).

The worst-case time complexity of the core loop (lines 6 to 16) is $O(|V| + |E|)$ as at most we can iterating across all $|V|$ nodes and their $|E|$ edges once.

In the worst case, the union of the two frontiers ξ and ξ_{TMP} will collect all nodes in the graph, hence $|\xi| + |\xi_{\text{TMP}}| \leq |V|$, and the relative memory requirements will be $O(m_V|V|)$.

All other operations take constant time and memory due to the PRAM assumption.

Therefore, the synthetic worst-case time complexity is:

$$\underbrace{\Theta(|V| + ^{|L|}/_t)}_{\boldsymbol{\varphi}_L \text{ init}} + \underbrace{O(|V| + |E|)}_{\text{main loop}} = O(|V| + |E|)$$

Furthermore, the worst-case space requirements are:

$$\underbrace{m_f \cdot |V|}_{\boldsymbol{\varphi}_L} + \underbrace{m_V \cdot |V|}_{\text{Frontiers } \xi + \xi_{\text{TMP}}} = (m_f + m_V)\,|V|$$

□

5.9.5 WINE complexities proof

ω-Hops WINE complexities proof

Theorem 5.9.11. *6.4 The worst-case time and memory upper bounds of algorithm 6 are $T_{\boldsymbol{\lambda}} = O(\omega(|V|+|E|))$ and $S_{\boldsymbol{\lambda}} = (2\cdot m_V + 3\cdot m_f)|V| = \Theta(|V|\log|V|)$ bits, respectively, if $m_f = O(m_V)$.*

Proof. The algorithm starts with the allocation of the co-occurrence counts vectors $\boldsymbol{\varphi}_L$ and $\boldsymbol{\delta}$, both with size $|\boldsymbol{\varphi}_L| = |\boldsymbol{\delta}| = |V|$, which are both initialized with the user-preference vector $\boldsymbol{u}_L$ associated to the landmark L. The allocation requires $\Theta(|V|)$ time, while the initialization to $\boldsymbol{u}_L$ requires $\Theta(^{|L|}/_t)$, which tallies to $\Theta(|V| + ^{|L|}/_t)$ time. Moreover, the allocation of the two vectors requires $2 \cdot m_f|V|$ bits (line 1).

In the main loop (lines 3 to 13) starts with the initialization of the variations vector δ, which requires $\Theta(|V|)$ time and $m_V \cdot |V|$ bits.

In the worst case, the frontier ξ_{TMP} will collect all nodes in the graph, hence $|\xi| \leq |V|$, and the relative memory requirements will be $O(m_V|V|)$.

In the core loop (lines 3 to 11), pushing an element onto the frontier ξ_{TMP} is in constant time (lemma 5.9.3), and the atomic saturating additions have amortized constant time (lemma 5.9.7). Therefore, the worst-case time complexity of the core loop is $O(|L|+|E|)$ as by iterating across all elements in the frontier ξ in the worst case, we may iterate across all the graph edges sequentially, such as in a star graph where the landmark contains only the center of the star. When $|\xi| = 1$ there is no concurrent execution, and while the exploration of the neighbours $\mathcal{N}(v)$ of nodes $v \in \xi$ may be possible in parallel, it introduces a considerable implementation complexity.

The core loop is repeated ω times, hence the main loop has worst case complexity $O(\omega(|V| + |E|)$.

All other operations take constant time and memory due to the PRAM assumption.

Therefore, the synthetic worst-case time complexity is:

$$\underbrace{\Theta(|V| + {}^{|L|}/_{t})}_{\boldsymbol{\varphi}_L \text{ and } \boldsymbol{\delta} \text{ init}} + \underbrace{O(\omega(|V| + |E|))}_{\text{main loop}} = O(\omega(|V| + |E|))$$

Furthermore, the worst-case space requirements are:

$$\begin{aligned} &\underbrace{m_f \cdot |V|}_{\boldsymbol{\varphi}_L} + \underbrace{m_f \cdot |V|}_{\text{Counts } \boldsymbol{\delta}} + \underbrace{m_f \cdot |V|}_{\text{Variations } \boldsymbol{\delta}_{\text{TMP}}} + \underbrace{m_V \cdot |V|}_{\text{Frontier } \xi} = \\ &= (3m_f + m_V)\,|V| = \Theta(|V|(\log|V| + m_f) \end{aligned}$$

□

2-Hops WINE complexities proof

Theorem 5.9.12. *6.5 The worst-case time and memory upper bounds of algorithm 7 are $O(|V| + |E|)$ and $(m_V + 2 \cdot m_f)|V|$ bits, respectively.*

Proof. The algorithm starts with the allocation of the co-occurrence counts vector $\boldsymbol{\varphi}_L$, with size $|\boldsymbol{\varphi}_L| = |V|$, and therefore requires $\Theta(|V|)$ time and $m_f|V|$ bits (line 1).

In the first loop (lines 3 to 8), pushing an element onto the frontier ξ is in constant time (lemma 5.9.3), and the atomic saturating additions have amortized constant time (lemma 5.9.7). Therefore, the worst-case time complexity of the first loop is $O(|L|+|E|)$ as by iterating across all landmarks in the worst case, we may iterate across all the graph edges sequentially, such as in a star graph where the landmark contains only the center of the star. When $|L| = 1$ there is no concurrent execution, and while the exploration of the neighbours $\mathcal{N}(v)$ of nodes $v \in \xi$ may be possible in parallel, it introduces a considerable implementation complexity.

In the worst case, the frontier ξ will collect all nodes in the graph, hence $|\xi| \leq |V|$, and the relative memory requirements will be $O(m_V|V|)$.

In the subsequent loop, we populate in parallel the frontier δ_{TMP}, where the concurrent push has constant time complexity (lemma 5.9.3). Thus, as $|\xi| \leq |V|$, its worst case time and memory complexities are $O({}^{V}/_{t})$ and $m_f|V|$ bits, respectively.

In the second loop, we iterate in parallel across the two frontiers ξ and δ, which have at most length $|V|$. As per the first loop, the saturating atomic additions have amortized constant time (lemma 5.9.7). Therefore, the worst-case time complexity of the second loop is $O(|V| + |E|)$.

All other operations take constant time and memory due to the PRAM assumption.

Therefore, the synthetic worst-case time complexity is:

$$\underbrace{O(|L| + |E|)}_{\text{first loop}} + \underbrace{O(|V| + |E|)}_{\text{second loop}} + \underbrace{\Theta(|V|)}_{\varphi_L} + \underbrace{O(|V|/t)}_{\text{populate } \delta_{\text{TMP}}} = \\ = O(|V| + |E|)$$

Moreover, the worst-case space requirements are:

$$\underbrace{m_f \cdot |V|}_{\varphi_L} + \underbrace{m_f \cdot |V|}_{\text{Variations } \delta_{\text{TMP}}} + \underbrace{m_V \cdot |V|}_{\text{Frontier } \xi} = \\ = (2m_f + m_V)\,|V| = \Theta(|V|(\log|V| + m_f)$$

□

Conclusions and future work

The first contribution of this thesis is the development of *GRAPE*, a software library designed to efficiently execute graph representation learning tasks, such as node embedding, edge prediction, and node-label prediction, on large graphs. *GRAPE* has been implemented in Rust with Python bindings to enhance user accessibility, and it focuses on scalability, enabling the execution of graph representation learning tasks on large graphs even on commodity hardware. This is achieved through holistic attention to synchronization-free parallelism, instruction-level parallelism based on SSE and AVX, efficient data structures, numerical stability, and mixed precision and MMAP where necessary. Among the many high-performance algorithms provided by *GRAPE*, the library includes *approximated* weighted DeepWalk, Walklets, and Node2Vec embedding models, which are capable of processing graphs containing high-degree nodes (degree $> 10^6$), an otherwise infeasible task using the corresponding *exact* algorithms, while still achieving edge prediction performance comparable to the exact versions. In addition, *GRAPE* can optionally use succinct data structures based on Elias-Fano to load graphs that would not otherwise fit in main memory, with memory usage close to the theoretical minimum.

The scalability improvements offered by *GRAPE* are particularly important for analyzing real-world biomedical knowledge graphs, which represent a central machine learning and computational biology challenge and often contain millions of nodes and edges that are beyond the capabilities of previous methods and software implementations. Compared to state-of-the-art algorithmic and software resources, *GRAPE* shows orders of magnitude improvements in empirical space and time complexity, as well as a statistically significant improvement in edge prediction and node label prediction performance.

GRAPE provides over 80,000 graphs from the literature and other sources, standardized interfaces for integrating third-party libraries, 61 node embedding methods, 25 inference models, and 3 modular pipelines to enable a FAIR and reproducible comparison of methods and libraries for graph processing and embedding. The scaling properties of *GRAPE* in relation to state-of-the-art resources have been evaluated through extensive experiments with real-world large graphs, including Wikipedia, the Comparative Toxicogenomic Database (CTD), and a large biomedical Knowledge Graph generated by PheKnowLator. *GRAPE* significantly outperforms state-of-the-art libraries in terms of empirical time and space complexity and edge prediction performance, and it is able to process large graphs even when other competing state-of-the-art resources fail.

As a second contribution for efficiently processing and analyzing big graphs, this thesis proposes a novel algorithmic framework, efficiently implemented in *GRAPE*, that we named ALPINE: Abstract Landmark Properties-Inferred Node Embedding. The breakthrough characteristics of this algorithmic framework allow us to deal with several issues affecting SOTA GRL methods:

1. The embedding features are independently computed, each from the others, thus overcoming the space and time complexity limitations due to their dependent computation.
2. Feature computation is based on "landmarks", i.e. sets of nodes representing meaningful concepts about the structure or the semantics of the underlying graph, thus assuring the explainability of the embeddings.
3. Small integers are used for embedded feature values: this assures a small memory footprint, hardware acceleration, and a good compression ratio because of the scale-free distribution of node degrees that often characterize biomedical networks.
4. ALPINE provides a "democratic" feature representation, thus avoiding the bias towards high degree nodes characteristic of embedding methods based on topological sampling.

Besides the algorithmic framework, we also provide two concrete ALPINE implementations, namely SPINE (Shortest Paths Inferred Node Embedding), based on the efficient computation of the shortest path distance from each node of the graph to the landmarks, and WINE (Windows Inferred Node Embedding), based on the efficient computation of the co-occurrences of each node and the landmarks in a breadth-first search within windows of a given size.

The presented ALPINE-based algorithms show outstanding scalability, allowing for unprecedented scale even on commodity desktop computers. The presented examples of landmark generation schemas achieve competitive performance with other state-of-the-art methods.

Nevertheless both *GRAPE* and ALPINE present drawbacks and limitations that we plan to consider in ongoing and future research work:

- *GRAPE* does not support dynamic graphs, i.e. efficient in-place addition and removal of nodes and edges without allocating a new graph. Generally, dynamic graph data structures have significantly higher memory requirements than static data structures and are outside the scope of this library. Hornet [20] is a notable graph library supporting efficient dynamic graphs.
- While the pipelines for easily designing and executing reproducible node-label, edge-label and edge prediction experiments can be readily executed in parallel across *SLURM* clusters and some of the supported node embedding models can be readily distributed across *SLURM* clusters, the library does not support any model designed for distributed computing systems such as *Apache SPARK* [114], *Map-Reduce* [35] or *HADOOP* [17]. Most commonly, distribution across computing clusters of node embedding models is difficult, requires random access to all values in the embedding, and is generally obtained through node partitioning and weight synchronization routines.
- *GRAPE* supports weighted heterogeneous multigraphs with multi-labelled nodes, i.e. there can be multiple edges between any two given nodes, the edges can be labelled and weighted, and the nodes can belong to multiple classes. Both the edges and nodes may have unknown classes. The generic data structures adopted for these metadata cover many use cases but are not optimized for specific

applications and can be significantly improved. Furthermore, use cases such as multiple weights, unknown weights, or multiple labels per single edge are not yet supported. We will expand the support for these features and, more generally, for more efficient data structures in future work.

- Some knowledge graphs provided variable metadata for nodes, edges, and in some instances, even for node and edge labels. Such highly variable metadata, which often contains many unknown (missing) values, require either generic inefficient data structures or efficient ad-hoc ones, which are highly dependent on the use case. Either use cases are currently outside the scope of *GRAPE*.
- The library comes equipped with node embedding, node-label, edge-label and edge prediction but does not currently include models for edge-weight prediction.
- Currently, *GRAPE* does not come equipped with models able to handle distributed gradient computation, which is commonly employed for anonymization systems.
- While the ALPINE algorithmic framework has been designed to allow straightforward feature engineering by selecting task-specific landmarks, this feature has not yet been explored concretely. In future work, we plan to collaborate with field experts to design ad-hoc landmark schemas for network medicine tasks on large graphs.
- ALPINE-based methods can compute node embedding for graphs with billions of nodes on commodity desktops, but currently, there are no classifier models for node-label or edge prediction able to scale on such large graphs. Future work will include models able to execute on such large instances.
- The orders of magnitude efficiency improvements realized by *GRAPE* opens the possibility for deploying real-time node embedding and classifiers to support the development of large biomedical ontologies. Moreover, it allows large-scale hyperparameters search for high-performance node embedding model selection.
- The ALPINE-based methods and, more generally, the *GRAPE* library are either actively used or planned to be used in several network medicine projects. More specifically, *GRAPE* is being used to research and measure biases present in synthetic lethality networks to predict synthetic lethality interactions for cancer therapy [2]. Moreover the methods proposed in this thesis can be used to significantly increase the size of the graphs employed to compute the Jaccard and Resnik scores composing the PhenoDigm scores [113] and *GRAPE* will be applied to drug target and drug repurposing tasks within the "National Center for Gene Therapy and Drugs based on RNA Technology" project funded by the NextGeneratioEU programme.

In conclusion both *GRAPE* and ALPINE represent a significant contribute to the design and implementation of scalable algorithms and software resources for graph processing and embedding, and we plan to further develop this research line to overcome the drawbacks and limitations previously discussed and to apply *GRAPE* and ALPINE to relevant bio-medical problems in the context of Network Medicine.

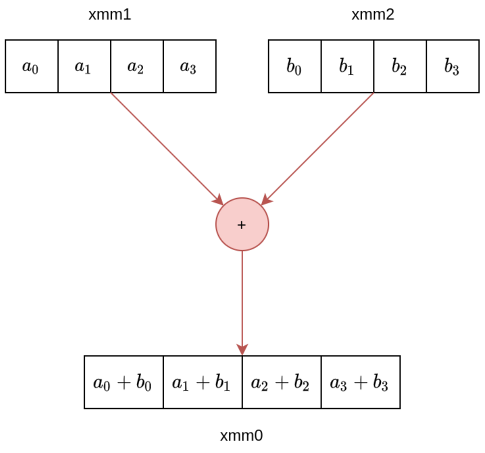
[2]In collaboration with Jackson Lab, USA

Appendix A

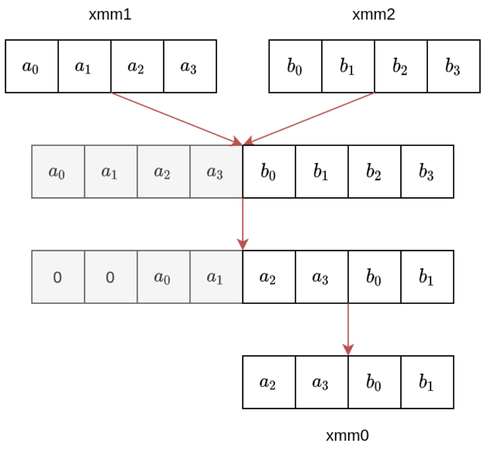
Efficient cumulative sum computation

The cumulative sum, also called prefix sum, is one of the most known examples in the field of parallel computing; in *Ensmallen* it is fundamental during the computation of random walks, to compute the un-normalized cumulative sum of the un-normalized transition probabilities to each node.

To obtain efficient computations, *Ensmallen*'s implementation exploits the Streaming SIMD Extensions (SSE 128-bit) instruction set, which is an extension of the SIMD instruction set for the x86_64 architecture. CPUs with SSE support have a special set of 8 128-bits registers, on which vector operations may be executed, that operate in parallel on 4 floats, therefore increasing the throughput by 4 times. The algorithm is based on two SSE instructions: *vaddps* (Fig. A.1a), which provides element-wise sum of two registers, each containing 4 32-bits floats, and *vpalignr* (Fig. A.1b), which concatenate two registers into a temporary register, shifts the result by a chosen number of bytes and then returns the lower 128-bits. The generalization of the algorithm for wider instruction sets such as AVX2 (256-bit) or AVX512 (512-bit) is limited by the lack of multi-lane logical shifts (the vpalignr instruction) which can be avoided by using opportunely shuffle and permutation instructions which introduces additional latency so the throughput don't scales linearly. While this does not improve the complexity of the algorithm it is almost ten times faster than the naive implementation and five time faster than the unrolled version. Loop unrolling [59] is an optimization technique for tight loops which increase the number of steps executed for each cycle, this reduces the overhead for the loop, allows to better exploit the CPU super-scalarity and reduces the number of branches. This technique increase the size of the function and thus if abused might degrade performances due to cache missing.



(a) vaddps xmm0, xmm1, xmm2


xmm1
xmm2
a_0 a_1 a_2 a_3
b_0 b_1 b_2 b_3
a_0 a_1 a_2 a_3 b_0 b_1 b_2 b_3
0 0 a_0 a_1 a_2 a_3 b_0 b_1
a_2 a_3 b_0 b_1
xmm0

(b) vpalignr xmm0, xmm1, xmm2, 2

Figure A.1: The two fundamental SIMD instructions for the prefix sum. Left: *vaddps* computes the elementwise sum of two registers, each containing 4 floats represented by 32 bits. Right: *vpalignr* concatenates two registers, shifts the result by a number of bytes provided as input and returns the lower 128-bits.

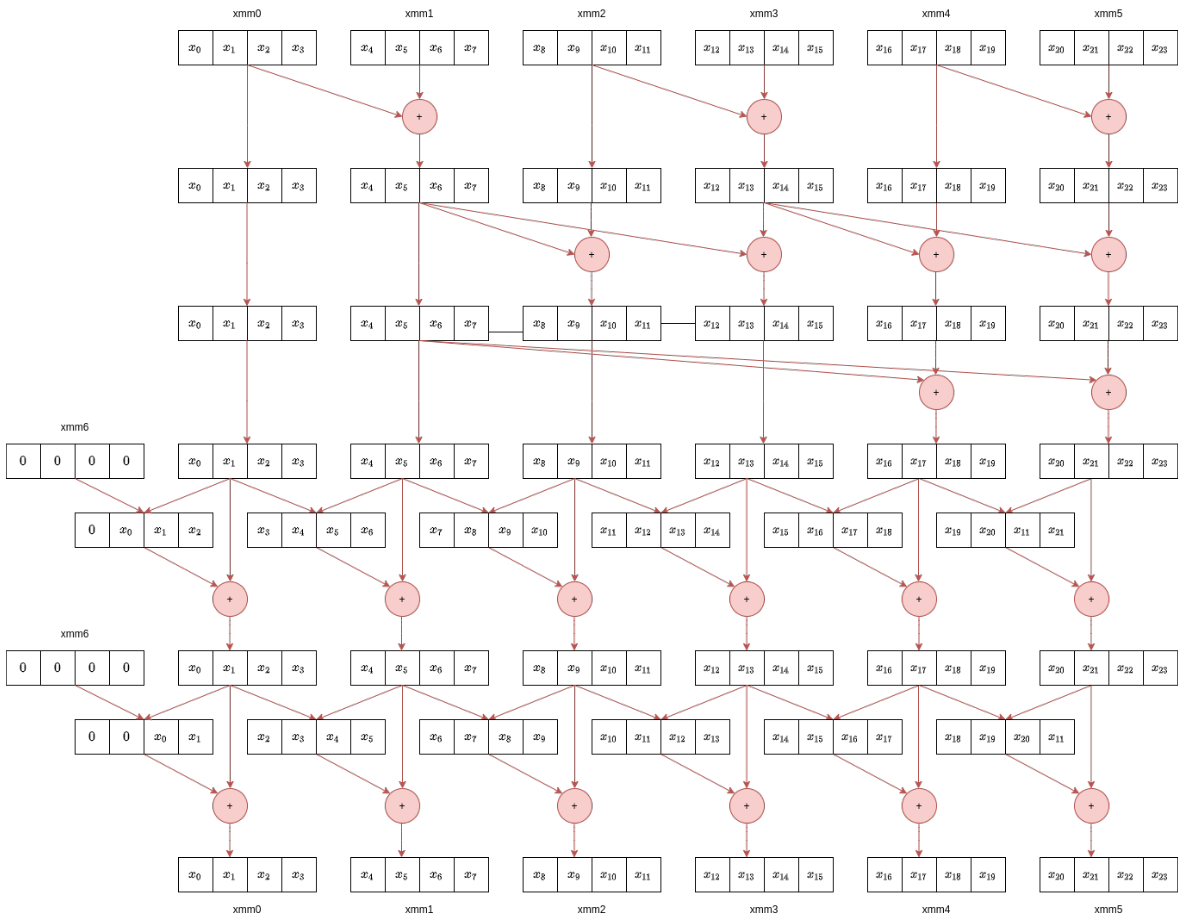


Figure A.2: SIMD prefix sum algorithm. To compute the prefix-sum of 24 float values, the prefix-sum we implemented uses mainly the *vaddps* and the *vpalignr* instructions. Note that, in the diagram the *vpalignr* instruction is represented in a compact way.

```
; first stage
vaddps xmm1, xmm0, xmm1
vaddps xmm2, xmm2, xmm3
vaddps xmm5, xmm4, xmm5
; second stage
vaddps xmm5, xmm3, xmm5
vaddps xmm4, xmm3, xmm4
vaddps xmm3, xmm1, xmm3
vaddps xmm2, xmm1, xmm2
; third stage
vaddps xmm5, xmm1, xmm5
vaddps xmm4, xmm1, xmm4
; fourth stage
vpalignr xmm7,  xmm0, xmm6, 12
vpalignr xmm8,  xmm1, xmm0, 12
vpalignr xmm9,  xmm2, xmm1, 12
vpalignr xmm10, xmm3, xmm2, 12
vpalignr xmm11, xmm4, xmm3, 12
vpalignr xmm12, xmm5, xmm4, 12
vaddps xmm0, xmm0, xmm7
vaddps xmm1, xmm1, xmm8
vaddps xmm2, xmm2, xmm9
vaddps xmm3, xmm3, xmm10
vaddps xmm4, xmm4, xmm11
vaddps xmm5, xmm5, xmm12
; fifth stage
vpalignr xmm7,  xmm0, xmm6, 8
vpalignr xmm8,  xmm1, xmm0, 8
vpalignr xmm9,  xmm2, xmm1, 8
vpalignr xmm10, xmm3, xmm2, 8
vpalignr xmm11, xmm4, xmm3, 8
vpalignr xmm12, xmm5, xmm4, 8
vaddps xmm0, xmm0, xmm7
vaddps xmm1, xmm1, xmm8
vaddps xmm2, xmm2, xmm9
vaddps xmm3, xmm3, xmm10
vaddps xmm4, xmm4, xmm11
vaddps xmm5, xmm5, xmm12
```

Figure A.3: Inner core of the SIMD prefix sum algorithm This code assumes that in the registers from xmm0 to xmm5 are loaded with the data and that xmm6 is zero filled. At the end of the computation, the result will also be in the registers from xmm0 to xmm5. We need 13 xmm registers but the SSE standard only provide 8 of them, luckily the AVX expansion increase this number to 16, thus making this algorithm possible.

Appendix B

Faster Pseudo-Random Numbers Generators (Vectorized xorshift)

Many of the algorithms inside of *Ensmallen*, e.g. the sampling of destination nodes during the random walk, or the generation of random negative edges for Skipgram [79] model, rely on the generation of random numbers.

Therefore, a random number generator algorithm could be a bottleneck if not efficiently implemented. To guarantee efficiency, in *Ensmallen* we use the two following fast pseudo-random number generators, Vigna's Xoshiro256+ [13] and the Marsaglia's Xorshift [76]. Xoshiro256+ has a shallower dependencies chain than Xorshift, which results in lower latency, while Xorshift has fewer instruction than Xoshiro256+, so that it can be implemented to achieve higher throughput. Therefore, Xoshiro256+ is faster in the generation of a single random value, e.g. during the sampling of the destination node in a step of the random walk. On the other side, Xorshift is faster in the generation of multiple random numbers, e.g. when generating the random negative edges for Skipgram. Modern CPUs allow to execute up to 4 different independent instructions in the same cycle and can have eight or more memory requests running at a time, so a common way to exploit the super-scalarity of modern CPUs and out-of-order execution is to interleave different instances of the same algorithm. This reduces the importance of the depth of dependency chain of an algorithm and favors the number of instructions. For these reasons, we interleave eight different instances of Xorshift and, to further improve its throughput, we implemented a vectorized version which exploits Intel's AVX2 instruction sets to execute 4 instances in parallel. An non-interleaved example of the vectorized Xorshift is illustrated in Fig. B.1. With the aforementioned implementation, we achieve a throughput of 256 random bytes (32 64-bits integers) at the cost of 4 concurrent cache misses (which are adjacent so the values should already be in the L1 cache). The complete implementation is available in figures B.3 and figures B.4; it achieves a throughput of more than 20 times higher than standard methods as shown in table B.1.

```rust
pub fn xorshift(mut seed: u64) -> u64 {
    seed ^= seed << 13;
    seed ^= seed >> 7;
    seed ^= seed << 17;
    seed
}
```

Figure B.1: The classic xorshift algorithm.

```asm
vmovdqu ymm0, ymmword ptr [rsi]
vpsllq ymm1, ymm0, 13
vpxor ymm0, ymm0, ymm1
vpsrlq ymm1, ymm0, 7
vpxor ymm0, ymm0, ymm1
vpsllq ymm1, ymm0, 17
vpxor ymm0, ymm0, ymm1
vmovdqu ymmword ptr [rsi], ymm0
```

Figure B.2: The vectorized xorshift which uses AVX2 to execute, through data parallellism, 4 64bits xorshifts.

Table B.1: The time taken by each method to generate a batch 32000 random 64-bits integers. This benchmark was executed on a single thread using Rust's default benchmark library which collects at least 50 samples. The performance difference, between the two considered CPUs, might be due to the fact that the AVX2 logical shifts instructions (vpsllq and vpsrlq) on Coffe Lake have twice the throughput when compared to Zen2.

	I7-8750H 4.1Ghz		Ryzen 3900x 4.0Ghz	
Method	Total Time (μs)	Throughput $\left(\frac{GB}{s}\right)$	Total Time (μs)	Throughput $\left(\frac{GB}{s}\right)$
thread rng	367.3 (±20.8)	0.7	349.3 (±20.66)	0.73
xorshift	48.1 (±7.89)	5.3	48.0 (±0.067)	5.3
xorshift avx2	36.9 (±1.88)	6.9	48.0 (±0.043)	5.3
xorshift avx2 4 interleaved	9.6 (±0.76)	26.7	14.5 (±0.014)	17.6
xorshift avx2 8 interleaved	10.0 (±1.07)	25.6	14.0 (±0.353)	18.3

```
; Load the data
vmovdqu ymm0, ymmword ptr [rsi]
vmovdqu ymm2, ymmword ptr [rsi + 32]
vmovdqu ymm4, ymmword ptr [rsi + 64]
vmovdqu ymm6, ymmword ptr [rsi + 96]
vmovdqu ymm8, ymmword ptr [rsi + 128]
vmovdqu ymm10, ymmword ptr [rsi + 160]
vmovdqu ymm12, ymmword ptr [rsi + 192]
vmovdqu ymm14, ymmword ptr [rsi + 224]
; << 13
vpsllq ymm1, ymm0, 13
vpsllq ymm3, ymm2, 13
vpsllq ymm5, ymm4, 13
vpsllq ymm7, ymm6, 13
vpsllq ymm9, ymm8, 13
vpsllq ymm11, ymm10, 13
vpsllq ymm13, ymm12, 13
vpsllq ymm15, ymm14, 13
; ^
vpxor ymm0, ymm0, ymm1
vpxor ymm2, ymm2, ymm3
vpxor ymm4, ymm4, ymm5
vpxor ymm6, ymm6, ymm7
vpxor ymm8, ymm9, ymm1
vpxor ymm10, ymm11, ymm3
vpxor ymm12, ymm13, ymm5
vpxor ymm14, ymm15, ymm7
; >> 7
vpsrlq ymm1, ymm0, 7
vpsrlq ymm3, ymm2, 7
vpsrlq ymm5, ymm4, 7
vpsrlq ymm7, ymm6, 7
vpsrlq ymm9, ymm8, 7
vpsrlq ymm11, ymm10, 7
vpsrlq ymm13, ymm12, 7
vpsrlq ymm15, ymm14, 7
```

Figure B.3: First part of the implementation of the 8 way interleaved AVX2 xorshift. It uses the vectorized xorshift.

```
; ^
vpxor ymm0, ymm0, ymm1
vpxor ymm2, ymm2, ymm3
vpxor ymm4, ymm4, ymm5
vpxor ymm6, ymm6, ymm7
vpxor ymm8, ymm9, ymm1
vpxor ymm10, ymm11, ymm3
vpxor ymm12, ymm13, ymm5
vpxor ymm14, ymm15, ymm7
; << 17
vpsllq ymm1, ymm0, 17
vpsllq ymm3, ymm2, 17
vpsllq ymm5, ymm4, 17
vpsllq ymm7, ymm6, 17
vpsllq ymm9, ymm8, 17
vpsllq ymm11, ymm10, 17
vpsllq ymm13, ymm12, 17
vpsllq ymm15, ymm14, 17
; ^
vpxor ymm0, ymm0, ymm1
vpxor ymm2, ymm2, ymm3
vpxor ymm4, ymm4, ymm5
vpxor ymm6, ymm6, ymm7
vpxor ymm8, ymm9, ymm1
vpxor ymm10, ymm11, ymm3
vpxor ymm12, ymm13, ymm5
vpxor ymm14, ymm15, ymm7
; Store the data
vmovdqu ymmword ptr [rdi], ymm0
vmovdqu ymmword ptr [rdi + 32], ymm2
vmovdqu ymmword ptr [rdi + 64], ymm4
vmovdqu ymmword ptr [rdi + 96], ymm6
vmovdqu ymmword ptr [rdi + 128], ymm8
vmovdqu ymmword ptr [rdi + 160], ymm10
vmovdqu ymmword ptr [rdi + 192], ymm12
vmovdqu ymmword ptr [rdi + 224], ymm14
```

Figure B.4: Second part of the implementation of the 8 way interleaved AVX2 xorshift. It uses the vectorized xorshift.

Appendix C

Alias method

The alias method [65] efficiently samples k integers from a discrete probability distribution. The algorithm is used since it requires $\mathcal{O}(n)$ steps in the pre-processing phase (when Vose's algorithm is used [126]) and $\mathcal{O}(1)$ steps for each point sampling. The algorithm is sketched in figure C.1, using a discrete probability distribution $[p_0, \ldots, p_{n-1}]$ where each p_i is the probability of sampling an integer $i \in [0, \ldots, n-1]$ ($n = 4$ in figure C.1). The heights of the bars correspond to $[p_0, \ldots, p_{n-1}]$ and their width is one, so that the histogram area is 1. Point sampling from the depicted distribution requires to draw a random value $0 \leq u \leq 1$ and find the segment (from p_0 to p_1, from p_1 to p_2, and so on) where u falls. This check could be performed by the following comparisons: if $u < p_0$ then pick 0; otherwise, if $p_0 \leq u < p_0 + p_1$ then sample 1; otherwise, if $p_0 + p_1 \leq u < p_0 + p_1 + p_2$ then sample 2; otherwise if $u \geq p_0 + p_1 + p_2$ then sample 3. This would be computationally expensive since each sampling would require, in the worst case, $n-1$ comparisons. To reduce the computational complexity, a more efficient algorithm could be designed that draws 2D points falling in the gray rectangle shown in figure C.1-b, where each bar represent the probability of sampling of one integer value according to the discrete probability distribution, or, in other words, each bar represents a sampling event. The algorithm needs only to check whether the drawn point falls in any of the bars. To this aim, an iterative process could be used that exploits only an indexed array with the discrete probabilities $P_{vec} = [p_0, \ldots, p_{n-1}]$ of each integer. Each iteration should draw two values: an index $0 \leq s \leq n$, that selects one of the elements of the array P_{vec}, and a random value $0 \leq u \leq 1$. The drawing continues until the drawn s and u are such that $u \leq P_{vec}[s]$, which leads to sampling s. In figure C.1-b the index $s = 1$ and the value (corresponding to the gray dot) $u \geq P_{vec}[1]$ correspond to a miss (the gray dot) that does not allow selecting 1 as the drawn value; conversely, the green dot corresponds to a drawn index $s = 3$ and a drawn $u \leq P_{vec}[3]$, which allows selecting the value 3 as the sampled value. However, as shown in figure C.1-b, usage the aforementioned method has the disadvantage of having an "empty space" that would cause repeated iterations, therefore increasing the computational time. To guarantee success at each draws, in [65] authors propose composing a rectangle where the probabilities ("rectangles" in the figure) are rearranged by splitting them, so as no empty spaces are left and each column contains at (figure C.1-c). In this way, if each draw of a 2D point would always fall into a decision region, therefore bringing to a decision. To this aim, the alias method applies a pre-processing phase, generally performed through Vose's algorithm [126] that, given the range $[0 - max]$ among which to sample, and the number n of integers to sample, computes two indexed vectors: the separating probability vector $P_{sep} = [p_0^{sep}, \ldots, p_{n-1}^{sep}]$, and the "other class" (also

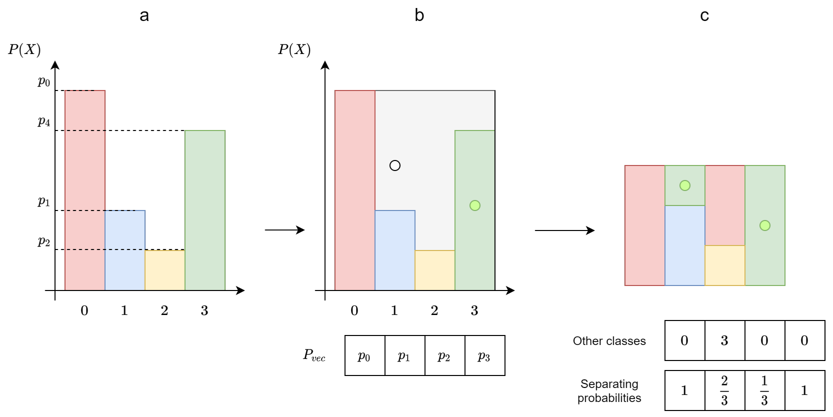


Figure C.1: **The Alias method**. Left: a discrete probability distribution. Center: a quasi-efficient sampling method containing empty spaces. Right: the alias method.

called alias) vector $OC = [oc_0, \dots, oc_{n-1}]$. The algorithm then draws a unique point $0 \leq x \leq 1$, from which it then computes an integer $s = \lfloor nx \rfloor$, uniformly distributed in $0, 2, \dots, n-1$, and a value $u = nx + 1 - s$ uniformly distributed on $[0, 1)$. If $u \leq P_{sep}[s]$ the alias method samples s, otherwise it samples $OC[s]$. Though efficient in the sampling process, when the range from which to sample becomes high, the pre-processing phase for computing the separating probabilities and the alias vector is impractical. Therefore, in *Ensmallen* we have implemented the SUSS algorithm, described in 2.3.2.

C.1 STRING and HPO datasets used for the edge-prediction experiments

HomoSapiens

The undirected graph HomoSapiens has 19.57K nodes and 252.98K edges. The graph contains 2.85K connected components (of which 2.75K are disconnected nodes), with the largest one containing 16.58K nodes and the smallest one containing a single node. The RAM requirements for the nodes and edges data structures are 2.67MB and 782.89KB respectively.

Degree centrality The minimum node degree is 0, the maximum node degree is 747, the mode degree is 0, the mean degree is 25.86 and the node degree median is 9. The nodes with the highest degree centrality are ENSP00000272317 (degree 747), ENSP00000269305 (degree 723), ENSP00000388107 (degree 669), ENSP00000441543 (degree 620) and ENSP00000362680 (degree 572).

Weights The minimum edge weight is 700, the maximum edge weight is 999 and the total edge weight is 437919420. The RAM requirement for the edge weights data structure is 2.02MB.

Topological Oddities A topological oddity is a set of nodes in the graph that *may be derived* by an error during the generation of the edge list of the graph and, depending on the task, could bias the results of topology-based models. In the following paragraph, we will describe the detected topological oddities.

Singleton nodes A singleton node is a node disconnected from all other nodes. We have detected 2.75K singleton nodes in the graph, involving a total of 2.75K nodes (14.07%). The detected singleton nodes are:

- ENSP00000005995
- ENSP00000006526
- ENSP00000006724
- ENSP00000007735
- ENSP00000010299
- ENSP00000039989
- ENSP00000064571
- ENSP00000084798
- ENSP00000086933
- ENSP00000158009
- ENSP00000159087
- ENSP00000161006
- ENSP00000162391
- ENSP00000164640
- ENSP00000190165

And other 2.74K singleton nodes.

Node tuples A node tuple is a connected component composed of two nodes. We have detected 77 node tuples in the graph, involving a total of 154 nodes (0.79%) and 77 edges (0.02%). The detected node tuples are:

- Node tuple containing the nodes ENSP00000484403 and ENSP00000484443.
- Node tuple containing the nodes ENSP00000480336 and ENSP00000485424.
- Node tuple containing the nodes ENSP00000454340 and ENSP00000485142.
- Node tuple containing the nodes ENSP00000448841 and ENSP00000481258.
- Node tuple containing the nodes ENSP00000436580 and ENSP00000436891.
- Node tuple containing the nodes ENSP00000436042 and ENSP00000436426.
- Node tuple containing the nodes ENSP00000435550 and ENSP00000441497.
- Node tuple containing the nodes ENSP00000430100 and ENSP00000454770.
- Node tuple containing the nodes ENSP00000429808 and ENSP00000433378.
- Node tuple containing the nodes ENSP00000419502 and ENSP00000483005.
- Node tuple containing the nodes ENSP00000406723 and ENSP00000480524.
- Node tuple containing the nodes ENSP00000399664 and ENSP00000472698.
- Node tuple containing the nodes ENSP00000391594 and ENSP00000460602.
- Node tuple containing the nodes ENSP00000388864 and ENSP00000396732.
- Node tuple containing the nodes ENSP00000384214 and ENSP00000397103.

And other 62 node tuples.

Isomorphic node groups Isomorphic groups are nodes with exactly the same neighbours and node types (if present in the graph). Nodes in such groups are topologically indistinguishable, that is swapping their ID would not change the graph topology. We have detected 21 isomorphic node groups in the graph, involving a total of 46 nodes (0.24%) and 558 edges (0.11%), with the largest one involving 5 nodes and 95 edges. The detected isomorphic node groups, sorted by decreasing size, are:

- Group with 5 nodes (degree 19): ENSP00000380488, ENSP00000423313, ENSP00000227756, ENSP00000344260 and ENSP00000297107.
- Group with 2 nodes (degree 29): ENSP00000391692 and ENSP00000391869.
- Group with 2 nodes (degree 21): ENSP00000436604 and ENSP00000444171.
- Group with 2 nodes (degree 18): ENSP00000384876 and ENSP00000373278.
- Group with 2 nodes (degree 18): ENSP00000347119 and ENSP00000354723.
- Group with 3 nodes (degree 11): ENSP00000369564, ENSP00000369566 and ENSP00000239347.
- Group with 2 nodes (degree 15): ENSP00000416741 and ENSP00000264363.
- Group with 2 nodes (degree 15): ENSP00000400157 and ENSP00000312700.
- Group with 2 nodes (degree 15): ENSP00000261196 and ENSP00000441269.
- Group with 2 nodes (degree 12): ENSP00000334681 and ENSP00000347810.
- Group with 2 nodes (degree 10): ENSP00000335281 and ENSP00000334330.
- Group with 2 nodes (degree 10): ENSP00000335307 and ENSP00000334364.
- Group with 2 nodes (degree 9): ENSP00000449334 and ENSP00000449223.
- Group with 2 nodes (degree 7): ENSP00000259205 and ENSP00000259211.
- Group with 2 nodes (degree 6): ENSP00000376955 and ENSP00000275884.

And other 6 isomorphic node groups.

Trees A tree is a connected component with n nodes and n-1 edges. We have detected 2 trees in the graph, involving a total of 14 nodes (0.07%) and 12 edges, with the largest one involving 9 nodes and 8 edges. The detected trees, sorted by decreasing size, are:

- Tree starting from the root node ENSP00000303028 (degree 2), and containing 9 nodes, with a maximal depth of 4, which are ENSP00000362386 (degree 3), ENSP00000385592, ENSP00000272438, ENSP00000364643 and ENSP00000384887 (degree 3).
- Tree starting from the root node ENSP00000355045 (degree 2), and containing 5 nodes, with a maximal depth of 2, which are ENSP00000379434, ENSP00000381857, ENSP00000352314 and ENSP00000473941.

Dendritic trees A dendritic tree is a tree-like structure starting from a root node that is part of another strongly connected component. We have detected 14 dendritic trees in the graph, involving a total of 61 nodes (0.31%) and 61 edges (0.01%), with the largest one involving 6 nodes and 6 edges. The detected dendritic trees, sorted by decreasing size, are:

- Dendritic tree starting from the root node ENSP00000254627 (degree 5), and containing 6 nodes, with a maximal depth of 3, which are ENSP00000407107 (degree 5), ENSP00000266022, ENSP00000299308, ENSP00000336693 and ENSP00000391404.

- Dendritic tree starting from the root node ENSP00000366542 (degree 5), and containing 6 nodes, with a maximal depth of 3, which are ENSP00000276127 (degree 3), ENSP00000281722, ENSP00000357889, ENSP00000362206 and ENSP00000306776.
- Dendritic tree starting from the root node ENSP00000310686 (degree 14), and containing 5 nodes, with a maximal depth of 3, which are ENSP00000482345, ENSP00000300896 (degree 4), ENSP00000478426, ENSP00000478683 and ENSP00000483965.
- Dendritic tree starting from the root node ENSP00000311336 (degree 5), and containing 5 nodes, with a maximal depth of 3, which are ENSP00000232892 (degree 3), ENSP00000353557, ENSP00000355156, ENSP00000362814 and ENSP00000348911.
- Dendritic tree starting from the root node ENSP00000446916 (degree 4), and containing 5 nodes, with a maximal depth of 2, which are ENSP00000373583 (degree 5), ENSP00000352299, ENSP00000357811, ENSP00000386118 and ENSP00000387298.
- Dendritic tree starting from the root node ENSP00000265322 (degree 43), and containing 4 nodes, with a maximal depth of 3, which are ENSP00000333183, ENSP00000467301 (degree 3), ENSP00000331435 and ENSP00000370801.

And other 8 dendritic trees.

Stars A star is a tree with a maximal depth of one, where nodes with maximal unique degree one are connected to a central root node with a high degree. We have detected 11 stars in the graph, involving a total of 35 nodes (0.18%) and 24 edges, with the largest one involving 4 nodes and 3 edges. The detected stars, sorted by decreasing size, are:

- Star starting from the root node ENSP00000334289 (degree 3), and containing 4 nodes, with a maximal depth of 1, which are ENSP00000324672, ENSP00000366223 and ENSP00000485629.
- Star starting from the root node ENSP00000345333 (degree 3), and containing 4 nodes, with a maximal depth of 1, which are ENSP00000328245, ENSP00000339314 and ENSP00000483077.
- Star starting from the root node ENSP00000281523 (degree 2), and containing 3 nodes, with a maximal depth of 1, which are ENSP00000297063 and ENSP00000418575.
- Star starting from the root node ENSP00000295012 (degree 2), and containing 3 nodes, with a maximal depth of 1, which are ENSP00000469705 and ENSP00000484537.
- Star starting from the root node ENSP00000319590 (degree 2), and containing 3 nodes, with a maximal depth of 1, which are ENSP00000230301 and ENSP00000354590.
- Star starting from the root node ENSP00000344129 (degree 2), and containing 3 nodes, with a maximal depth of 1, which are ENSP00000380071 and ENSP00000469417.

And other 5 stars.

Dendritic stars A dendritic star is a dendritic tree with a maximal depth of one, where nodes with maximal unique degree one are connected to a central root node with high degree and inside a strongly connected component. We have detected 90 dendritic stars in the graph, involving a total of 220 nodes (1.12%) and 220 edges (0.04%), with the largest one involving 19 nodes and 19 edges. The detected dendritic stars, sorted by decreasing size, are:

- Dendritic star starting from the root node ENSP00000334051 (degree 82), and containing 19 nodes, with a maximal depth of 1, which are ENSP00000284287, ENSP00000307726, ENSP00000311605, ENSP00000318997 and ENSP00000319071.
- Dendritic star starting from the root node ENSP00000432561 (degree 14), and containing 7 nodes, with a maximal depth of 1, which are ENSP00000254166, ENSP00000303533, ENSP00000344162, ENSP00000379464 and ENSP00000385163.
- Dendritic star starting from the root node ENSP00000319673 (degree 16), and containing 6 nodes, with a maximal depth of 1, which are ENSP00000360360, ENSP00000392283, ENSP00000393315, ENSP00000399711 and ENSP00000410400.
- Dendritic star starting from the root node ENSP00000376345 (degree 380), and containing 4 nodes, with a maximal depth of 1, which are ENSP00000320038, ENSP00000356093, ENSP00000384593 and ENSP00000470441.
- Dendritic star starting from the root node ENSP00000267017 (degree 20), and containing 4 nodes, with a maximal depth of 1, which are ENSP00000309782, ENSP00000314042, ENSP00000330612 and ENSP00000366331.
- Dendritic star starting from the root node ENSP00000246801 (degree 7), and containing 3 nodes, with a maximal depth of 1, which are ENSP00000362634, ENSP00000375081 and ENSP00000382544.

And other 84 dendritic stars.

Dendritic tendril stars A dendritic tendril star is a dendritic tree with a depth greater than one, where the arms of the star are tendrils. We have detected 13 dendritic tendril stars in the graph, involving a total of 48 nodes (0.25%) and 48 edges, with the largest one involving 9 nodes and 9 edges. The detected dendritic tendril stars, sorted by decreasing size, are:

- Dendritic tendril star starting from the root node ENSP00000371594 (degree 68), and containing 9 nodes, with a maximal depth of 2, which are ENSP00000302199, ENSP00000312403, ENSP00000323853, ENSP00000329982 and ENSP00000330904.
- Dendritic tendril star starting from the root node ENSP00000310275 (degree 58), and containing 5 nodes, with a maximal depth of 4, which are ENSP00000356121, ENSP00000360320, ENSP00000351727, ENSP00000355840 and ENSP00000340887.
- Dendritic tendril star starting from the root node ENSP00000363382 (degree 4), and containing 4 nodes, with a maximal depth of 3, which are ENSP00000332663, ENSP00000363894, ENSP00000448580 and ENSP00000363899.
- Dendritic tendril star starting from the root node ENSP00000216862 (degree 24), and containing 3 nodes, with a maximal depth of 2, which are ENSP00000309649, ENSP00000318912 and ENSP00000242315.
- Dendritic tendril star starting from the root node ENSP00000276590 (degree 10), and containing 3 nodes, with a maximal depth of 2, which are ENSP00000262288, ENSP00000271375 and ENSP00000348033.

- Dendritic tendril star starting from the root node ENSP00000283946 (degree 10), and containing 3 nodes, with a maximal depth of 2, which are ENSP00000359819, ENSP00000377577 and ENSP00000229002.

And other 7 dendritic tendril stars.

Tendrils A tendril is a path starting from a node of degree one, connected to a strongly connected component. We have detected 995 tendrils in the graph, involving a total of 1.08K nodes (5.52%) and 1.08K edges (0.21%), with the largest one involving 4 nodes and 4 edges. The detected tendrils, sorted by decreasing size, are:

- Tendril starting from the root node ENSP00000374014 (degree 7), and containing 4 nodes, with a maximal depth of 4, which are ENSP00000291481, ENSP00000317000, ENSP00000394178 and ENSP00000296862.
- Tendril starting from the root node ENSP00000371512 (degree 13), and containing 3 nodes, with a maximal depth of 3, which are ENSP00000310182, ENSP00000344331 and ENSP00000386389.
- Tendril starting from the root node ENSP00000264554 (degree 44), and containing 3 nodes, with a maximal depth of 3, which are ENSP00000344465, ENSP00000336661 and ENSP00000430271.
- Tendril starting from the root node ENSP00000302578 (degree 4), and containing 3 nodes, with a maximal depth of 3, which are ENSP00000264735, ENSP00000295092 and ENSP00000370724.
- Tendril starting from the root node ENSP00000302843 (degree 6), and containing 3 nodes, with a maximal depth of 3, which are ENSP00000414920, ENSP00000206020 and ENSP00000317289.
- Tendril starting from the root node ENSP00000307636 (degree 6), and containing 3 nodes, with a maximal depth of 3, which are ENSP00000386563, ENSP00000367792 and ENSP00000411253.

And other 989 tendrils.

MusMusculus

The undirected graph MusMusculus has 22.05K nodes and 233.76K edges. The graph contains 5.57K connected components (of which 5.42K are disconnected nodes), with the largest one containing 16.18K nodes and the smallest one containing a single node. The RAM requirements for the nodes and edges data structures are 3.28MB and 739.96KB respectively.

Degree centrality The minimum node degree is 0, the maximum node degree is 684, the mode degree is 0, the mean degree is 21.21 and the node degree median is 6. The nodes with the highest degree centrality are ENSMUSP00000099909 (degree 684), ENSMUSP00000104298 (degree 649), ENSMUSP00000007130 (degree 585), ENSMUSP00000029175 (degree 538) and ENSMUSP00000001780 (degree 517).

Weights The minimum edge weight is 700, the maximum edge weight is 999 and the total edge weight is 394822140. The RAM requirement for the edge weights data structure is 1.87MB.

Topological Oddities A topological oddity is a set of nodes in the graph that *may be derived* by an error during the generation of the edge list of the graph and, depending on the task, could bias the results of topology-based models. In the following paragraph, we will describe the detected topological oddities.

Singleton nodes A singleton node is a node disconnected from all other nodes. We have detected 5.42K singleton nodes in the graph, involving a total of 5.42K nodes (24.56%). The detected singleton nodes are:

- ENSMUSP00000000208
- ENSMUSP00000000266
- ENSMUSP00000000327
- ENSMUSP00000000449
- ENSMUSP00000000619
- ENSMUSP00000000717
- ENSMUSP00000000755
- ENSMUSP00000001009
- ENSMUSP00000001122
- ENSMUSP00000001172
- ENSMUSP00000001185
- ENSMUSP00000001242
- ENSMUSP00000001456
- ENSMUSP00000001544
- ENSMUSP00000001626

And other 5.40K singleton nodes.

Node tuples A node tuple is a connected component composed of two nodes. We have detected 98 node tuples in the graph, involving a total of 196 nodes (0.89%) and 98 edges (0.02%). The detected node tuples are:

- Node tuple containing the nodes ENSMUSP00000139506 and ENSMUSP00000140944.
- Node tuple containing the nodes ENSMUSP00000139148 and ENSMUSP00000140416.
- Node tuple containing the nodes ENSMUSP00000137187 and ENSMUSP00000137299.
- Node tuple containing the nodes ENSMUSP00000135899 and ENSMUSP00000136712.
- Node tuple containing the nodes ENSMUSP00000126635 and ENSMUSP00000132971.
- Node tuple containing the nodes ENSMUSP00000125936 and ENSMUSP00000136153.
- Node tuple containing the nodes ENSMUSP00000121288 and ENSMUSP00000129762.
- Node tuple containing the nodes ENSMUSP00000113317 and ENSMUSP00000128826.
- Node tuple containing the nodes ENSMUSP00000109660 and ENSMUSP00000111141.
- Node tuple containing the nodes ENSMUSP00000102037 and ENSMUSP00000133699.
- Node tuple containing the nodes ENSMUSP00000101431 and ENSMUSP00000111088.
- Node tuple containing the nodes ENSMUSP00000100068 and ENSMUSP00000107129.
- Node tuple containing the nodes ENSMUSP00000097268 and ENSMUSP00000100772.
- Node tuple containing the nodes ENSMUSP00000097041 and ENSMUSP00000112932.
- Node tuple containing the nodes ENSMUSP00000096773 and ENSMUSP00000114092.

And other 83 node tuples.

Isomorphic node groups Isomorphic groups are nodes with exactly the same neighbours and node types (if present in the graph). Nodes in such groups are topologically indistinguishable, that is swapping their ID would not change the graph topology. We have detected 66 isomorphic node groups in the graph, involving a total of 179 nodes (0.81%) and 6.42K edges (1.37%), with the largest one involving 11 nodes and 590 edges. The detected isomorphic node groups, sorted by decreasing size, are:

- Group with 5 nodes (degree 118): ENSMUSP00000132343, ENSMUSP00000096592, ENSMUSP00000128232, ENSMUSP00000100655 and ENSMUSP00000129788.
- Group with 5 nodes (degree 103): ENSMUSP00000100589, ENSMUSP00000071496, ENSMUSP00000138961, ENSMUSP00000087632 and ENSMUSP00000092345.
- Group with 2 nodes (degree 219): ENSMUSP00000126892 and ENSMUSP00000128443.
- Group with 2 nodes (degree 207): ENSMUSP00000138342 and ENSMUSP00000127859.
- Group with 3 nodes (degree 137): ENSMUSP00000126334, ENSMUSP00000110610 and ENSMUSP00000132241.
- Group with 2 nodes (degree 191): ENSMUSP00000128489 and ENSMUSP00000131471.
- Group with 2 nodes (degree 186): ENSMUSP00000137545 and ENSMUSP00000100516.
- Group with 2 nodes (degree 172): ENSMUSP00000100805 and ENSMUSP00000100528.
- Group with 2 nodes (degree 160): ENSMUSP00000129583 and ENSMUSP00000129695.
- Group with 2 nodes (degree 147): ENSMUSP00000136791 and ENSMUSP00000080543.
- Group with 2 nodes (degree 138): ENSMUSP00000093124 and ENSMUSP00000136275.
- Group with 2 nodes (degree 137): ENSMUSP00000082867 and ENSMUSP00000078747.
- Group with 11 nodes (degree 22): ENSMUSP00000025322, ENSMUSP00000109371, ENSMUSP00000040435, ENSMUSP00000058686, ENSMUSP00000084411 and other 6.
- Group with 5 nodes (degree 40): ENSMUSP00000100769, ENSMUSP00000136763, ENSMUSP00000088719, ENSMUSP00000137043 and ENSMUSP00000126567.
- Group with 2 nodes (degree 82): ENSMUSP00000138211 and ENSMUSP00000138181.

And other 51 isomorphic node groups.

Dendritic trees A dendritic tree is a tree-like structure starting from a root node that is part of another strongly connected component. We have detected 19 dendritic trees in the graph, involving a total of 81 nodes (0.37%) and 81 edges (0.02%), with the largest one involving 9 nodes and 9 edges. The detected dendritic trees, sorted by decreasing size, are:

- Dendritic tree starting from the root node ENSMUSP00000132092 (degree 4), and containing 9 nodes, with a maximal depth of 7, which are ENSMUSP00000092725, ENSMUSP00000094845, ENSMUSP00000049864 (degree 3), ENSMUSP00000037596 and ENSMUSP00000054292.
- Dendritic tree starting from the root node ENSMUSP00000038744 (degree 202), and containing 6 nodes, with a maximal depth of 2, which are ENSMUSP00000038638,

ENSMUSP00000055692, ENSMUSP00000064785, ENSMUSP00000106128 and ENSMUSP00000063248.

- Dendritic tree starting from the root node ENSMUSP00000044998 (degree 10), and containing 6 nodes, with a maximal depth of 5, which are ENSMUSP00000053849, ENSMUSP00000039821 (degree 3), ENSMUSP00000036503, ENSMUSP00000050902 and ENSMUSP00000043513.
- Dendritic tree starting from the root node ENSMUSP00000086311 (degree 3), and containing 6 nodes, with a maximal depth of 4, which are ENSMUSP00000085528, ENSMUSP00000088822 (degree 3), ENSMUSP00000028283, ENSMUSP00000085531 and ENSMUSP00000109664.
- Dendritic tree starting from the root node ENSMUSP00000014830 (degree 11), and containing 5 nodes, with a maximal depth of 2, which are ENSMUSP00000057433 (degree 5), ENSMUSP00000032520, ENSMUSP00000032663, ENSMUSP00000047914 and ENSMUSP00000092344.
- Dendritic tree starting from the root node ENSMUSP00000063839 (degree 62), and containing 5 nodes, with a maximal depth of 3, which are ENSMUSP00000029800 (degree 3), ENSMUSP00000035263, ENSMUSP00000062837, ENSMUSP00000096097 and ENSMUSP00000096443.

And other 13 dendritic trees.

Stars A star is a tree with a maximal depth of one, where nodes with maximal unique degree one are connected to a central root node with a high degree. We have detected 26 stars in the graph, involving a total of 82 nodes (0.37%) and 56 edges (0.01%), with the largest one involving 5 nodes and 4 edges. The detected stars, sorted by decreasing size, are:

- Star starting from the root node ENSMUSP00000062719 (degree 4), and containing 5 nodes, with a maximal depth of 1, which are ENSMUSP00000076107, ENSMUSP00000078756, ENSMUSP00000079039 and ENSMUSP00000095815.
- Star starting from the root node ENSMUSP00000110147 (degree 3), and containing 4 nodes, with a maximal depth of 1, which are ENSMUSP00000063474, ENSMUSP00000070718 and ENSMUSP00000132644.
- Star starting from the root node ENSMUSP00000076458 (degree 3), and containing 4 nodes, with a maximal depth of 1, which are ENSMUSP00000033575, ENSMUSP00000076472 and ENSMUSP00000083892.
- Star starting from the root node ENSMUSP00000038678 (degree 2), and containing 3 nodes, with a maximal depth of 1, which are ENSMUSP00000077650 and ENSMUSP00000110164.
- Star starting from the root node ENSMUSP00000113392 (degree 2), and containing 3 nodes, with a maximal depth of 1, which are ENSMUSP00000097485 and ENSMUSP00000107204.
- Star starting from the root node ENSMUSP00000096752 (degree 2), and containing 3 nodes, with a maximal depth of 1, which are ENSMUSP00000135685 and ENSMUSP00000137012.

And other 20 stars.

Dendritic stars A dendritic star is a dendritic tree with a maximal depth of one, where nodes with maximal unique degree one are connected to a central root node with high degree and inside a strongly connected component. We have detected 106 dendritic stars in the graph, involving a total of 352 nodes (1.60%) and 352 edges (0.08%), with the largest one involving 95 nodes and 95 edges. The detected dendritic stars, sorted by decreasing size, are:

- Dendritic star starting from the root node ENSMUSP00000025402 (degree 190), and containing 95 nodes, with a maximal depth of 1, which are ENSMUSP00000038992, ENSMUSP00000041524, ENSMUSP00000050544, ENSMUSP00000050833 and ENSMUSP00000051280.
- Dendritic star starting from the root node ENSMUSP00000033300 (degree 20), and containing 16 nodes, with a maximal depth of 1, which are ENSMUSP00000071783, ENSMUSP00000074745, ENSMUSP00000076041, ENSMUSP00000076665 and ENSMUSP00000079376.
- Dendritic star starting from the root node ENSMUSP00000041483 (degree 43), and containing 6 nodes, with a maximal depth of 1, which are ENSMUSP00000015585, ENSMUSP00000015587, ENSMUSP00000015594, ENSMUSP00000022757 and ENSMUSP00000080742.
- Dendritic star starting from the root node ENSMUSP00000131269 (degree 167), and containing 6 nodes, with a maximal depth of 1, which are ENSMUSP00000052312, ENSMUSP00000071824, ENSMUSP00000077990, ENSMUSP00000079893 and ENSMUSP00000087276.
- Dendritic star starting from the root node ENSMUSP00000006235 (degree 37), and containing 5 nodes, with a maximal depth of 1, which are ENSMUSP00000023619, ENSMUSP00000078181, ENSMUSP00000087054, ENSMUSP00000093794 and ENSMUSP00000093795.
- Dendritic star starting from the root node ENSMUSP00000074436 (degree 33), and containing 4 nodes, with a maximal depth of 1, which are ENSMUSP00000082127, ENSMUSP00000105584, ENSMUSP00000113945 and ENSMUSP00000140024.

And other 100 dendritic stars.

Dendritic tendril stars A dendritic tendril star is a dendritic tree with a depth greater than one, where the arms of the star are tendrils. We have detected 24 dendritic tendril stars in the graph, involving a total of 93 nodes (0.42%) and 93 edges (0.02%), with the largest one involving 15 nodes and 15 edges. The detected dendritic tendril stars, sorted by decreasing size, are:

- Dendritic tendril star starting from the root node ENSMUSP00000122219 (degree 36), and containing 15 nodes, with a maximal depth of 2, which are ENSMUSP00000021776, ENSMUSP00000021778, ENSMUSP00000021779, ENSMUSP00000023602 and ENSMUSP00000046522.
- Dendritic tendril star starting from the root node ENSMUSP00000031011 (degree 134), and containing 7 nodes, with a maximal depth of 2, which are ENSMUSP00000032185, ENSMUSP00000040429, ENSMUSP00000059419, ENSMUSP00000094334 and ENSMUSP00000130758.

- Dendritic tendril star starting from the root node ENSMUSP00000130738 (degree 144), and containing 5 nodes, with a maximal depth of 2, which are ENSMUSP00000002880, ENSMUSP00000086835, ENSMUSP00000094097, ENSMUSP00000114489 and ENSMUSP00000033210.
- Dendritic tendril star starting from the root node ENSMUSP00000049228 (degree 128), and containing 4 nodes, with a maximal depth of 2, which are ENSMUSP00000030878, ENSMUSP00000036916, ENSMUSP00000076935 and ENSMUSP00000033414.
- Dendritic tendril star starting from the root node ENSMUSP00000104298 (degree 649), and containing 4 nodes, with a maximal depth of 2, which are ENSMUSP00000031434, ENSMUSP00000118809, ENSMUSP00000130176 and ENSMUSP00000071896.
- Dendritic tendril star starting from the root node ENSMUSP00000051068 (degree 22), and containing 4 nodes, with a maximal depth of 3, which are ENSMUSP00000044228, ENSMUSP00000110316, ENSMUSP00000034413 and ENSMUSP00000136717.

And other 18 dendritic tendril stars.

Free-floating chains A free-floating chain is a tree with maximal degree two. We have detected 2 free-floating chains in the graph, involving a total of 9 nodes (0.04%) and 7 edges, with the largest one involving 5 nodes and 4 edges. The detected free-floating chains, sorted by decreasing size, are:

- Free-floating chain starting from the root node ENSMUSP00000069019 (degree 3), and containing 5 nodes, with a maximal depth of 2, which are ENSMUSP00000090857, ENSMUSP00000101667, ENSMUSP00000117294 and ENSMUSP00000033804.
- Free-floating chain starting from the root node ENSMUSP00000104573 (degree 2), and containing 4 nodes, with a maximal depth of 2, which are ENSMUSP00000136794, ENSMUSP00000137575 and ENSMUSP00000104609.

Tendrils A tendril is a path starting from a node of degree one, connected to a strongly connected component. We have detected 1.05K tendrils in the graph, involving a total of 1.16K nodes (5.27%) and 1.16K edges (0.25%), with the largest one involving 4 nodes and 4 edges. The detected tendrils, sorted by decreasing size, are:

- Tendril starting from the root node ENSMUSP00000030446 (degree 14), and containing 4 nodes, with a maximal depth of 4, which are ENSMUSP00000098200, ENSMUSP00000087257, ENSMUSP00000020926 and ENSMUSP00000065613.
- Tendril starting from the root node ENSMUSP00000061753 (degree 4), and containing 3 nodes, with a maximal depth of 3, which are ENSMUSP00000101679, ENSMUSP00000120070 and ENSMUSP00000086041.
- Tendril starting from the root node ENSMUSP00000023673 (degree 70), and containing 3 nodes, with a maximal depth of 3, which are ENSMUSP00000088678, ENSMUSP00000078374 and ENSMUSP00000068904.
- Tendril starting from the root node ENSMUSP00000026986 (degree 4), and containing 3 nodes, with a maximal depth of 3, which are ENSMUSP00000001080, ENSMUSP00000058119 and ENSMUSP00000104752.

- Tendril starting from the root node ENSMUSP00000131648 (degree 102), and containing 3 nodes, with a maximal depth of 3, which are ENSMUSP00000058810, ENSMUSP00000090326 and ENSMUSP00000067699.
- Tendril starting from the root node ENSMUSP00000011526 (degree 8), and containing 3 nodes, with a maximal depth of 3, which are ENSMUSP00000029325, ENSMUSP00000029326 and ENSMUSP00000129444.

And other 1.04K tendrils.

Homo Phenotype Ontology

The undirected multigraph HP has 43.47K homogeneous nodes and 88.96K heterogeneous edges. The graph contains 551 connected components (of which 543 are disconnected nodes), with the largest one containing 42.83K nodes and the smallest one containing a single node. The RAM requirements for the nodes and edges data structures are 5.45MB and 359.04KB respectively.

Degree centrality The minimum node degree is 0, the maximum node degree is 1.96K, the mode degree is 1, the mean degree is 4.09 and the node degree median is 2. The nodes with the highest degree centrality are GO:0065007 (degree 1.96K and node type biolink:NamedThing), UBERON_CORE:pheno_slim (degree 1.72K and node type biolink:NamedThing), OBO:chebi#3_STAR (degree 1.52K and node type biolink:NamedThing), UBERON_CORE:uberon_slim (degree 900 and node type biolink:NamedThing) and OBO:hp#hposlim_core (degree 840 and node type biolink:NamedThing).

Node types The graph has a single node type, which is biolink:NamedThing. The RAM requirement for the node types data structure is 2.26MB.

Homogeneous node types Homogeneous node types are node types that are assigned to all the nodes in the graph, making the node type relatively meaningless, as it adds no more information than the fact that the node is in the graph. The graph contains a homogeneous node type, which is biolink:NamedThing.

Edge types The graph has 46 edge types, of which the 10 most common are biolink:subclass_of (95.86K edges, 53.90%), biolink:related_to (24.77K edges, 13.93%), biolink:has_part (14.28K edges, 8.03%), biolink:part_of (13.77K edges, 7.74%), biolink:close_match (11.76K edges, 6.61%), biolink:develops_from (2.59K edges, 1.46%), biolink:has_attribute (1.60K edges, 0.90%), biolink:regulates (1.48K edges, 0.83%), biolink:positively_regulates (1.25K edges, 0.71%) and biolink:negatively_regulates (1.25K edges, 0.70%). The RAM requirement for the edge types data structure is 1.43MB.

Topological Oddities A topological oddity is a set of nodes in the graph that *may be derived* by an error during the generation of the edge list of the graph and, depending on the task, could bias the results of topology-based models. In the following paragraph, we will describe the detected topological oddities.

Singleton nodes A singleton node is a node disconnected from all other nodes. We have detected 539 singleton nodes in the graph, involving a total of 539 nodes (1.24%). The detected singleton nodes are:

- is_substituent_group_from (node type biolink:NamedThing)
- BFO:0000056 (node type biolink:NamedThing)
- OBO:cl#lacks_part (node type biolink:NamedThing)
- OBO:pr#has_gene_template (node type biolink:NamedThing)
- GOREL:0002003 (node type biolink:NamedThing)
- RO:0015011 (node type biolink:NamedThing)
- UBERON_CORE:channels_from (node type biolink:NamedThing)
- UBERON_CORE:synapsed_by (node type biolink:NamedThing)
- OBO:nbo#has_participant (node type biolink:NamedThing)
- UBERON_CORE:filtered_through (node type biolink:NamedThing)
- UBERON_CORE:site_of (node type biolink:NamedThing)
- OBO:chebi#is_tautomer_of (node type biolink:NamedThing)
- RO:0015012 (node type biolink:NamedThing)
- UBERON_CORE:indirectly_supplies (node type biolink:NamedThing)
- OBO:nbo#has-input (node type biolink:NamedThing)

And other 524 singleton nodes.

Singleton nodes with self-loops A singleton node with self-loops is a node disconnected from all other nodes except itself. We have detected 4 singleton nodes with self-loops in the graph, involving a total of 4 nodes and 4 edges. The detected singleton nodes with self-loops are:

- dc:description (node type biolink:NamedThing)
- dcterms-license (node type biolink:NamedThing)
- dc:title (node type biolink:NamedThing)
- dc:contributor (node type biolink:NamedThing)

Node tuples A node tuple is a connected component composed of two nodes. We have detected 2 node tuples in the graph, involving a total of 4 nodes and 2 edges. The detected node tuples are:

- Node tuple containing the nodes BSPO:0000098 (node type biolink:NamedThing) and ventral_to (node type biolink:NamedThing).
- Node tuple containing the nodes OBO:chebi#is_conjugate_acid_of (node type biolink:NamedThing) and OBO:chebi#is_conjugate_base_of (node type biolink:NamedThing

Isomorphic node groups Isomorphic groups are nodes with exactly the same neighbours and node types (if present in the graph). Nodes in such groups are topologically indistinguishable, that is swapping their ID would not change the graph topology. We have detected 4 isomorphic node groups in the graph, involving a total of 9 nodes (0.02%) and 50 edges (0.03%), with the largest one involving 3 nodes and 18 edges. The detected isomorphic node groups, sorted by decreasing size, are:

- Group with 3 nodes (degree 6 and node type biolink:NamedThing): RO:0002296, RO:0002298 and RO:0002299.

- Group with 2 nodes (degree 6 and node type biolink:NamedThing): UBERON:0013773 and UBERON:0013772.
- Group with 2 nodes (degree 5 and node type biolink:NamedThing): RO:0002232 and RO:0002231.
- Group with 2 nodes (degree 5 and node type biolink:NamedThing): NCBITaxon:3176 and NCBITaxon:3378.

Dendritic trees A dendritic tree is a tree-like structure starting from a root node that is part of another strongly connected component. We have detected 721 dendritic trees in the graph, involving a total of 9.92K nodes (22.81%) and 9.92K edges (5.58%), with the largest one involving 249 nodes and 249 edges. The detected dendritic trees, sorted by decreasing size, are:

- Dendritic tree starting from the root node HP:0002960 (degree 9), and containing 249 nodes, with a maximal depth of 5, which are HP:0030057 (degree 164), HP:0002725, Nbf08f17380a543868ede822e13da03e7, HP:0032265 and HP:5000021 (degree 3). Its nodes have a single node type, which is biolink:NamedThing. Its edges have 2 edge types, which are biolink:subclass_of (493 edges, 99.80%) and biolink:has_part.
- Dendritic tree starting from the root node HP:0033354 (degree 45), and containing 140 nodes, with a maximal depth of 4, which are HP:0011279 (degree 4), HP:0033187, HP:0012404 (degree 4), HP:0003541 (degree 3) and HP:0011814. Its nodes have a single node type, which is biolink:NamedThing. Its edges have 2 edge types, which are biolink:subclass_of (157 edges, 64.08%) and biolink:has_part (88 edges, 35.92%).
- Dendritic tree starting from the root node HP:0010876 (degree 55), and containing 111 nodes, with a maximal depth of 5, which are HP:0031222, HP:0032463, HP:0012509, HP:0002152 and HP:0025201 (degree 3). Its nodes have a single node type, which is biolink:NamedThing. Its edges have 2 edge types, which are biolink:subclass_of (149 edges, 84.66%) and biolink:has_part (27 edges, 15.34%).
- Dendritic tree starting from the root node HP:0004303 (degree 28), and containing 103 nodes, with a maximal depth of 3, which are HP:0030089 (degree 19), HP:0012269 (degree 4), HP:0003791, HP:0030230 and HP:0100298. Its nodes have a single node type, which is biolink:NamedThing. Its edges have 2 edge types, which are biolink:subclass_of (157 edges, 87.22%) and biolink:has_part (23 edges, 12.78%).
- Dendritic tree starting from the root node HP:0000708 (degree 46), and containing 98 nodes, with a maximal depth of 4, which are HP:4000009, HP:0033051 (degree 6), HP:0033676 (degree 3), HP:0030858 (degree 3) and HP:0000711 (degree 3). Its nodes have a single node type, which is biolink:NamedThing. Its edges have 2 edge types, which are biolink:subclass_of (142 edges, 85.03%) and biolink:has_part (25 edges, 14.97%).
- Dendritic tree starting from the root node HP:0012379 (degree 52), and containing 96 nodes, with a maximal depth of 4, which are HP:0000816 (degree 5), HP:0033168, HP:0032459 (degree 3), HP:0031821 (degree 3) and HP:0034202 (degree 3). Its nodes have a single node type, which is biolink:NamedThing. Its

edges have 2 edge types, which are biolink:subclass_of (92 edges, 62.16%) and biolink:has_part (56 edges, 37.84%).

And other 715 dendritic trees.

Stars A star is a tree with a maximal depth of one, where nodes with maximal unique degree one are connected to a central root node with a high degree. We have detected 2 stars in the graph, involving a total of 9 nodes (0.02%) and 7 edges, with the largest one involving 6 nodes and 5 edges. The detected stars, sorted by decreasing size, are:

- Star starting from the root node OIO:SynonymTypeProperty (degree 5), and containing 6 nodes, with a maximal depth of 1, which are OBO:hp#layperson, OBO:hp#uk_spelling, OBO:hp#obsolete_synonym, OBO:hp#abbreviation and OBO:hp#plural_form. Its nodes have a single node type, which is biolink:NamedThing. Its edges have a single edge type, which is biolink:subclass_of.
- Star starting from the root node CHEBI:21241 (degree 2), and containing 3 nodes, with a maximal depth of 1, which are IAO:0000227 and CHEBI:176783. Its nodes have a single node type, which is biolink:NamedThing. Its edges have a single edge type, which is biolink:related_to.

Dendritic stars A dendritic star is a dendritic tree with a maximal depth of one, where nodes with maximal unique degree one are connected to a central root node with high degree and inside a strongly connected component. We have detected 2.47K dendritic stars in the graph, involving a total of 6.87K nodes (15.81%) and 6.87K edges (3.86%), with the largest one involving 16 nodes and 16 edges. The detected dendritic stars, sorted by decreasing size, are:

- Dendritic star starting from the root node MAXO:0000072 (degree 28), and containing 16 nodes, with a maximal depth of 1, which are MAXO:0010202, MAXO:0010211, MAXO:0010222, MAXO:0010218 and MAXO:0010219. Its nodes have a single node type, which is biolink:NamedThing. Its edges have a single edge type, which is biolink:subclass_of.
- Dendritic star starting from the root node BFO:0000050 (degree 40), and containing 13 nodes, with a maximal depth of 1, which are BSPO:0015102, BSPO:0001115, BSPO:0005001, BSPO:0000122 and BSPO:0015101. Its nodes have a single node type, which is biolink:NamedThing. Its edges have 3 edge types, which are biolink:subclass_of (10 edges, 76.92%), rdfs:seeAlso (2 edges, 15.38%) and biolink:related_to.
- Dendritic star starting from the root node UBERON:0002149 (degree 24), and containing 12 nodes, with a maximal depth of 1, which are http:&#x2f;&#x2f;braininfo.rprc.was http:&#x2f;&#x2f;www.snomedbrowser.com&#x2f;Codes&#x2f;Details&#x2f;369028007, UBERON:0003011, UBERON:0002143 and UBERON:0002963. Its nodes have a single node type, which is biolink:NamedThing. Its edges have 2 edge types, which are biolink:related_to (8 edges, 66.67%) and biolink:close_match (4 edges, 33.33%).
- Dendritic star starting from the root node HP:0003540 (degree 13), and containing 11 nodes, with a maximal depth of 1, which are HP:0031128, HP:0011894, N709ece82e4b04f06a9daa5b99744af65, HP:0008320 and HP:0031129. Its nodes

have a single node type, which is biolink:NamedThing. Its edges have 2 edge types, which are biolink:subclass_of (10 edges, 90.91%) and biolink:has_part.

- Dendritic star starting from the root node HP:0000972 (degree 14), and containing 11 nodes, with a maximal depth of 1, which are HP:0007548, HP:0007497, HP:0000982, HP:0007553 and HP:0007613. Its nodes have a single node type, which is biolink:NamedThing. Its edges have a single edge type, which is biolink:subclass_of.

- Dendritic star starting from the root node UBERON:0002871 (degree 21), and containing 11 nodes, with a maximal depth of 1, which are UBERON:0002155, UBERON:0002559, UBERON:0002127, http:&#x2f;&#x2f;www.snomedbrowser.com&#x2f;Code and UBERON:0002154. Its nodes have a single node type, which is biolink:NamedThing. Its edges have 2 edge types, which are biolink:related_to (8 edges, 72.73%) and biolink:close_match (3 edges, 27.27%).

And other 2.47K dendritic stars.

Dendritic tendril stars A dendritic tendril star is a dendritic tree with a depth greater than one, where the arms of the star are tendrils. We have detected 391 dendritic tendril stars in the graph, involving a total of 1.60K nodes (3.69%) and 1.60K edges (0.90%), with the largest one involving 17 nodes and 17 edges. The detected dendritic tendril stars, sorted by decreasing size, are:

- Dendritic tendril star starting from the root node HP:0001878 (degree 18), and containing 17 nodes, with a maximal depth of 2, which are HP:0004863, HP:0005511, HP:0004802, HP:0001930 and HP:0005524. Its nodes have a single node type, which is biolink:NamedThing. Its edges have 2 edge types, which are biolink:subclass_of (15 edges, 83.33%) and biolink:has_part (3 edges, 16.67%).

- Dendritic tendril star starting from the root node MAXO:0010365 (degree 14), and containing 13 nodes, with a maximal depth of 2, which are MAXO:0010359, MAXO:0010361, MAXO:0010351, MAXO:0010355 and MAXO:0010357. Its nodes have a single node type, which is biolink:NamedThing. Its edges have a single edge type, which is biolink:subclass_of.

- Dendritic tendril star starting from the root node HP:0031331 (degree 12), and containing 11 nodes, with a maximal depth of 2, which are HP:0033997, HP:0031319, HP:0031332, HP:0031333 and HP:0031339. Its nodes have a single node type, which is biolink:NamedThing. Its edges have a single edge type, which is biolink:subclass_of.

- Dendritic tendril star starting from the root node HP:0000962 (degree 23), and containing 11 nodes, with a maximal depth of 2, which are HP:0005595, HP:0007501, HP:0025080, HP:0007490 and HP:0031288. Its nodes have a single node type, which is biolink:NamedThing. Its edges have 2 edge types, which are biolink:subclass_of (9 edges, 75.00%) and biolink:has_part (3 edges, 25.00%).

- Dendritic tendril star starting from the root node HP:0001832 (degree 27), and containing 10 nodes, with a maximal depth of 2, which are HP:0005194, HP:0008125, HP:0008078, HP:0004699 and HP:0010508. Its nodes have a single node type, which is biolink:NamedThing. Its edges have 2 edge types, which are biolink:subclass_of (8 edges, 72.73%) and biolink:has_part (3 edges, 27.27%).

- Dendritic tendril star starting from the root node HP:0001369 (degree 12), and containing 9 nodes, with a maximal depth of 2, which are HP:0040310, HP:0001370, N49475b12e70c4737aa3ca006af35921e, HP:0040311 and HP:0033037. Its nodes have a single node type, which is biolink:NamedThing. Its edges have 2 edge types, which are biolink:subclass_of (9 edges) and biolink:has_part.

And other 385 dendritic tendril stars.

Free-floating chains A free-floating chain is a tree with maximal degree two. We have detected a single free-floating chain in the graph, involving a total of 5 nodes (0.01%) and 4 edges.

- Free-floating chain starting from the root node RO:0002563 (degree 3), and containing 5 nodes, with a maximal depth of 2, which are RO:0002564, RO:0002464, http:&#x2f;&#x2f;ontologydesignpatterns.org&#x2f;wiki&#x2f;Submissions:N-Ary_Relation_Pattern_%28OWL_2%29 and RO:0002481. Its nodes have a single node type, which is biolink:NamedThing. Its edges have 2 edge types, which are biolink:subclass_of (4 edges) and rdfs:seeAlso.

Tendrils A tendril is a path starting from a node of degree one, connected to a strongly connected component. We have detected 4.32K tendrils in the graph, involving a total of 4.35K nodes (10.00%) and 4.35K edges (2.44%), with the largest one involving 3 nodes and 3 edges. The detected tendrils, sorted by decreasing size, are:

- Tendril starting from the root node NBO:0000607 (degree 3), and containing 3 nodes, with a maximal depth of 3, which are NBO:0000006, NBO:0000170 and NBO:0000304. Its nodes have a single node type, which is biolink:NamedThing. Its edges have a single edge type, which is biolink:subclass_of.
- Tendril starting from the root node NBO:0000389 (degree 3), and containing 3 nodes, with a maximal depth of 3, which are NBO:0000411, NBO:0000416 and NBO:0000417. Its nodes have a single node type, which is biolink:NamedThing. Its edges have a single edge type, which is biolink:subclass_of.
- Tendril starting from the root node HP:0100836 (degree 6), and containing 3 nodes, with a maximal depth of 3, which are HP:0002885, HP:0007129 and N520ca489d82d46018dce6dd76acb53a4. Its nodes have a single node type, which is biolink:NamedThing. Its edges have 2 edge types, which are biolink:subclass_of (3 edges) and biolink:has_part (2 edges).
- Tendril starting from the root node HP:0010786 (degree 5), and containing 3 nodes, with a maximal depth of 3, which are HP:0009725, HP:0002862 and HP:0006740. Its nodes have a single node type, which is biolink:NamedThing. Its edges have a single edge type, which is biolink:subclass_of.
- Tendril starting from the root node HP:0006304 (degree 3), and containing 2 nodes, with a maximal depth of 2, which are HP:0001566 and N11df713e04764d4ba280ea7097fd2 Its nodes have a single node type, which is biolink:NamedThing. Its edges have 2 edge types, which are biolink:has_part (2 edges) and biolink:subclass_of.
- Tendril starting from the root node HP:0500148 (degree 5), and containing 2 nodes, with a maximal depth of 2, which are HP:0410068 and Na4825961d95b44a9ae1d3cf34e55f Its nodes have a single node type, which is biolink:NamedThing. Its edges have 2 edge types, which are biolink:has_part (2 edges) and biolink:subclass_of.

And other 4.31K tendrils.

C.2 Cora, CiteSeer, and PubMed datasets used for node-label prediction experiments

Cora

The undirected graph Cora has 2.71K heterogeneous nodes and 5.28K edges. The graph contains 78 connected components, with the largest one containing 2.48K nodes and the smallest one containing 2 nodes. The RAM requirements for the nodes and edges data structures are 207.68KB and 16.34KB respectively.

Degree centrality The minimum node degree is 1, the maximum node degree is 168, the mode degree is 2, the mean degree is 3.90 and the node degree median is 3. The nodes with the highest degree centrality are 35 (degree 168 and node type Genetic_Algorithms), 6213 (degree 78 and node type Reinforcement_Learning), 1365 (degree 74 and node type Neural_Networks), 3229 (degree 65 and node type Neural_Networks) and 910 (degree 44 and node type Neural_Networks).

Node types The graph has 7 node types, which are Neural_Networks (818 nodes, 30.21%), Probabilistic_Methods (426 nodes, 15.73%), Genetic_Algorithms (418 nodes, 15.44%), Theory (351 nodes, 12.96%), Case_Based (298 nodes, 11.00%), Reinforcement_Learning (217 nodes, 8.01%) and Rule_Learning (180 nodes, 6.65%). The RAM requirement for the node types data structure is 141.72KB.

Topological Oddities A topological oddity is a set of nodes in the graph that *may be derived* by an error during the generation of the edge list of the graph and, depending on the task, could bias the results of topology-based models. In the following paragraph, we will describe the detected topological oddities.

Node tuples A node tuple is a connected component composed of two nodes. We have detected 57 node tuples in the graph, involving a total of 114 nodes (4.21%) and 57 edges (0.54%). The detected node tuples are:

- Node tuple containing the nodes 1105622 (node type Neural_Networks) and 430574 (node type Neural_Networks).
- Node tuple containing the nodes 116512 (node type Neural_Networks) and 1107808 (node type Neural_Networks).
- Node tuple containing the nodes 1107728 (node type Neural_Networks) and 115188 (node type Neural_Networks).
- Node tuple containing the nodes 1136040 (node type Neural_Networks) and 754594 (node type Neural_Networks).
- Node tuple containing the nodes 73972 (node type Case_Based) and 50980 (node type Case_Based).
- Node tuple containing the nodes 628458 (node type Neural_Networks) and 628459 (node type Neural_Networks).

- Node tuple containing the nodes 180301 (node type Probabilistic_Methods) and 1110628 (node type Probabilistic_Methods).
- Node tuple containing the nodes 1133008 (node type Neural_Networks) and 688824 (node type Neural_Networks).
- Node tuple containing the nodes 654519 (node type Genetic_Algorithms) and 1131754 (node type Genetic_Algorithms).
- Node tuple containing the nodes 49720 (node type Probabilistic_Methods) and 49753 (node type Probabilistic_Methods).
- Node tuple containing the nodes 133628 (node type Theory) and 1108570 (node type Theory).
- Node tuple containing the nodes 617378 (node type Neural_Networks) and 1130069 (node type Neural_Networks).
- Node tuple containing the nodes 529165 (node type Neural_Networks) and 1126315 (node type Neural_Networks).
- Node tuple containing the nodes 824245 (node type Neural_Networks) and 1139009 (node type Neural_Networks).
- Node tuple containing the nodes 820661 (node type Neural_Networks) and 817774 (node type Neural_Networks).

And other 42 node tuples.

Isomorphic node groups Isomorphic groups are nodes with exactly the same neighbours and node types (if present in the graph). Nodes in such groups are topologically indistinguishable, that is swapping their ID would not change the graph topology. We have detected 3 isomorphic node groups in the graph, involving a total of 6 nodes (0.22%) and 30 edges (0.28%). The detected isomorphic node groups, sorted by decreasing size, are:

- Group with 2 nodes (degree 5 and node type Genetic_Algorithms): 1104999 and 63832.
- Group with 2 nodes (degree 5 and node type Neural_Networks): 43698 and 31336.
- Group with 2 nodes (degree 5 and node type Neural_Networks): 1154123 and 1154124.

Dendritic trees A dendritic tree is a tree-like structure starting from a root node that is part of another strongly connected component. We have detected 13 dendritic trees in the graph, involving a total of 64 nodes (2.36%) and 64 edges (0.61%), with the largest one involving 9 nodes and 9 edges. The detected dendritic trees, sorted by decreasing size, are:

- Dendritic tree starting from the root node 16819 (degree 14), and containing 9 nodes, with a maximal depth of 4, which are 1131274, 643003, 644843, 1131189 and 645016 (degree 5). Its nodes have a single node type, which is Probabilistic_Methods.

- Dendritic tree starting from the root node 35 (degree 168), and containing 7 nodes, with a maximal depth of 2, which are 1152508, 1137466, 1128945, 1119505 and 15670. Its nodes have a single node type, which is Genetic_Algorithms.
- Dendritic tree starting from the root node 16437 (degree 6), and containing 6 nodes, with a maximal depth of 3, which are 51831, 430329, 127940 (degree 4), 416964 and 1114364. Its nodes have a single node type, which is Neural_Networks.
- Dendritic tree starting from the root node 424540 (degree 3), and containing 5 nodes, with a maximal depth of 3, which are 18536 (degree 3), 1106854, 86923 (degree 3), 18532 and 1114184. Its nodes have a single node type, which is Neural_Networks.
- Dendritic tree starting from the root node 910 (degree 44 and node type Neural_Networks), and containing 5 nodes, with a maximal depth of 2, which are 94953 (node type Neural_Networks), 1122460 (node type Neural_Networks), 1114118 (node type Neural_Networks), 245288 (node type Reinforcement_Learning) and 119712 (node type Genetic_Algorithms). Its nodes have 3 node types, which are Neural_Networks (3 nodes, 0.11%), Reinforcement_Learning and Genetic_Algorithms.
- Dendritic tree starting from the root node 13885 (degree 7 and node type Neural_Networks), and containing 5 nodes, with a maximal depth of 3, which are 84459 (node type Theory), 6238 (degree 4 and node type Theory), 1123991 (node type Probabilistic_Methods), 10793 (node type Theory) and 1130356 (node type Theory). Its nodes have 2 node types, which are Theory (4 nodes, 0.15%) and Probabilistic_Methods.

And other 7 dendritic trees.

Stars A star is a tree with a maximal depth of one, where nodes with maximal unique degree one are connected to a central root node with a high degree. We have detected 3 stars in the graph, involving a total of 9 nodes (0.33%) and 6 edges (0.06%). The detected stars are:

- Star starting from the root node 1112071 (degree 2), and containing 3 nodes, with a maximal depth of 1, which are 212107 and 212097. Its nodes have a single node type, which is Probabilistic_Methods.
- Star starting from the root node 1123215 (degree 2), and containing 3 nodes, with a maximal depth of 1, which are 288107 and 149139. Its nodes have a single node type, which is Theory.
- Star starting from the root node 9559 (degree 2), and containing 3 nodes, with a maximal depth of 1, which are 1102794 and 252725. Its nodes have a single node type, which is Rule_Learning.

Dendritic stars A dendritic star is a dendritic tree with a maximal depth of one, where nodes with maximal unique degree one are connected to a central root node with high degree and inside a strongly connected component. We have detected 29 dendritic stars in the graph, involving a total of 84 nodes (3.10%) and 84 edges (0.80%), with the largest one involving 12 nodes and 12 edges. The detected dendritic stars, sorted by decreasing size, are:

- Dendritic star starting from the root node 1365 (degree 74 and node type Neural_Networks), and containing 12 nodes, with a maximal depth of 1, which are 1105062 (node type Reinforcement_Learning), 853150 (node type Neural_Networks), 949318 (node type Neural_Networks), 1136442 (node type Neural_Networks) and 1132922 (node type Neural_Networks). Its nodes have 2 node types, which are Neural_Networks (11 nodes, 0.41%) and Reinforcement_Learning.
- Dendritic star starting from the root node 20193 (degree 23), and containing 11 nodes, with a maximal depth of 1, which are 1153877, 1153879, 1153889, 1130653 and 1130657. Its nodes have a single node type, which is Case_Based.
- Dendritic star starting from the root node 6913 (degree 12), and containing 4 nodes, with a maximal depth of 1, which are 1105011, 1131230, 703953 and 646289. Its nodes have a single node type, which is Probabilistic_Methods.
- Dendritic star starting from the root node 205196 (degree 9), and containing 4 nodes, with a maximal depth of 1, which are 628766, 1130568, 1130586 and 815073. Its nodes have a single node type, which is Neural_Networks.
- Dendritic star starting from the root node 89547 (degree 13 and node type Theory), and containing 3 nodes, with a maximal depth of 1, which are 1152379 (node type Theory), 1116328 (node type Neural_Networks) and 237376 (node type Theory). Its nodes have 2 node types, which are Theory (2 nodes, 0.07%) and Neural_Networks.
- Dendritic star starting from the root node 31353 (degree 19), and containing 3 nodes, with a maximal depth of 1, which are 286562, 1063773 and 686559. Its nodes have a single node type, which is Neural_Networks.

And other 23 dendritic stars.

Dendritic tendril stars A dendritic tendril star is a dendritic tree with a depth greater than one, where the arms of the star are tendrils. We have detected 7 dendritic tendril stars in the graph, involving a total of 28 nodes (1.03%) and 28 edges (0.27%), with the largest one involving 6 nodes and 6 edges. The detected dendritic tendril stars, sorted by decreasing size, are:

- Dendritic tendril star starting from the root node 3229 (degree 65 and node type Neural_Networks), and containing 6 nodes, with a maximal depth of 3, which are 919885 (node type Neural_Networks), 1125082 (node type Genetic_Algorithms), 7022 (node type Neural_Networks), 1112767 (node type Neural_Networks) and 226698 (node type Neural_Networks). Its nodes have 2 node types, which are Neural_Networks (5 nodes, 0.18%) and Genetic_Algorithms.
- Dendritic tendril star starting from the root node 643069 (degree 4 and node type Probabilistic_Methods), and containing 6 nodes, with a maximal depth of 5, which are 14090 (node type Probabilistic_Methods), 1131192 (node type Probabilistic_Methods), 1103016 (node type Neural_Networks), 14083 (node type Neural_Networks) and 62676 (node type Neural_Networks). Its nodes have 2 node types, which are Neural_Networks (4 nodes, 0.15%) and Probabilistic_Methods (2 nodes, 0.07%).
- Dendritic tendril star starting from the root node 3243 (degree 12 and node type Theory), and containing 4 nodes, with a maximal depth of 3, which are

1103610 (node type Theory), 854434 (node type Neural_Networks), 8961 (node type Reinforcement_Learning) and 1133390 (node type Theory). Its nodes have 3 node types, which are Theory (2 nodes, 0.07%), Neural_Networks and Reinforcement_Learning.

- Dendritic tendril star starting from the root node 5086 (degree 12), and containing 3 nodes, with a maximal depth of 2, which are 354004, 1105698 and 1118546. Its nodes have a single node type, which is Probabilistic_Methods.
- Dendritic tendril star starting from the root node 35863 (degree 5 and node type Reinforcement_Learning), and containing 3 nodes, with a maximal depth of 2, which are 28359 (node type Reinforcement_Learning), 134060 (node type Theory) and 481073 (node type Reinforcement_Learning). Its nodes have 2 node types, which are Reinforcement_Learning (2 nodes, 0.07%) and Theory.
- Dendritic tendril star starting from the root node 162080 (degree 5), and containing 3 nodes, with a maximal depth of 2, which are 738941, 1135345 and 1135455. Its nodes have a single node type, which is Neural_Networks.

And another dendritic tendril star.

Free-floating chains A free-floating chain is a tree with maximal degree two. We have detected 2 free-floating chains in the graph, involving a total of 8 nodes (0.30%) and 6 edges (0.06%). The detected free-floating chains are:

- Free-floating chain starting from the root node 375825 (degree 2), and containing 4 nodes, with a maximal depth of 2, which are 1119623, 421481 and 111770. Its nodes have a single node type, which is Probabilistic_Methods.
- Free-floating chain starting from the root node 430711 (degree 2), and containing 4 nodes, with a maximal depth of 2, which are 671052, 1132416 and 1132406. Its nodes have a single node type, which is Neural_Networks.

Tendrils A tendril is a path starting from a node of degree one, connected to a strongly connected component. We have detected 224 tendrils in the graph, involving a total of 265 nodes (9.79%) and 265 edges (2.51%), with the largest one involving 4 nodes and 4 edges. The detected tendrils, sorted by decreasing size, are:

- Tendril starting from the root node 83847 (degree 4), and containing 4 nodes, with a maximal depth of 4, which are 1130678, 630890, 233106 and 12275. Its nodes have a single node type, which is Neural_Networks.
- Tendril starting from the root node 1140543 (degree 5), and containing 3 nodes, with a maximal depth of 3, which are 120817, 1109873 and 163235. Its nodes have a single node type, which is Neural_Networks.
- Tendril starting from the root node 683404 (degree 4), and containing 3 nodes, with a maximal depth of 3, which are 683360, 522338 and 1132864. Its nodes have a single node type, which is Probabilistic_Methods.
- Tendril starting from the root node 20534 (degree 10 and node type Reinforcement_Learning), and containing 3 nodes, with a maximal depth of 3, which are 13972 (node type Reinforcement_Learning), 1126050 (node type Reinforcement_Learning) and 93318 (node type Neural_Networks). Its nodes have 2 node types, which are Reinforcement_Learning (2 nodes, 0.07%) and Neural_Networks.

- Tendril starting from the root node 66751 (degree 5), and containing 3 nodes, with a maximal depth of 3, which are 1138043, 77108 and 77112. Its nodes have a single node type, which is Theory.
- Tendril starting from the root node 3231 (degree 36 and node type Theory), and containing 2 nodes, with a maximal depth of 2, which are 1113926 (node type Neural_Networks) and 250566 (node type Case_Based). Its nodes have 2 node types, which are Neural_Networks and Case_Based.

And other 218 tendrils.

CiteSeer

The undirected graph CiteSeer has 3.31K heterogeneous nodes and 4.66K edges. The graph contains 438 connected components (of which 48 are disconnected nodes), with the largest one containing 2.11K nodes and the smallest one containing a single node. The RAM requirements for the nodes and edges data structures are 335.92KB and 14.76KB respectively.

Degree centrality The minimum node degree is 1, the maximum node degree is 99, the mode degree is 1, the mean degree is 2.78 and the node degree median is 2. The nodes with the highest degree centrality are brin98anatomy (degree 99 and node type IR), rao95bdi (degree 51 and node type Agents), chakrabarti98automatic (degree 35 and node type IR), bharat98improved (degree 34 and node type IR) and lawrence98searching (degree 30 and node type IR).

Node types The graph has 6 node types, which are DB (701 nodes, 21.17%), IR (668 nodes, 20.17%), Agents (596 nodes, 18.00%), ML (590 nodes, 17.81%), HCI (508 nodes, 15.34%) and AI (249 nodes, 7.52%). The RAM requirement for the node types data structure is 172.75KB.

Topological Oddities A topological oddity is a set of nodes in the graph that *may be derived* by an error during the generation of the edge list of the graph and, depending on the task, could bias the results of topology-based models. In the following paragraph, we will describe the detected topological oddities.

Singleton nodes with self-loops A singleton node with self-loops is a node disconnected from all other nodes except itself. We have detected 48 singleton nodes with self-loops in the graph, involving a total of 48 nodes (1.45%) and 48 edges (0.52%). The detected singleton nodes with self-loops are:

- 408356 (node type DB)
- eiter98firstorder (node type AI)
- 156949 (node type DB)
- bruno01stholes (node type DB)
- kumar01behaviorbased (node type ML)
- lin01efficiently (node type DB)
- park01segmentbased (node type DB)
- tabuada01feasible (node type Agents)
- 146066 (node type DB)
- 190915 (node type DB)
- 202521 (node type DB)

- 274436 (node type HCI)
- 346149 (node type ML)
- 423028 (node type HCI)
- 43511 (node type DB)

And other 33 singleton nodes with self-loops.

Node tuples A node tuple is a connected component composed of two nodes. We have detected 252 node tuples in the graph, involving a total of 504 nodes (15.22%) and 252 edges (2.74%). The detected node tuples are:

- Node tuple containing the nodes zhang99situated (node type ML) and zhang99towards (node type IR).
- Node tuple containing the nodes zhang01evolutionary (node type DB) and zhang99evolving (node type ML).
- Node tuple containing the nodes wolski00design (node type DB) and wolski98fuzzy (node type ML).
- Node tuple containing the nodes wills00open (node type ML) and wills01open (node type ML).
- Node tuple containing the nodes vilalta00quantification (node type ML) and vilalta01rule (node type ML).
- Node tuple containing the nodes vazov01system (node type IR) and wonsever01contextual (node type IR).
- Node tuple containing the nodes vasconcelos00bayesian (node type ML) and vasconcelos99probabilistic (node type ML).
- Node tuple containing the nodes valencia98algebraic (node type AI) and valencia98hitch (node type AI).
- Node tuple containing the nodes tzouramanis99overlapping (node type DB) and tzouramanis99processing (node type DB).
- Node tuple containing the nodes tourapis01advanced (node type ML) and tourapis01temporal (node type ML).
- Node tuple containing the nodes sterritt00exploring (node type AI) and sterritt01soft (degree 2 and node type ML).
- Node tuple containing the nodes rosenthal00view (node type DB) and rosenthal01administering (node type DB).
- Node tuple containing the nodes paulson00inductive (node type ML) and steel02finding (node type ML).
- Node tuple containing the nodes oria99defining (node type DB) and oria99visualmoql (node type DB).
- Node tuple containing the nodes oard01clef (node type HCI) and oard01evaluating (node type HCI).

And other 237 node tuples.

Isomorphic node groups Isomorphic groups are nodes with exactly the same neighbours and node types (if present in the graph). Nodes in such groups are topologically indistinguishable, that is swapping their ID would not change the graph topology. We have detected 11 isomorphic node groups in the graph, involving a total of 23 nodes (0.69%) and 152 edges (1.65%), with the largest one involving 3 nodes and 24 edges. The detected isomorphic node groups, sorted by decreasing size, are:

- Group with 2 nodes (degree 12 and node type IR): 65816 and baker98distributional.
- Group with 3 nodes (degree 6 and node type IR): 540380, 532128 and 536016.
- Group with 2 nodes (degree 8 and node type HCI): 150449 and billinghurst98wearable.
- Group with 2 nodes (degree 7 and node type DB): 254693 and 352789.
- Group with 2 nodes (degree 6 and node type IR): 323867 and lawrence99searching.
- Group with 2 nodes (degree 6 and node type IR): 509763 and nguyen00active.
- Group with 2 nodes (degree 6 and node type Agents): 295535 and kumar00adaptive.
- Group with 2 nodes (degree 6 and node type DB): artale01reasoning and 454077.
- Group with 2 nodes (degree 6 and node type DB): jagadish99querying and 106339.
- Group with 2 nodes (degree 5 and node type DB): 486074 and 506324.
- Group with 2 nodes (degree 5 and node type AI): 28223 and 40513.

Trees A tree is a connected component with n nodes and n-1 edges. We have detected 7 trees in the graph, involving a total of 46 nodes (1.39%) and 39 edges (0.42%), with the largest one involving 9 nodes and 8 edges. The detected trees, sorted by decreasing size, are:

- Tree starting from the root node edmond98achieving (degree 2 and node type DB), and containing 9 nodes, with a maximal depth of 3, which are barros97towards (node type HCI), manolescu01microworkflow (node type IR), barros96business (degree 4 and node type HCI), muth99integrating (node type DB) and 70863 (node type HCI). Its nodes have 3 node types, which are HCI (5 nodes, 0.15%), DB (2 nodes, 0.06%) and IR.
- Tree starting from the root node bonnet99query (degree 3 and node type DB), and containing 8 nodes, with a maximal depth of 2, which are 272797 (node type DB), bonnet00query (node type DB), heidemann01building (degree 3 and node type IR), chen00algebraic (node type DB) and dekhtyar99probabilistic (node type DB). Its nodes have 3 node types, which are DB (4 nodes, 0.12%), IR (2 nodes, 0.06%) and Agents.
- Tree starting from the root node kim01secret (degree 2), and containing 7 nodes, with a maximal depth of 3, which are 467998, vagina03cryptographic (degree 4), 482071, 496883 and tan01trust. Its nodes have a single node type, which is Agents.
- Tree starting from the root node cadoli98survey (degree 2 and node type AI), and containing 6 nodes, with a maximal depth of 2, which are 83444 (degree 4 and node type AI), prendinger00hyper (node type DB), 210930 (node type AI),

bellardo00implementing (node type AI) and eiter00difference (node type DB). Its nodes have 2 node types, which are AI (3 nodes, 0.09%) and DB (2 nodes, 0.06%).

- Tree starting from the root node das98rule (degree 2 and node type ML), and containing 6 nodes, with a maximal depth of 2, which are 169000 (degree 4 and node type DB), 74893 (node type AI), 469106 (node type DB), 534720 (node type DB) and rodriguez00learning (node type DB). Its nodes have 2 node types, which are DB (4 nodes, 0.12%) and AI.
- Tree starting from the root node sampaio98deductive (degree 2), and containing 5 nodes, with a maximal depth of 2, which are murray98kaleidoquery (degree 3), sampaio00design, 116696 and fegaras99voodoo. Its nodes have a single node type, which is DB.

And another tree.

Dendritic trees A dendritic tree is a tree-like structure starting from a root node that is part of another strongly connected component. We have detected 44 dendritic trees in the graph, involving a total of 291 nodes (8.79%) and 291 edges (3.16%), with the largest one involving 26 nodes and 26 edges. The detected dendritic trees, sorted by decreasing size, are:

- Dendritic tree starting from the root node rao95bdi (degree 51 and node type Agents), and containing 26 nodes, with a maximal depth of 4, which are 243827 (node type Agents), 257383 (node type Agents), 270678 (node type Agents), 318212 (node type Agents) and 445758 (node type Agents). Its nodes have 3 node types, which are Agents (23 nodes, 0.69%), AI (2 nodes, 0.06%) and IR.
- Dendritic tree starting from the root node brin98anatomy (degree 99 and node type IR), and containing 21 nodes, with a maximal depth of 5, which are 128239 (node type IR), 165504 (node type IR), 500641 (node type IR), 520488 (node type IR) and 528932 (degree 3 and node type IR). Its nodes have 4 node types, which are IR (14 nodes, 0.42%), ML (5 nodes, 0.15%), DB and HCI.
- Dendritic tree starting from the root node essa99computer (degree 3 and node type HCI), and containing 14 nodes, with a maximal depth of 4, which are 24549 (degree 9 and node type DB), 28031 (node type ML), baker00hallucinating (node type ML), martinez00recognition (node type ML) and moghaddam98beyond (degree 4 and node type DB). Its nodes have 3 node types, which are ML (10 nodes, 0.30%), DB (3 nodes, 0.09%) and HCI.
- Dendritic tree starting from the root node howe97savvysearch (degree 21 and node type IR), and containing 12 nodes, with a maximal depth of 7, which are scime01websifter (node type IR), tzitzikas01democratic (node type IR), 496354 (node type DB), 537920 (node type IR) and mcilraith01semantic (degree 3 and node type IR). Its nodes have 5 node types, which are IR (5 nodes, 0.15%), DB (4 nodes, 0.12%), Agents, ML and HCI.
- Dendritic tree starting from the root node 90507 (degree 10 and node type IR), and containing 12 nodes, with a maximal depth of 2, which are cunningham01developing (degree 4 and node type IR), he00comparative (node type IR), itskevitch01automatic (degree 7 and node type IR), cunningham99experience

(node type IR) and gaizauskas98information (node type IR). Its nodes have 2 node types, which are IR (10 nodes, 0.30%) and DB (2 nodes, 0.06%).

- Dendritic tree starting from the root node bergamaschi99semantic (degree 11 and node type DB), and containing 12 nodes, with a maximal depth of 5, which are 254597 (degree 5 and node type IR), 311175 (node type DB), 471747 (node type IR), 496736 (node type IR) and feng01towards (degree 4 and node type IR). Its nodes have 2 node types, which are IR (8 nodes, 0.24%) and DB (4 nodes, 0.12%).

And other 38 dendritic trees.

Stars A star is a tree with a maximal depth of one, where nodes with maximal unique degree one are connected to a central root node with a high degree. We have detected 65 stars in the graph, involving a total of 207 nodes (6.25%) and 142 edges (1.54%), with the largest one involving 5 nodes and 4 edges. The detected stars, sorted by decreasing size, are:

- Star starting from the root node white98towards (degree 4), and containing 5 nodes, with a maximal depth of 1, which are 205160, 409610, ferguson95role and wittner00network. Its nodes have a single node type, which is Agents.
- Star starting from the root node 196348 (degree 3), and containing 4 nodes, with a maximal depth of 1, which are 263968, 63224 and sengupta99constructing. Its nodes have a single node type, which is DB.
- Star starting from the root node degaris99building (degree 3), and containing 4 nodes, with a maximal depth of 1, which are 261630, degaris00simulating and degaris99evolving. Its nodes have a single node type, which is ML.
- Star starting from the root node markatos99caching (degree 3 and node type DB), and containing 4 nodes, with a maximal depth of 1, which are glance00community (node type HCI), markatos98effective (node type IR) and xie02locality (node type IR). Its nodes have 2 node types, which are IR (2 nodes, 0.06%) and HCI.
- Star starting from the root node takahashi00location (degree 3 and node type IR), and containing 4 nodes, with a maximal depth of 1, which are 539969 (node type IR), ishida99digital (node type Agents) and takahashi98mobile (node type IR). Its nodes have 2 node types, which are IR (2 nodes, 0.06%) and Agents.
- Star starting from the root node timm01synthesis (degree 3 and node type Agents), and containing 4 nodes, with a maximal depth of 1, which are timm00multiagent (node type IR), timm01enterprise (node type Agents) and toenshoff01flexible (node type Agents). Its nodes have 2 node types, which are Agents (2 nodes, 0.06%) and IR.

And other 59 stars.

Dendritic stars A dendritic star is a dendritic tree with a maximal depth of one, where nodes with maximal unique degree one are connected to a central root node with high degree and inside a strongly connected component. We have detected 61 dendritic stars in the graph, involving a total of 137 nodes (4.14%) and 137 edges (1.49%), with the largest one involving 5 nodes and 5 edges. The detected dendritic stars, sorted by decreasing size, are:

- Dendritic star starting from the root node liu98relationlog (degree 11), and containing 5 nodes, with a maximal depth of 1, which are 82903, fraternali98proceedings, liu01rulebased, liu98logical and liu99partial. Its nodes have a single node type, which is DB.
- Dendritic star starting from the root node decker95environment (degree 17 and node type Agents), and containing 5 nodes, with a maximal depth of 1, which are bilgic97risk (node type Agents), bilgic97system (node type Agents), decker98coordinating (node type Agents), obrst97constraints (node type Agents) and prasad96offline (node type ML). Its nodes have 2 node types, which are Agents (4 nodes, 0.12%) and ML.
- Dendritic star starting from the root node 78547 (degree 10 and node type IR), and containing 4 nodes, with a maximal depth of 1, which are 194227 (node type IR), ahanger99technique (node type IR), jaimes00integrating (node type ML) and slaughter00open (node type IR). Its nodes have 2 node types, which are IR (3 nodes, 0.09%) and ML.
- Dendritic star starting from the root node schattenberg00planning (degree 6 and node type Agents), and containing 3 nodes, with a maximal depth of 1, which are kitano99robocup (node type HCI), logan00distributed (node type Agents) and rintanen98planning (node type AI). Its nodes have 3 node types, which are HCI, Agents and AI.
- Dendritic star starting from the root node khaled98gado (degree 7), and containing 3 nodes, with a maximal depth of 1, which are bourdeau99three, davison98applying and rasheed98adaptive. Its nodes have a single node type, which is ML.
- Dendritic star starting from the root node godfrey98integrity (degree 5 and node type DB), and containing 3 nodes, with a maximal depth of 1, which are 71092 (node type DB), chan99possible (node type DB) and godfrey97minimization (node type IR). Its nodes have 2 node types, which are DB (2 nodes, 0.06%) and IR.

And other 55 dendritic stars.

Dendritic tendril stars A dendritic tendril star is a dendritic tree with a depth greater than one, where the arms of the star are tendrils. We have detected 23 dendritic tendril stars in the graph, involving a total of 93 nodes (2.81%) and 93 edges (1.01%), with the largest one involving 7 nodes and 7 edges. The detected dendritic tendril stars, sorted by decreasing size, are:

- Dendritic tendril star starting from the root node aha91casebased (degree 21), and containing 7 nodes, with a maximal depth of 2, which are 5234, 75123, 77029, mair99investigation and petrak95objectoriented. Its nodes have a single node type, which is ML.
- Dendritic tendril star starting from the root node holtman98automatic (degree 6), and containing 6 nodes, with a maximal depth of 3, which are 300668, 340027, schaller99objectivitydb, stockinger01design and 35804. Its nodes have a single node type, which is DB.
- Dendritic tendril star starting from the root node papadopoulos00models (degree 7 and node type HCI), and containing 5 nodes, with a maximal depth of

3, which are 532291 (node type Agents), chen99dynamic (node type Agents), skarmeas99component (node type Agents), bardram97plans (node type HCI) and reddy01coordinating (node type HCI). Its nodes have 2 node types, which are Agents (3 nodes, 0.09%) and HCI (2 nodes, 0.06%).

- Dendritic tendril star starting from the root node beigi97metaseek (degree 10 and node type IR), and containing 5 nodes, with a maximal depth of 2, which are 291240 (node type HCI), 305534 (node type ML), 420817 (node type HCI), laaksonen99picsom (node type IR) and koskela00picsom (node type HCI). Its nodes have 3 node types, which are HCI (3 nodes, 0.09%), ML and IR.
- Dendritic tendril star starting from the root node langley95applications (degree 6 and node type ML), and containing 5 nodes, with a maximal depth of 3, which are 162298 (node type ML), giraud-carrier98beyond (node type AI), lau00version (node type ML), 415731 (node type IR) and lau99programming (node type ML). Its nodes have 3 node types, which are ML (3 nodes, 0.09%), AI and IR.
- Dendritic tendril star starting from the root node thomas98wearable (degree 6), and containing 4 nodes, with a maximal depth of 3, which are 503243, 534400, ockerman98preliminary and pentland99digital. Its nodes have a single node type, which is HCI.

And other 17 dendritic tendril stars.

Free-floating chains A free-floating chain is a tree with maximal degree two. We have detected 15 free-floating chains in the graph, involving a total of 65 nodes (1.96%) and 50 edges (0.54%), with the largest one involving 7 nodes and 6 edges. The detected free-floating chains, sorted by decreasing size, are:

- Free-floating chain starting from the root node mcroy95repair (degree 5 and node type Agents), and containing 7 nodes, with a maximal depth of 2, which are 3489 (node type AI), ardissono00plan (node type Agents), ardissono96uso (node type Agents), mcroy98achieving (node type AI) and traum99speech (node type Agents). Its nodes have 2 node types, which are Agents (4 nodes, 0.12%) and AI (2 nodes, 0.06%).
- Free-floating chain starting from the root node liu00extended (degree 3 and node type ML), and containing 5 nodes, with a maximal depth of 2, which are 521000 (node type ML), frank98generating (node type ML), hekanaho98dogma (node type AI) and fertig99fuzzy (node type ML). Its nodes have 2 node types, which are ML (3 nodes, 0.09%) and AI.
- Free-floating chain starting from the root node hatzilygeroudis00neurules (degree 4), and containing 5 nodes, with a maximal depth of 2, which are hatzilygeroudis02multiinference, prentzas01webbased, prentzas02webbased and hatzilygeroudis01hymes. Its nodes have a single node type, which is AI.
- Free-floating chain starting from the root node chen98learningbased (degree 2), and containing 4 nodes, with a maximal depth of 2, which are sreerupa98dynamic, weng98visionguided and 208646. Its nodes have a single node type, which is ML.
- Free-floating chain starting from the root node 288424 (degree 2), and containing 4 nodes, with a maximal depth of 2, which are granlund01patternsupported, vanwelie00patterns and borchers00pattern. Its nodes have a single node type, which is HCI.

- Free-floating chain starting from the root node 461740 (degree 2 and node type ML), and containing 4 nodes, with a maximal depth of 2, which are 149759 (node type AI), nguyen98strict (node type AI) and 496719 (node type ML). Its nodes have 2 node types, which are AI (2 nodes, 0.06%) and ML.

And other 9 free-floating chains.

Tendrils A tendril is a path starting from a node of degree one, connected to a strongly connected component. We have detected 240 tendrils in the graph, involving a total of 321 nodes (9.69%) and 321 edges (3.49%), with the largest one involving 3 nodes and 3 edges. The detected tendrils, sorted by decreasing size, are:

- Tendril starting from the root node bharat99comparison (degree 5), and containing 3 nodes, with a maximal depth of 3, which are 502499, almeida01analyzing and heinonen96www. Its nodes have a single node type, which is IR.
- Tendril starting from the root node kubiatowicz00oceanstore (degree 4 and node type HCI), and containing 3 nodes, with a maximal depth of 3, which are 525023 (node type HCI), grimm01systems (node type Agents) and jennings01aspects (node type HCI). Its nodes have 2 node types, which are HCI (2 nodes, 0.06%) and Agents.
- Tendril starting from the root node stolzenburg01from (degree 3 and node type Agents), and containing 3 nodes, with a maximal depth of 3, which are 335912 (node type ML), boutilier01partialorder (node type Agents) and brafman98knowledge (node type AI). Its nodes have 3 node types, which are ML, Agents and AI.
- Tendril starting from the root node 35592 (degree 5), and containing 3 nodes, with a maximal depth of 3, which are 69807, onoda98asymptotic and 12247. Its nodes have a single node type, which is ML.
- Tendril starting from the root node 242172 (degree 10), and containing 3 nodes, with a maximal depth of 3, which are rao96agentspeakl, hindriks00architecture and 491166. Its nodes have a single node type, which is Agents.
- Tendril starting from the root node oviatt99ten (degree 4), and containing 3 nodes, with a maximal depth of 3, which are 443913, conati00toward and 452812. Its nodes have a single node type, which is HCI.

And other 234 tendrils.

Cora

The undirected graph Cora has 2.71K heterogeneous nodes and 5.28K edges. The graph contains 78 connected components, with the largest one containing 2.48K nodes and the smallest one containing 2 nodes. The RAM requirements for the nodes and edges data structures are 207.68KB and 16.34KB respectively.

Degree centrality The minimum node degree is 1, the maximum node degree is 168, the mode degree is 2, the mean degree is 3.90 and the node degree median is 3. The nodes with the highest degree centrality are 35 (degree 168 and node type Genetic_Algorithms), 6213 (degree 78 and node type Reinforcement_Learning), 1365 (degree 74 and node type Neural_Networks), 3229 (degree 65 and node type Neural_Networks) and 910 (degree 44 and node type Neural_Networks).

Node types The graph has 7 node types, which are Neural_Networks (818 nodes, 30.21%), Probabilistic_Methods (426 nodes, 15.73%), Genetic_Algorithms (418 nodes, 15.44%), Theory (351 nodes, 12.96%), Case_Based (298 nodes, 11.00%), Reinforcement_Learning (217 nodes, 8.01%) and Rule_Learning (180 nodes, 6.65%). The RAM requirement for the node types data structure is 141.72KB.

Topological Oddities A topological oddity is a set of nodes in the graph that *may be derived* by an error during the generation of the edge list of the graph and, depending on the task, could bias the results of topology-based models. In the following paragraph, we will describe the detected topological oddities.

Node tuples A node tuple is a connected component composed of two nodes. We have detected 57 node tuples in the graph, involving a total of 114 nodes (4.21%) and 57 edges (0.54%). The detected node tuples are:

- Node tuple containing the nodes 1105622 (node type Neural_Networks) and 430574 (node type Neural_Networks).
- Node tuple containing the nodes 116512 (node type Neural_Networks) and 1107808 (node type Neural_Networks).
- Node tuple containing the nodes 1107728 (node type Neural_Networks) and 115188 (node type Neural_Networks).
- Node tuple containing the nodes 1136040 (node type Neural_Networks) and 754594 (node type Neural_Networks).
- Node tuple containing the nodes 73972 (node type Case_Based) and 50980 (node type Case_Based).
- Node tuple containing the nodes 628458 (node type Neural_Networks) and 628459 (node type Neural_Networks).
- Node tuple containing the nodes 180301 (node type Probabilistic_Methods) and 1110628 (node type Probabilistic_Methods).
- Node tuple containing the nodes 1133008 (node type Neural_Networks) and 688824 (node type Neural_Networks).
- Node tuple containing the nodes 654519 (node type Genetic_Algorithms) and 1131754 (node type Genetic_Algorithms).
- Node tuple containing the nodes 49720 (node type Probabilistic_Methods) and 49753 (node type Probabilistic_Methods).
- Node tuple containing the nodes 133628 (node type Theory) and 1108570 (node type Theory).
- Node tuple containing the nodes 617378 (node type Neural_Networks) and 1130069 (node type Neural_Networks).
- Node tuple containing the nodes 529165 (node type Neural_Networks) and 1126315 (node type Neural_Networks).
- Node tuple containing the nodes 824245 (node type Neural_Networks) and 1139009 (node type Neural_Networks).

- Node tuple containing the nodes 820661 (node type Neural_Networks) and 817774 (node type Neural_Networks).

And other 42 node tuples.

Isomorphic node groups Isomorphic groups are nodes with exactly the same neighbours and node types (if present in the graph). Nodes in such groups are topologically indistinguishable, that is swapping their ID would not change the graph topology. We have detected 3 isomorphic node groups in the graph, involving a total of 6 nodes (0.22%) and 30 edges (0.28%). The detected isomorphic node groups, sorted by decreasing size, are:

- Group with 2 nodes (degree 5 and node type Genetic_Algorithms): 1104999 and 63832.
- Group with 2 nodes (degree 5 and node type Neural_Networks): 43698 and 31336.
- Group with 2 nodes (degree 5 and node type Neural_Networks): 1154123 and 1154124.

Dendritic trees A dendritic tree is a tree-like structure starting from a root node that is part of another strongly connected component. We have detected 13 dendritic trees in the graph, involving a total of 64 nodes (2.36%) and 64 edges (0.61%), with the largest one involving 9 nodes and 9 edges. The detected dendritic trees, sorted by decreasing size, are:

- Dendritic tree starting from the root node 16819 (degree 14), and containing 9 nodes, with a maximal depth of 4, which are 1131274, 643003, 644843, 1131189 and 645016 (degree 5). Its nodes have a single node type, which is Probabilistic_Methods.
- Dendritic tree starting from the root node 35 (degree 168), and containing 7 nodes, with a maximal depth of 2, which are 1152508, 1137466, 1128945, 1119505 and 15670. Its nodes have a single node type, which is Genetic_Algorithms.
- Dendritic tree starting from the root node 16437 (degree 6), and containing 6 nodes, with a maximal depth of 3, which are 51831, 430329, 127940 (degree 4), 416964 and 1114364. Its nodes have a single node type, which is Neural_Networks.
- Dendritic tree starting from the root node 424540 (degree 3), and containing 5 nodes, with a maximal depth of 3, which are 18536 (degree 3), 1106854, 86923 (degree 3), 18532 and 1114184. Its nodes have a single node type, which is Neural_Networks.
- Dendritic tree starting from the root node 910 (degree 44 and node type Neural_Networks), and containing 5 nodes, with a maximal depth of 2, which are 94953 (node type Neural_Networks), 1122460 (node type Neural_Networks), 1114118 (node type Neural_Networks), 245288 (node type Reinforcement_Learning) and 119712 (node type Genetic_Algorithms). Its nodes have 3 node types, which are Neural_Networks (3 nodes, 0.11%), Reinforcement_Learning and Genetic_Algorithms.

- Dendritic tree starting from the root node 13885 (degree 7 and node type Neural_Networks), and containing 5 nodes, with a maximal depth of 3, which are 84459 (node type Theory), 6238 (degree 4 and node type Theory), 1123991 (node type Probabilistic_Methods), 10793 (node type Theory) and 1130356 (node type Theory). Its nodes have 2 node types, which are Theory (4 nodes, 0.15%) and Probabilistic_Methods.

And other 7 dendritic trees.

Stars A star is a tree with a maximal depth of one, where nodes with maximal unique degree one are connected to a central root node with a high degree. We have detected 3 stars in the graph, involving a total of 9 nodes (0.33%) and 6 edges (0.06%). The detected stars are:

- Star starting from the root node 1112071 (degree 2), and containing 3 nodes, with a maximal depth of 1, which are 212107 and 212097. Its nodes have a single node type, which is Probabilistic_Methods.
- Star starting from the root node 1123215 (degree 2), and containing 3 nodes, with a maximal depth of 1, which are 288107 and 149139. Its nodes have a single node type, which is Theory.
- Star starting from the root node 9559 (degree 2), and containing 3 nodes, with a maximal depth of 1, which are 1102794 and 252725. Its nodes have a single node type, which is Rule_Learning.

Dendritic stars A dendritic star is a dendritic tree with a maximal depth of one, where nodes with maximal unique degree one are connected to a central root node with high degree and inside a strongly connected component. We have detected 29 dendritic stars in the graph, involving a total of 84 nodes (3.10%) and 84 edges (0.80%), with the largest one involving 12 nodes and 12 edges. The detected dendritic stars, sorted by decreasing size, are:

- Dendritic star starting from the root node 1365 (degree 74 and node type Neural_Networks), and containing 12 nodes, with a maximal depth of 1, which are 1105062 (node type Reinforcement_Learning), 853150 (node type Neural_Networks), 949318 (node type Neural_Networks), 1136442 (node type Neural_Networks) and 1132922 (node type Neural_Networks). Its nodes have 2 node types, which are Neural_Networks (11 nodes, 0.41%) and Reinforcement_Learning.
- Dendritic star starting from the root node 20193 (degree 23), and containing 11 nodes, with a maximal depth of 1, which are 1153877, 1153879, 1153889, 1130653 and 1130657. Its nodes have a single node type, which is Case_Based.
- Dendritic star starting from the root node 6913 (degree 12), and containing 4 nodes, with a maximal depth of 1, which are 1105011, 1131230, 703953 and 646289. Its nodes have a single node type, which is Probabilistic_Methods.
- Dendritic star starting from the root node 205196 (degree 9), and containing 4 nodes, with a maximal depth of 1, which are 628766, 1130568, 1130586 and 815073. Its nodes have a single node type, which is Neural_Networks.
- Dendritic star starting from the root node 89547 (degree 13 and node type Theory), and containing 3 nodes, with a maximal depth of 1, which are 1152379

(node type Theory), 1116328 (node type Neural_Networks) and 237376 (node type Theory). Its nodes have 2 node types, which are Theory (2 nodes, 0.07%) and Neural_Networks.

- Dendritic star starting from the root node 31353 (degree 19), and containing 3 nodes, with a maximal depth of 1, which are 286562, 1063773 and 686559. Its nodes have a single node type, which is Neural_Networks.

And other 23 dendritic stars.

Dendritic tendril stars A dendritic tendril star is a dendritic tree with a depth greater than one, where the arms of the star are tendrils. We have detected 7 dendritic tendril stars in the graph, involving a total of 28 nodes (1.03%) and 28 edges (0.27%), with the largest one involving 6 nodes and 6 edges. The detected dendritic tendril stars, sorted by decreasing size, are:

- Dendritic tendril star starting from the root node 3229 (degree 65 and node type Neural_Networks), and containing 6 nodes, with a maximal depth of 3, which are 919885 (node type Neural_Networks), 1125082 (node type Genetic_Algorithms), 7022 (node type Neural_Networks), 1112767 (node type Neural_Networks) and 226698 (node type Neural_Networks). Its nodes have 2 node types, which are Neural_Networks (5 nodes, 0.18%) and Genetic_Algorithms.
- Dendritic tendril star starting from the root node 643069 (degree 4 and node type Probabilistic_Methods), and containing 6 nodes, with a maximal depth of 5, which are 14090 (node type Probabilistic_Methods), 1131192 (node type Probabilistic_Methods), 1103016 (node type Neural_Networks), 14083 (node type Neural_Networks) and 62676 (node type Neural_Networks). Its nodes have 2 node types, which are Neural_Networks (4 nodes, 0.15%) and Probabilistic_Methods (2 nodes, 0.07%).
- Dendritic tendril star starting from the root node 3243 (degree 12 and node type Theory), and containing 4 nodes, with a maximal depth of 3, which are 1103610 (node type Theory), 854434 (node type Neural_Networks), 8961 (node type Reinforcement_Learning) and 1133390 (node type Theory). Its nodes have 3 node types, which are Theory (2 nodes, 0.07%), Neural_Networks and Reinforcement_Learning.
- Dendritic tendril star starting from the root node 5086 (degree 12), and containing 3 nodes, with a maximal depth of 2, which are 354004, 1105698 and 1118546. Its nodes have a single node type, which is Probabilistic_Methods.
- Dendritic tendril star starting from the root node 35863 (degree 5 and node type Reinforcement_Learning), and containing 3 nodes, with a maximal depth of 2, which are 28359 (node type Reinforcement_Learning), 134060 (node type Theory) and 481073 (node type Reinforcement_Learning). Its nodes have 2 node types, which are Reinforcement_Learning (2 nodes, 0.07%) and Theory.
- Dendritic tendril star starting from the root node 162080 (degree 5), and containing 3 nodes, with a maximal depth of 2, which are 738941, 1135345 and 1135455. Its nodes have a single node type, which is Neural_Networks.

And another dendritic tendril star.

Free-floating chains A free-floating chain is a tree with maximal degree two. We have detected 2 free-floating chains in the graph, involving a total of 8 nodes (0.30%) and 6 edges (0.06%). The detected free-floating chains are:

- Free-floating chain starting from the root node 375825 (degree 2), and containing 4 nodes, with a maximal depth of 2, which are 1119623, 421481 and 111770. Its nodes have a single node type, which is Probabilistic_Methods.
- Free-floating chain starting from the root node 430711 (degree 2), and containing 4 nodes, with a maximal depth of 2, which are 671052, 1132416 and 1132406. Its nodes have a single node type, which is Neural_Networks.

Tendrils A tendril is a path starting from a node of degree one, connected to a strongly connected component. We have detected 224 tendrils in the graph, involving a total of 265 nodes (9.79%) and 265 edges (2.51%), with the largest one involving 4 nodes and 4 edges. The detected tendrils, sorted by decreasing size, are:

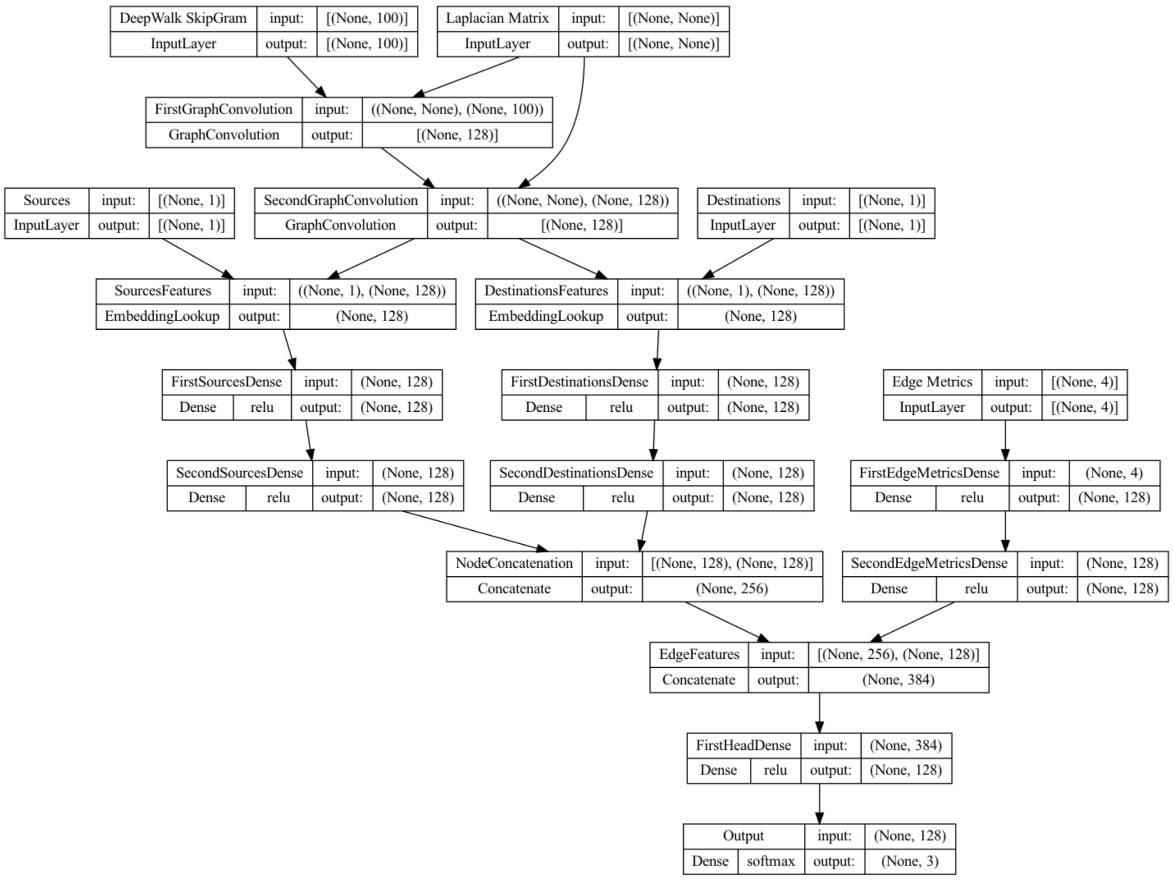
- Tendril starting from the root node 83847 (degree 4), and containing 4 nodes, with a maximal depth of 4, which are 1130678, 630890, 233106 and 12275. Its nodes have a single node type, which is Neural_Networks.
- Tendril starting from the root node 1140543 (degree 5), and containing 3 nodes, with a maximal depth of 3, which are 120817, 1109873 and 163235. Its nodes have a single node type, which is Neural_Networks.
- Tendril starting from the root node 683404 (degree 4), and containing 3 nodes, with a maximal depth of 3, which are 683360, 522338 and 1132864. Its nodes have a single node type, which is Probabilistic_Methods.
- Tendril starting from the root node 20534 (degree 10 and node type Reinforcement_Learning), and containing 3 nodes, with a maximal depth of 3, which are 13972 (node type Reinforcement_Learning), 1126050 (node type Reinforcement_Learning) and 93318 (node type Neural_Networks). Its nodes have 2 node types, which are Reinforcement_Learning (2 nodes, 0.07%) and Neural_Networks.
- Tendril starting from the root node 66751 (degree 5), and containing 3 nodes, with a maximal depth of 3, which are 1138043, 77108 and 77112. Its nodes have a single node type, which is Theory.
- Tendril starting from the root node 3231 (degree 36 and node type Theory), and containing 2 nodes, with a maximal depth of 2, which are 1113926 (node type Neural_Networks) and 250566 (node type Case_Based). Its nodes have 2 node types, which are Neural_Networks and Case_Based.

And other 218 tendrils.

Appendix D

Visualization of Kipf GCN Models

In the current section, we show visualization based on Keras dot model visualizations of *Kipf GCN* models for node-label, edge-label and edge prediction. Analogous visualizations are available for all other support TensorFlow/Keras models.



DeepWalk SkipGram | InputLayer | input: [(None, 100)] | output: [(None, 100)]
Laplacian Matrix | InputLayer | input: [(None, None)] | output: [(None, None)]
FirstGraphConvolution | GraphConvolution | input: ((None, None), (None, 100)) | output: [(None, 128)]
Sources | InputLayer | input: [(None, 1)] | output: [(None, 1)]
SecondGraphConvolution | GraphConvolution | input: ((None, None), (None, 128)) | output: [(None, 128)]
Destinations | InputLayer | input: [(None, 1)] | output: [(None, 1)]
SourcesFeatures | EmbeddingLookup | input: ((None, 1), (None, 128)) | output: (None, 128)
DestinationsFeatures | EmbeddingLookup | input: ((None, 1), (None, 128)) | output: (None, 128)
FirstSourcesDense | Dense | relu | input: (None, 128) | output: (None, 128)
FirstDestinationsDense | Dense | relu | input: (None, 128) | output: (None, 128)
Edge Metrics | InputLayer | input: [(None, 4)] | output: [(None, 4)]
SecondSourcesDense | Dense | relu | input: (None, 128) | output: (None, 128)
SecondDestinationsDense | Dense | relu | input: (None, 128) | output: (None, 128)
FirstEdgeMetricsDense | Dense | relu | input: (None, 4) | output: (None, 128)
NodeConcatenation | Concatenate | input: [(None, 128), (None, 128)] | output: (None, 256)
SecondEdgeMetricsDense | Dense | relu | input: (None, 128) | output: (None, 128)
EdgeFeatures | Concatenate | input: [(None, 256), (None, 128)] | output: (None, 384)
FirstHeadDense | Dense | relu | input: (None, 384) | output: (None, 128)
Output | Dense | softmax | input: (None, 128) | output: (None, 3)

Figure D.1: Default Kipf GCN model for multiclass edge-label prediction, using also the edge metrics (Jaccard Coefficient, Adamic-Adar, Preferential Attachment, and Resource Allocation Index).

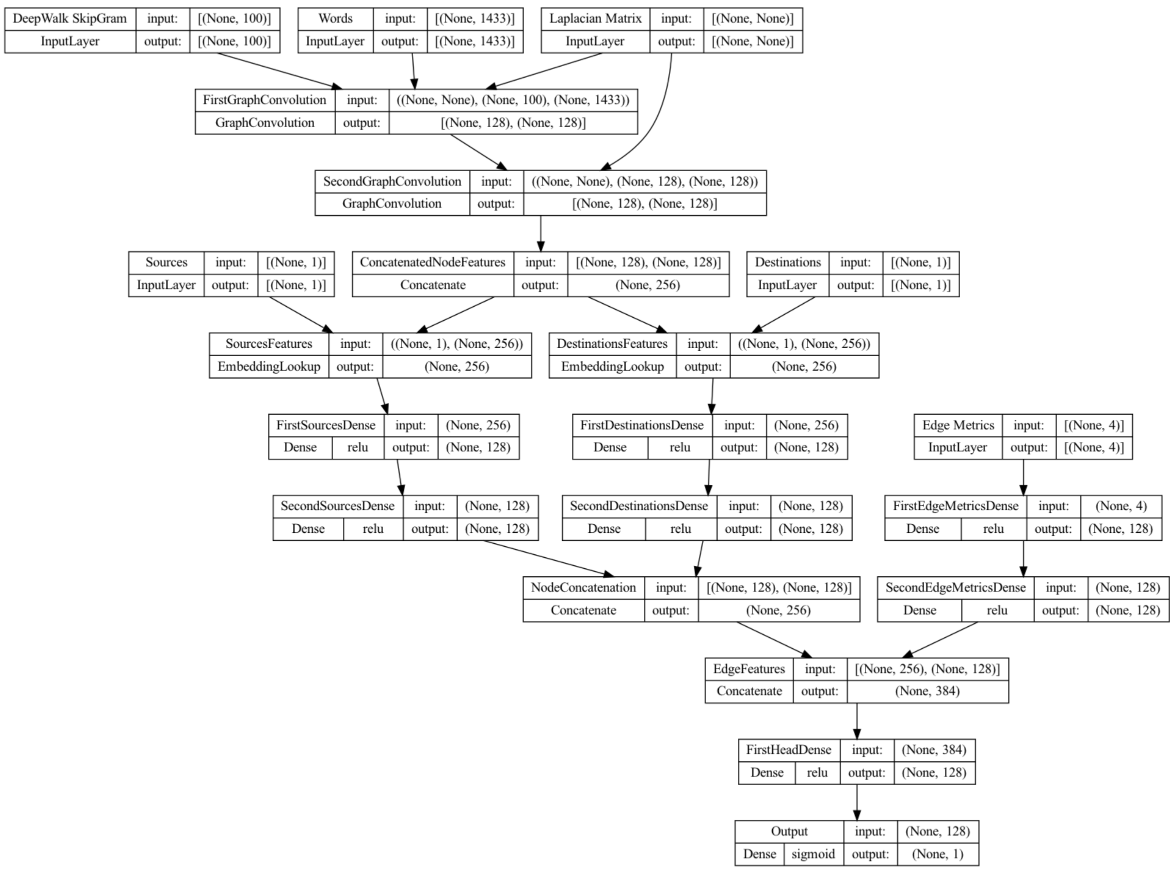

DeepWalk SkipGram | InputLayer | input: [(None, 100)] | output: [(None, 100)]
Words | InputLayer | input: [(None, 1433)] | output: [(None, 1433)]
Laplacian Matrix | InputLayer | input: [(None, None)] | output: [(None, None)]
FirstGraphConvolution | GraphConvolution | input: ((None, None), (None, 100), (None, 1433)) | output: [(None, 128), (None, 128)]
SecondGraphConvolution | GraphConvolution | input: ((None, None), (None, 128), (None, 128)) | output: [(None, 128), (None, 128)]
Sources | InputLayer | input: [(None, 1)] | output: [(None, 1)]
ConcatenatedNodeFeatures | Concatenate | input: [(None, 128), (None, 128)] | output: (None, 256)
Destinations | InputLayer | input: [(None, 1)] | output: [(None, 1)]
SourcesFeatures | EmbeddingLookup | input: ((None, 1), (None, 256)) | output: (None, 256)
DestinationsFeatures | EmbeddingLookup | input: ((None, 1), (None, 256)) | output: (None, 256)
FirstSourcesDense | Dense | relu | input: (None, 256) | output: (None, 128)
FirstDestinationsDense | Dense | relu | input: (None, 256) | output: (None, 128)
Edge Metrics | InputLayer | input: [(None, 4)] | output: [(None, 4)]
SecondSourcesDense | Dense | relu | input: (None, 128) | output: (None, 128)
SecondDestinationsDense | Dense | relu | input: (None, 128) | output: (None, 128)
FirstEdgeMetricsDense | Dense | relu | input: (None, 4) | output: (None, 128)
NodeConcatenation | Concatenate | input: [(None, 128), (None, 128)] | output: (None, 256)
SecondEdgeMetricsDense | Dense | relu | input: (None, 128) | output: (None, 128)
EdgeFeatures | Concatenate | input: [(None, 256), (None, 128)] | output: (None, 384)
FirstHeadDense | Dense | relu | input: (None, 384) | output: (None, 128)
Output | Dense | sigmoid | input: (None, 128) | output: (None, 1)

Figure D.2: Default Kipf GCN model for multi-modal edge prediction, using also the edge metrics (Jaccard Coefficient, Adamic-Adar, Preferential Attachment, and Resource Allocation Index).

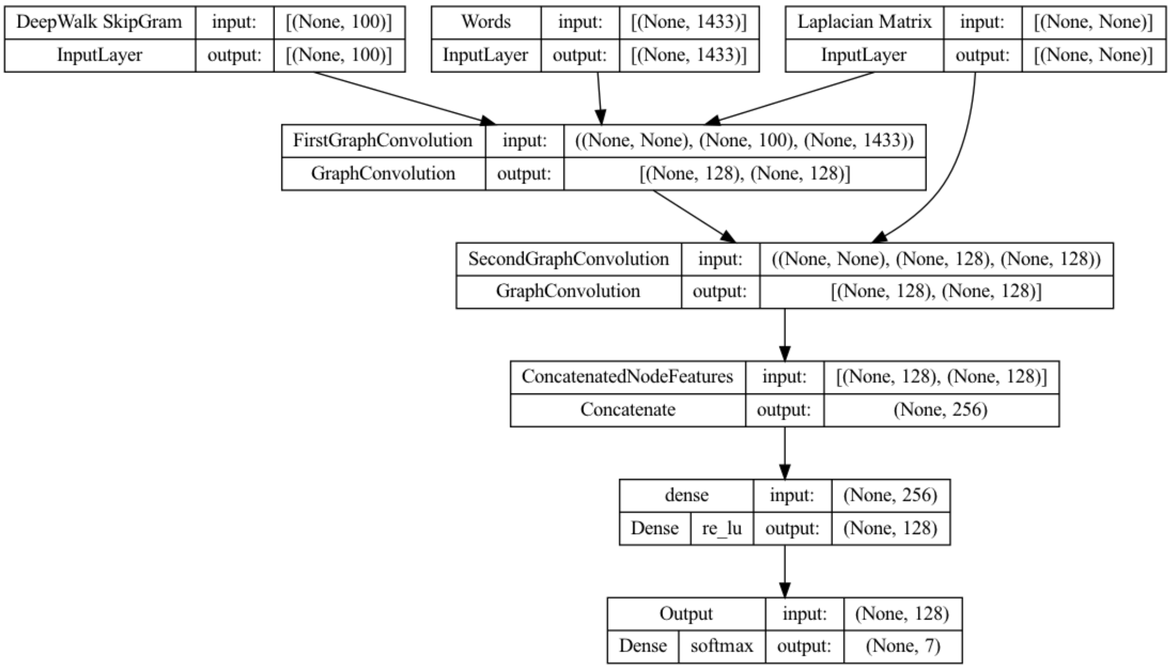

DeepWalk SkipGram | InputLayer | input: [(None, 100)] | output: [(None, 100)]
Words | InputLayer | input: [(None, 1433)] | output: [(None, 1433)]
Laplacian Matrix | InputLayer | input: [(None, None)] | output: [(None, None)]
FirstGraphConvolution | GraphConvolution | input: ((None, None), (None, 100), (None, 1433)) | output: [(None, 128), (None, 128)]
SecondGraphConvolution | GraphConvolution | input: ((None, None), (None, 128), (None, 128)) | output: [(None, 128), (None, 128)]
ConcatenatedNodeFeatures | Concatenate | input: [(None, 128), (None, 128)] | output: (None, 256)
dense | Dense | re_lu | input: (None, 256) | output: (None, 128)
Output | Dense | softmax | input: (None, 128) | output: (None, 7)

Figure D.3: Default Kipf GCN model for multi-modal node-label prediction.

Appendix E

Models and Parameters used for the experimental assessment of GRAPE

In this Appendix we report the details of the machine learning models we used to evaliuate the performance of *GRAPE* in the node embedding task (subsection E.1), the edge-prediction task (subsection E.2), and the node-label prediction task (subsection E.3).

E.1 Evaluated node embedding models

In the following section we report the node embedding model parameters used within the context of the *GRAPE* pipelines for the evaluation of models on node-label and edge prediction tasks.

Node2Vec CBOW

The parameters used for the node embedding model *Node2Vec CBOW* in *GRAPE* evaluation pipelines for the node-label and edge prediction tasks are reported in table E.1.

Table E.1: Node2Vec CBOW model parameters.

Parameter name	Value
Embedding size	100
Epochs	10
Number of negative samples	10
Walk length	128
Iterations	10
Window size	10
Return weight	0.25
Explore weight	4
Max neighbours	100
Learning rate	0.01
Learning rate decay	0.9

Node2Vec GloVe

The parameters used for the node embedding model *Node2Vec GloVe* in *GRAPE* evaluation pipelines for the node-label and edge prediction tasks are reported in table E.2.

Table E.2: Node2Vec GloVe model parameters.

Parameter name	Value
Embedding size	100
Alpha	0.75
Walk length	128
Iterations	10
Window size	10
Return weight	0.25
Explore weight	4
Max neighbours	100
Epochs	500
Learning rate	0.01
Learning rate decay	0.9

Node2Vec SkipGram

The parameters used for the node embedding model *Node2Vec SkipGram* in *GRAPE* evaluation pipelines for the node-label and edge prediction tasks are reported in table E.3.

Table E.3: Node2Vec SkipGram model parameters.

Parameter name	Value
Embedding size	100
Epochs	10
Number of negative samples	10
Walk length	128
Iterations	10
Window size	10
Return weight	0.25
Explore weight	4
Max neighbours	100
Learning rate	0.01
Learning rate decay	0.9

DeepWalk CBOW

The parameters used for the node embedding model *DeepWalk CBOW* in *GRAPE* evaluation pipelines for the node-label and edge prediction tasks are reported in table E.4.

Table E.4: DeepWalk CBOW model parameters.

Parameter name	Value
Embedding size	100
Epochs	10
Number of negative samples	10
Walk length	128
Iterations	10
Window size	10
Max neighbours	100
Learning rate	0.01
Learning rate decay	0.9

DeepWalk GloVe

The parameters used for the node embedding model *DeepWalk GloVe* in *GRAPE* evaluation pipelines for the node-label and edge prediction tasks are reported in table E.5.

Table E.5: DeepWalk GloVe model parameters.

Parameter name	Value
Embedding size	100
Alpha	0.75
Walk length	128
Iterations	10
Window size	10
Max neighbours	100
Epochs	500
Learning rate	0.01
Learning rate decay	0.9

DeepWalk SkipGram

The parameters used for the node embedding model *DeepWalk SkipGram* in *GRAPE* evaluation pipelines for the node-label and edge prediction tasks are reported in table E.6.

Table E.6: DeepWalk SkipGram model parameters.

Parameter name	Value
Embedding size	100
Epochs	10
Number of negative samples	10
Walk length	128
Iterations	10
Window size	10
Max neighbours	100
Learning rate	0.01
Learning rate decay	0.9

First Order LINE

The parameters used for the node embedding model *First Order LINE* in *GRAPE* evaluation pipelines for the node-label and edge prediction tasks are reported in table E.7.

Table E.7: First Order LINE model parameters.

Parameter name	Value
Embedding size	100
Epochs	500
Batch size	1024
Optimizer	nadam
Early stopping min delta	0.001
Early stopping patience	10
Learning rate plateau min delta	0.001
Learning rate plateau patience	5
Negative samples rate	0.5

Second Order LINE

The parameters used for the node embedding model *Second Order LINE* in *GRAPE* evaluation pipelines for the node-label and edge prediction tasks are reported in table E.8.

Table E.8: Second Order LINE model parameters.

Parameter name	Value
Embedding size	100
Epochs	500
Batch size	1024
Optimizer	nadam
Early stopping min delta	0.001
Early stopping patience	10
Learning rate plateau min delta	0.001
Learning rate plateau patience	5
Negative samples rate	0.5

NMFADMM

The parameters used for the node embedding model *NMFADMM* in *GRAPE* evaluation pipelines for the node-label and edge prediction tasks are reported in table E.9.

Table E.9: NMFADMM model parameters.

Parameter name	Value
Embedding size	128
Iterations	100
Rho	1

RandNE

The parameters used for the node embedding model *RandNE* in *GRAPE* evaluation pipelines for the node-label and edge prediction tasks are reported in table E.10.

Table E.10: RandNE model parameters.

Parameter name	Value
Embedding size	100
Alphas	(0.5, 0.5)

GraRep

The parameters used for the node embedding model *GraRep* in *GRAPE* evaluation pipelines for the node-label and edge prediction tasks are reported in table E.11.

Table E.11: GraRep model parameters.

Parameter name	Value
Embedding size	128
Iteration	10
Order	5

DeepWalk Walklets SkipGram

The parameters used for the node embedding model *Walklets SkipGram* in *GRAPE* evaluation pipelines for the node-label and edge prediction tasks are reported in table E.12.

Table E.12: DeepWalk Walklets SkipGram model parameters.

Parameter name	Value
Embedding size	100
Epochs	10
Number of negative samples	10
Walk length	128
Iterations	10
Window size	10
Max neighbours	100
Learning rate	0.01
Learning rate decay	0.9

NetMF

The parameters used for the node embedding model *NetMF* in *GRAPE* evaluation pipelines for the node-label and edge prediction tasks are reported in table E.13.

Table E.13: NetMF model parameters.

Parameter name	Value
Embedding size	100
Walk length	128
Iterations	10
Window size	10
Max neighbours	100

GLEE

The parameters used for the node embedding model *GLEE* in *GRAPE* evaluation pipelines for the node-label and edge prediction tasks are reported in table E.14.

Table E.14: GLEE model parameters.

Parameter name	Value
Embedding size	100

HOPE

The parameters used for the node embedding model *HOPE* in *GRAPE* evaluation pipelines for the node-label and edge prediction tasks are reported in table E.15.

Table E.15: HOPE model parameters.

Parameter name	Value
Embedding size	100
Metric	Neighbours Intersection size
Root node name	None

Role2Vec

The parameters used for the node embedding model *Role2Vec* in *GRAPE* evaluation pipelines for the node-label and edge prediction tasks are reported in table E.16.

Table E.16: Role2Vec model parameters.

Parameter name	Value
Embedding size	100
Walk number	10
Walk length	80
Window size	5
Epochs	10
Learning rate	0.05
Min count	1
Down sampling	0.0001
Weisfeiler lehman hashing iterations	2
Erase base features	False

E.2 Evaluated edge prediction models

Decision Tree Classifier

The parameters used for the edge prediction model *Decision Tree Classifier* in *GRAPE* evaluation pipelines for the node-label and edge prediction tasks are reported in table E.17.

Table E.17: Parameters of Decision Tree Classifier model for edge prediction.

Parameter name	Value
Edge embedding method	Hadamard
Training unbalance rate	1
Criterion	gini
Splitter	best
Max depth	10
Min samples split	2
Min samples leaf	1
Min weight fraction leaf	0
Max features	None
Max leaf nodes	None
Min impurity decrease	0
CCP alpha	0

Perceptron

The parameters used for the edge prediction model *Perceptron* in *GRAPE* evaluation pipelines for the node-label and edge prediction tasks are reported in table E.18.

Table E.18: Parameters of Perceptron model for edge prediction.

Parameter name	Value
Edge embeddings	Hadamard
Cooccurrence iterations	100
Cooccurrence window size	10
Number of epochs	100
Number of edges per mini batch	256
Learning rate	0.001
First order decay factor	0.9
Second order decay factor	0.999

E.3 Evaluated node-label prediction models

Decision Tree Classifier

The parameters used for the node label prediction model *Decision Tree Classifier* in *GRAPE* evaluation pipelines for the node-label and edge prediction tasks are reported in table E.19.

Table E.19: Parameters of Decision Tree Classifier model for node-label prediction.

Parameter name	Value
Criterion	gini
Splitter	best
Max depth	10
Min samples split	2
Min samples leaf	1
Min weight fraction leaf	0
Max features	None
Max leaf nodes	None
Min impurity decrease	0
Class weight	balanced
CCP alpha	0

Random Forest Classifier

The parameters used for the node label prediction model *Random Forest Classifier* in *GRAPE* evaluation pipelines for the node-label and edge prediction tasks are reported in table E.20.

Table E.20: Parameters of Random Forest Classifier model for node-label prediction.

Parameter name	Value
N estimators	1000
Criterion	gini
Max depth	10
Min samples split	2
Min samples leaf	1
Min weight fraction leaf	0
Max features	sqrt
Max leaf nodes	None
Min impurity decrease	0
Bootstrap	True
Oob score	False
Warm start	False
Class weight	balanced
CCP alpha	0
Max samples	None

Appendix F

Edge and node label prediction performance

In this section, we report the full performance results, estimated with different metrics, using the *GRAPE* pipelines for the evaluation of edge and node-label prediction tasks.

F.1 Edge prediction performance

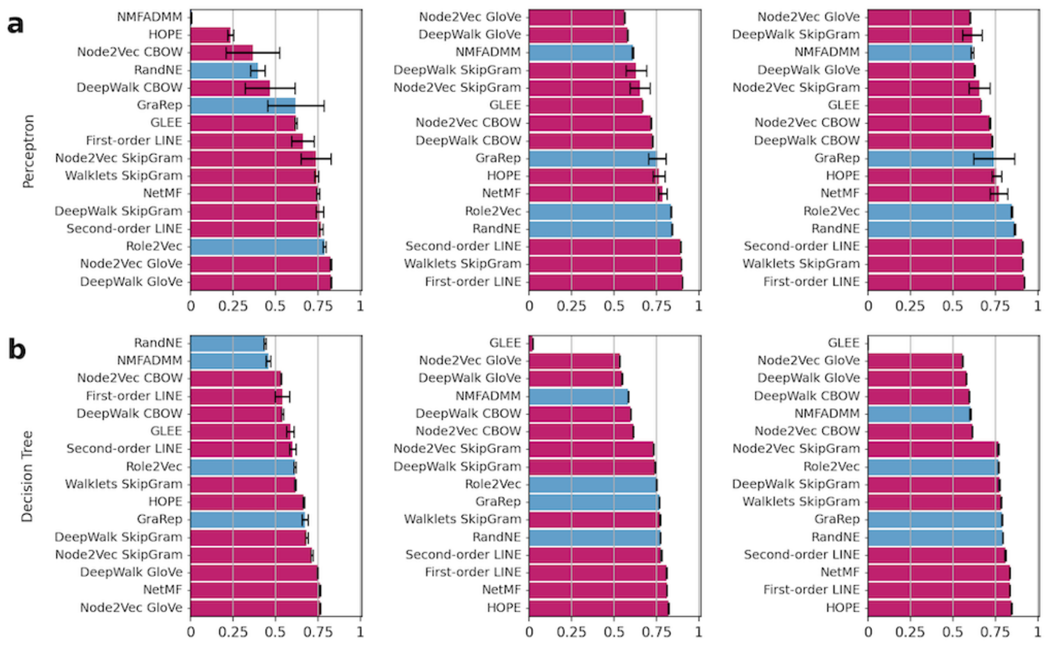


Figure F.1: Average f1 score of edge prediction models trained on embedding methods. Results are averaged across ten holdouts. Embedding models are sorted for each task; methods implemented in *GRAPE* are in purple, while methods integrated through the *GRAPE* interface are in cyan. We show perceptron and decision tree in row **a** and **b**, respectively. From left to right, Human Phenotype Ontology, STRING Homo sapiens and STRING Mus musculus.

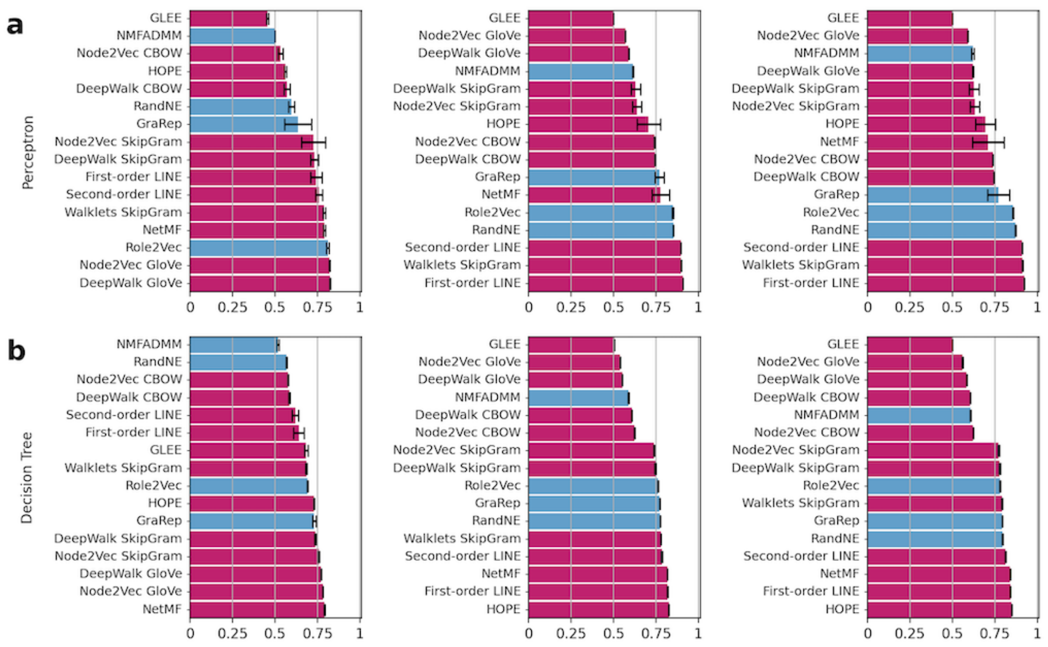


Figure F.2: Average balanced accuracy of edge prediction models trained on embedding methods. Results are averaged across ten holdouts. Embedding models are sorted for each task; methods implemented in *GRAPE* are in purple, while methods integrated through the *GRAPE* interface are in cyan. We show perceptron and decision tree in row **a** and **b**, respectively. From left to right, Human Phenotype Ontology, STRING Homo sapiens and STRING Mus musculus.

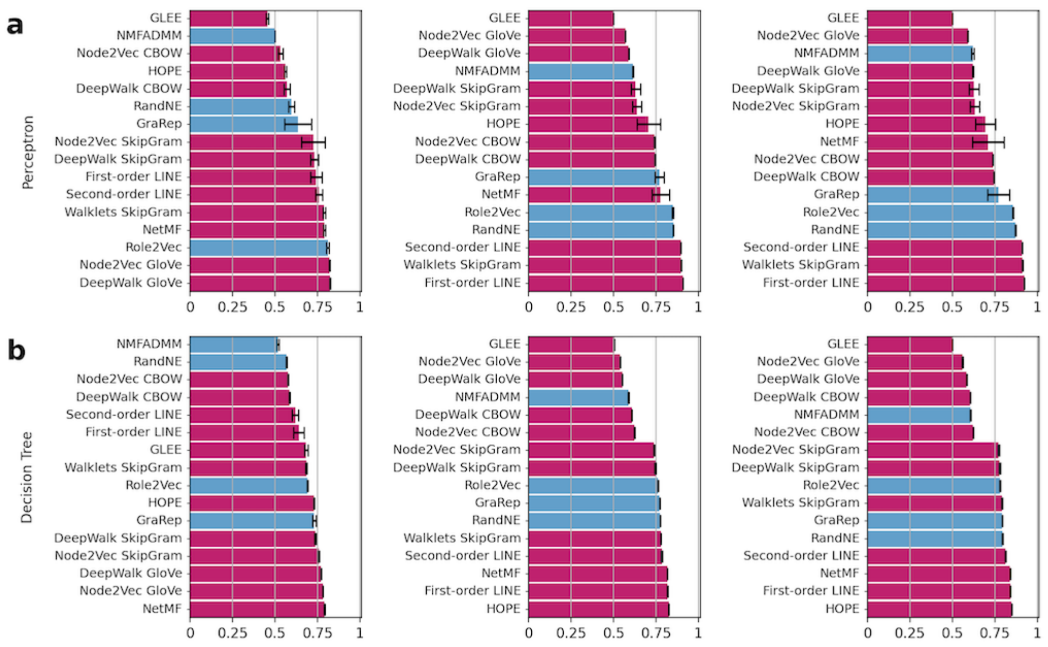


Figure F.3: Average accuracy of edge prediction models trained on embedding methods. Results are averaged across ten holdouts. Embedding models are sorted for each task; methods implemented in *GRAPE* are in purple, while methods integrated through the *GRAPE* interface are in cyan. We show perceptron and decision tree in row **a** and **b**, respectively. From left to right, Human Phenotype Ontology, STRING Homo sapiens and STRING Mus musculus.

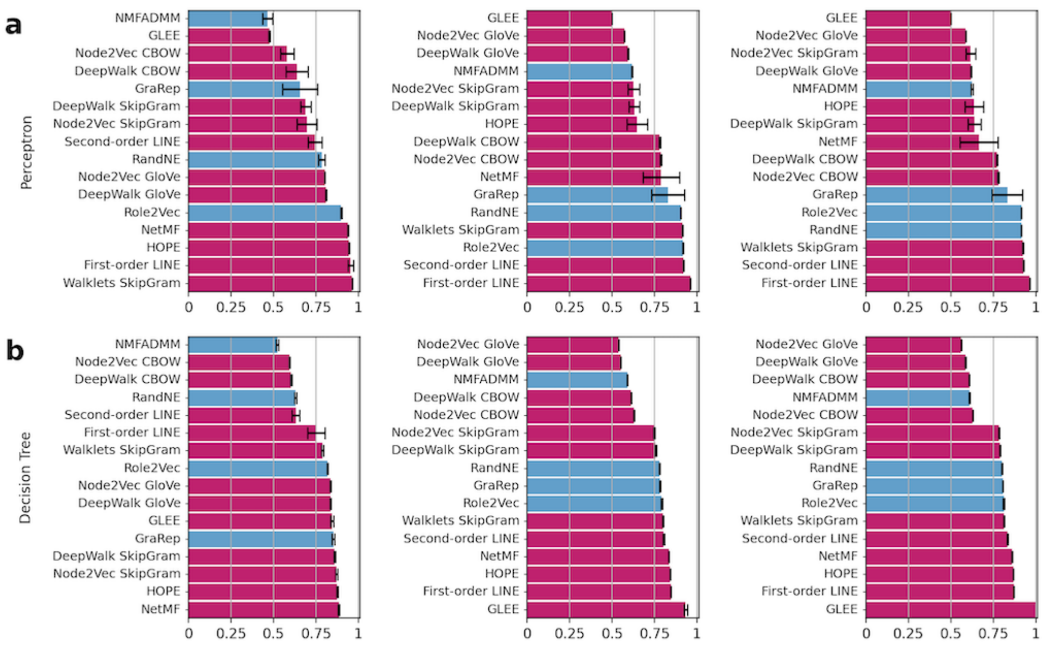


Figure F.4: Average precision of edge prediction models trained on embedding methods. Results are averaged across ten holdouts. Embedding models are sorted for each task; methods implemented in *GRAPE* are in purple, while methods integrated through the *GRAPE* interface are in cyan. We show perceptron and decision tree in row **a** and **b**, respectively. From left to right, Human Phenotype Ontology, STRING Homo sapiens and STRING Mus musculus.

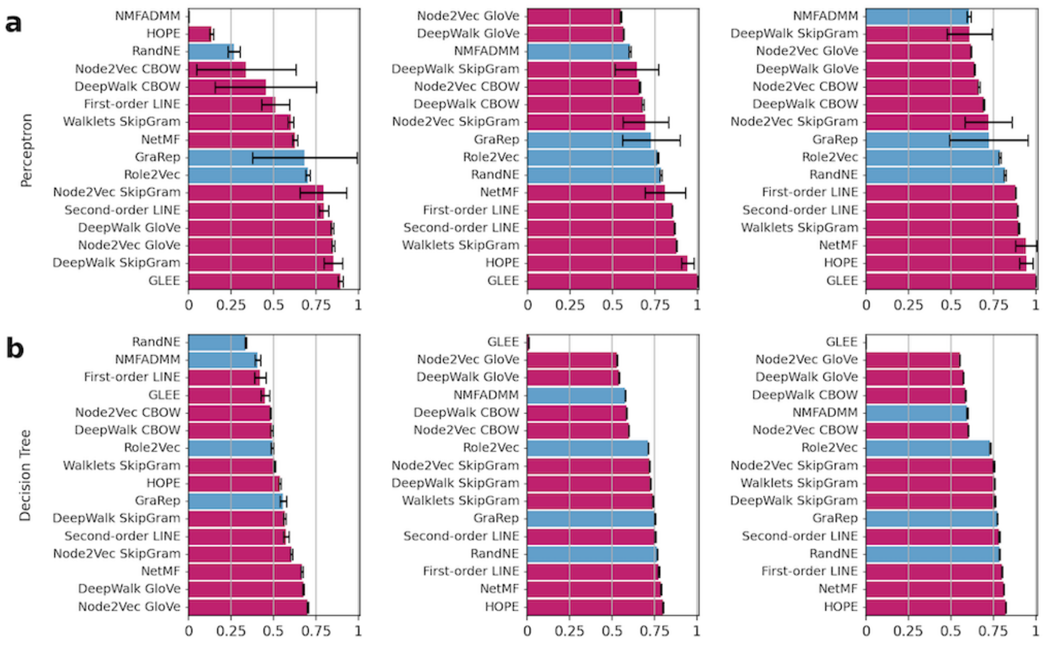


Figure F.5: Average recall of edge prediction models trained on embedding methods. Results are averaged across ten holdouts. Embedding models are sorted for each task; methods implemented in *GRAPE* are in purple, while methods integrated through the *GRAPE* interface are in cyan. We show perceptron and decision tree in row **a** and **b**, respectively. From left to right, Human Phenotype Ontology, STRING Homo sapiens and STRING Mus musculus.

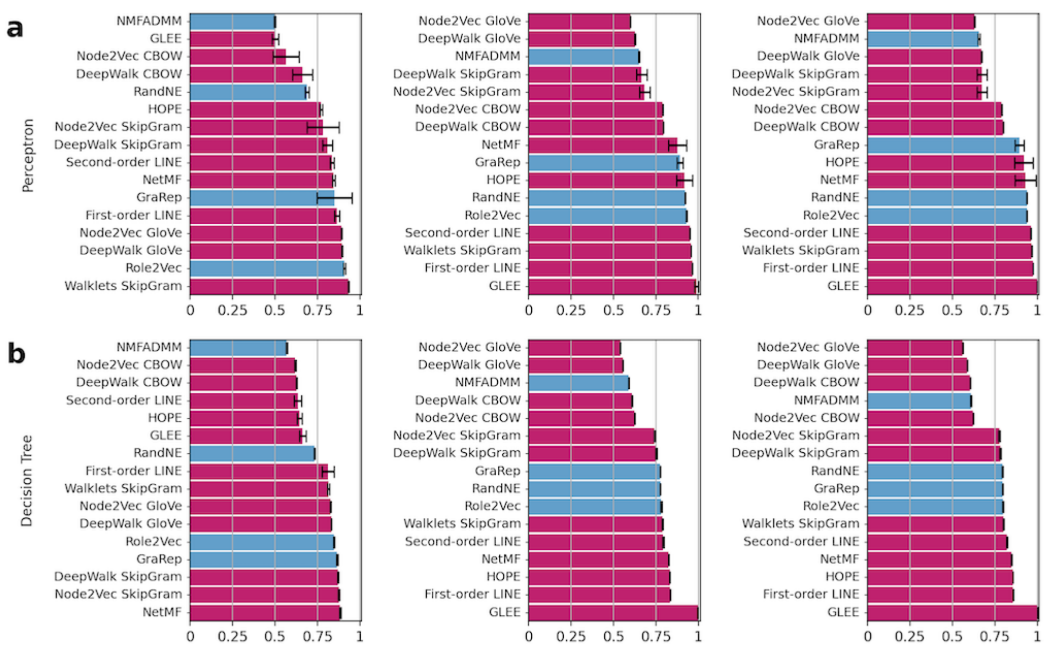


Figure F.6: Average auroc of edge prediction models trained on embedding methods. Results are averaged across ten holdouts. Embedding models are sorted for each task; methods implemented in *GRAPE* are in purple, while methods integrated through the *GRAPE* interface are in cyan. We show perceptron and decision tree in row **a** and **b**, respectively. From left to right, Human Phenotype Ontology, STRING Homo sapiens and STRING Mus musculus.

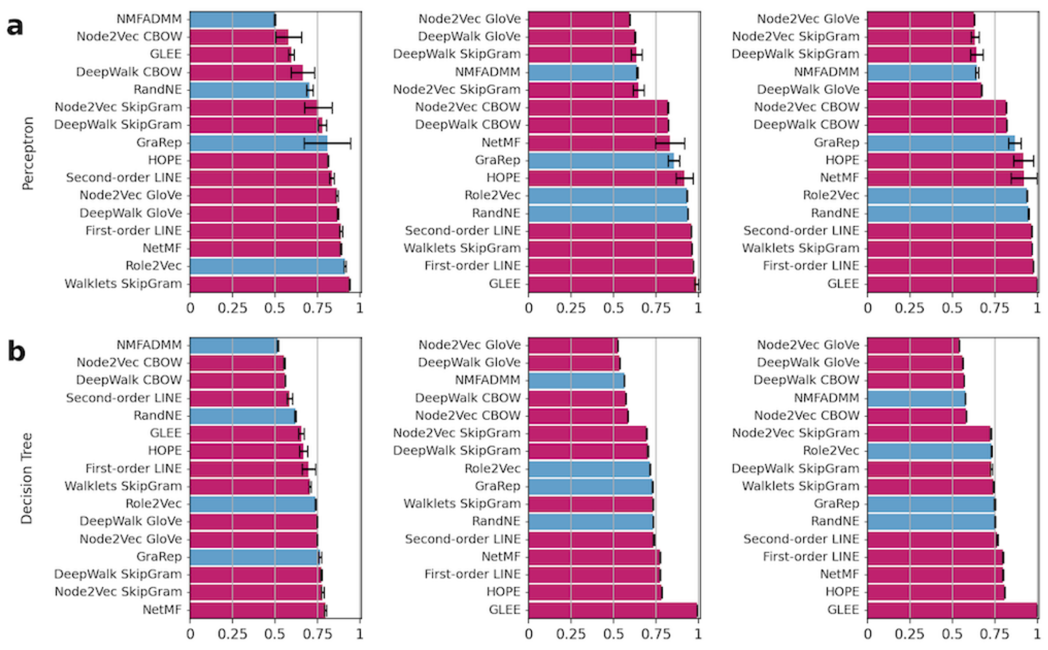


Figure F.7: Average auprc of edge prediction models trained on embedding methods. Results are averaged across ten holdouts. Embedding models are sorted for each task; methods implemented in *GRAPE* are in purple, while methods integrated through the *GRAPE* interface are in cyan. We show perceptron and decision tree in row **a** and **b**, respectively. From left to right, Human Phenotype Ontology, STRING Homo sapiens and STRING Mus musculus.

Table F.1: Decision Tree edge prediction performance in test evaluation on Human Phenotype Ontology. The rows are sorted by balanced accuracy.

	Balanced Accuracy	F1 Score	AUPRC	Accuracy	Recall	AUROC	Precision
NMFADMM	.52 ± .006	.46 ± .013	.52 ± .005	.52 ± .006	.41 ± .016	.57 ± .004	.52 ± .007
RandNE	.57 ± .003	.44 ± .005	.62 ± .004	.57 ± .003	.34 ± .005	.73 ± .004	.63 ± .006
Node2Vec CBOW	.58 ± .003	.53 ± .003	.55 ± .004	.58 ± .003	.48 ± .003	.62 ± .004	.60 ± .004
DeepWalk CBOW	.59 ± .004	.54 ± .006	.56 ± .002	.59 ± .004	.49 ± .008	.63 ± .003	.61 ± .004
Second-order LINE	.62 ± .019	.60 ± .018	.59 ± .017	.62 ± .019	.58 ± .015	.64 ± .022	.63 ± .023
First-order LINE	.64 ± .031	.54 ± .041	.70 ± .039	.64 ± .031	.42 ± .035	.81 ± .035	.75 ± .051
GLEE	.68 ± .01	.59 ± .022	.66 ± .015	.68 ± .01	.45 ± .026	.66 ± .018	.84 ± .009
Walklets	.69 ± .004	.62 ± .005	.71 ± .006	.69 ± .004	.51 ± .005	.82 ± .006	.79 ± .007
Role2Vec	.69 ± .004	.62 ± .006	.74 ± .004	.69 ± .004	.49 ± .008	.85 ± .003	.82 ± .004
HOPE	.73 ± .004	.67 ± .005	.67 ± .023	.73 ± .004	.54 ± .005	.64 ± .015	.88 ± .004
GraRep	.73 ± .01	.67 ± .016	.77 ± .01	.73 ± .01	.56 ± .02	.87 ± .004	.85 ± .006
DeepWalk SkipGram	.74 ± .004	.68 ± .006	.77 ± .004	.74 ± .004	.57 ± .007	.87 ± .003	.86 ± .004
Node2Vec SkipGram	.76 ± .005	.72 ± .006	.78 ± .007	.76 ± .005	.61 ± .006	.88 ± .005	.87 ± .006
DeepWalk GloVe	.77 ± .002	.75 ± .003	.75 ± .002	.77 ± .002	.68 ± .004	.83 ± .002	.84 ± .002
Node2Vec GloVe	.78 ± .002	.76 ± .003	.75 ± .002	.78 ± .002	.70 ± .005	.83 ± .002	.84 ± .002
NetMF	.79 ± .003	.76 ± .004	.80 ± .005	.79 ± .003	.67 ± .006	.88 ± .004	.89 ± .004

Table F.2: Perceptron edge prediction performance in test evaluation on Human Phenotype Ontology. The rows are sorted by balanced accuracy.

	Balanced Accuracy	F1 Score	AUPRC	Accuracy	Recall	AUROC	Precision
GLEE	.46 ± .007	.62 ± .007	.60 ± .017	.46 ± .007	.89 ± .014	.50 ± .019	.48 ± .004
NMFADMM	.50 ± .0002	.01 ± .0004	.50 ± .004	.50 ± .0002	.00 ± .0002	.50 ± .004	.46 ± .03
Node2Vec CBOW	.53 ± .014	.37 ± .157	.58 ± .077	.53 ± .014	.34 ± .292	.57 ± .077	.58 ± .041
HOPE	.56 ± .006	.24 ± .018	.81 ± .004	.56 ± .006	.13 ± .012	.77 ± .007	.94 ± .003
DeepWalk CBOW	.57 ± .018	.47 ± .147	.66 ± .069	.57 ± .018	.45 ± .298	.66 ± .059	.64 ± .064
RandNE	.60 ± .017	.40 ± .043	.70 ± .018	.60 ± .017	.27 ± .035	.69 ± .012	.79 ± .019
GraRep	.63 ± .08	.62 ± .165	.81 ± .137	.63 ± .08	.68 ± .307	.85 ± .103	.66 ± .103
Node2Vec SkipGram	.73 ± .07	.74 ± .089	.76 ± .08	.73 ± .07	.79 ± .136	.78 ± .094	.70 ± .058
DeepWalk SkipGram	.73 ± .025	.76 ± .021	.78 ± .025	.73 ± .025	.85 ± .054	.81 ± .029	.69 ± .03
First-order LINE	.74 ± .035	.66 ± .066	.89 ± .009	.74 ± .035	.51 ± .083	.86 ± .015	.95 ± .017
Second-order LINE	.76 ± .022	.77 ± .011	.84 ± .013	.76 ± .022	.80 ± .027	.84 ± .012	.75 ± .041
Walklets	.79 ± .007	.74 ± .012	.94 ± .002	.79 ± .007	.60 ± .016	.93 ± .002	.96 ± .002
NetMF	.79 ± .006	.75 ± .01	.89 ± .005	.79 ± .006	.63 ± .014	.85 ± .008	.94 ± .002
Role2Vec	.81 ± .006	.79 ± .008	.91 ± .005	.81 ± .006	.70 ± .013	.91 ± .006	.90 ± .005
Node2Vec GloVe	.82 ± .003	.83 ± .004	.87 ± .005	.82 ± .003	.85 ± .007	.89 ± .003	.80 ± .002
DeepWalk GloVe	.82 ± .003	.83 ± .003	.87 ± .006	.82 ± .003	.85 ± .006	.90 ± .003	.81 ± .003

Table F.3: Decision Tree edge prediction performance in test evaluation on Mus Musculus. The rows are sorted by balanced accuracy.

	Balanced Accuracy	F1 Score	AUPRC	Accuracy	Recall	AUROC	Precision
GLEE	.50 ± .	.00 ± .0001	1.00 ± .	.50 ± .	.00 ± .	1.00 ± .	1.00 ± .
Node2Vec GloVe	.56 ± .002	.56 ± .002	.54 ± .002	.56 ± .002	.55 ± .002	.56 ± .002	.56 ± .002
DeepWalk GloVe	.58 ± .002	.58 ± .001	.56 ± .001	.58 ± .002	.57 ± .002	.59 ± .002	.59 ± .002
DeepWalk CBOW	.61 ± .001	.60 ± .001	.57 ± .001	.61 ± .001	.59 ± .002	.60 ± .002	.61 ± .002
NMFADMM	.61 ± .003	.60 ± .004	.58 ± .002	.61 ± .003	.60 ± .004	.61 ± .003	.61 ± .003
Node2Vec CBOW	.62 ± .002	.62 ± .002	.58 ± .001	.62 ± .002	.60 ± .002	.62 ± .002	.63 ± .002
Node2Vec SkipGram	.77 ± .004	.77 ± .004	.73 ± .004	.77 ± .004	.75 ± .005	.78 ± .004	.78 ± .004
DeepWalk SkipGram	.78 ± .004	.77 ± .004	.73 ± .005	.78 ± .004	.76 ± .005	.78 ± .004	.79 ± .004
Role2Vec	.78 ± .003	.77 ± .003	.73 ± .003	.78 ± .003	.73 ± .004	.80 ± .003	.81 ± .003
Walklets	.79 ± .003	.78 ± .003	.74 ± .003	.79 ± .003	.76 ± .003	.80 ± .003	.81 ± .003
GraRep	.79 ± .002	.79 ± .002	.75 ± .004	.79 ± .002	.77 ± .003	.80 ± .003	.81 ± .002
RandNE	.80 ± .003	.79 ± .003	.75 ± .003	.80 ± .003	.79 ± .003	.80 ± .003	.80 ± .003
Second-order LINE	.81 ± .003	.81 ± .003	.76 ± .004	.81 ± .003	.79 ± .003	.82 ± .003	.83 ± .003
NetMF	.84 ± .001	.84 ± .001	.80 ± .003	.84 ± .001	.81 ± .001	.85 ± .002	.86 ± .002
First-order LINE	.84 ± .002	.84 ± .002	.80 ± .002	.84 ± .002	.80 ± .003	.86 ± .001	.87 ± .001
HOPE	.85 ± .002	.85 ± .002	.81 ± .001	.85 ± .002	.82 ± .002	.86 ± .001	.87 ± .002

Table F.4: Perceptron edge prediction performance in test evaluation on Mus Musculus. The rows are sorted by balanced accuracy.

	Balanced Accuracy	F1 Score	AUPRC	Accuracy	Recall	AUROC	Precision
GLEE	.50 ± .	.67 ± .	1.00 ± .	.50 ± .	1.00 ± .	1.00 ± .0001	.50 ± .
Node2Vec GloVe	.59 ± .002	.60 ± .002	.63 ± .002	.59 ± .002	.62 ± .004	.63 ± .003	.59 ± .002
NMFADMM	.62 ± .006	.61 ± .008	.65 ± .008	.62 ± .006	.61 ± .012	.66 ± .006	.62 ± .005
DeepWalk GloVe	.62 ± .003	.63 ± .003	.67 ± .002	.62 ± .003	.64 ± .004	.67 ± .003	.62 ± .003
DeepWalk SkipGram	.63 ± .03	.61 ± .059	.64 ± .036	.63 ± .03	.61 ± .132	.67 ± .029	.64 ± .039
Node2Vec SkipGram	.63 ± .027	.66 ± .062	.63 ± .024	.63 ± .027	.72 ± .139	.67 ± .03	.62 ± .028
HOPE	.70 ± .059	.76 ± .028	.92 ± .059	.70 ± .059	.94 ± .041	.92 ± .054	.64 ± .055
NetMF	.71 ± .092	.77 ± .05	.92 ± .077	.71 ± .092	.94 ± .061	.93 ± .063	.67 ± .112
Node2Vec CBOW	.74 ± .002	.72 ± .003	.82 ± .002	.74 ± .002	.66 ± .007	.79 ± .002	.78 ± .003
DeepWalk CBOW	.74 ± .002	.73 ± .002	.82 ± .002	.74 ± .002	.69 ± .005	.80 ± .002	.77 ± .003
GraRep	.77 ± .065	.74 ± .121	.87 ± .036	.77 ± .065	.72 ± .232	.90 ± .027	.83 ± .09
Role2Vec	.86 ± .003	.85 ± .004	.94 ± .002	.86 ± .003	.79 ± .006	.94 ± .002	.91 ± .002
RandNE	.87 ± .003	.86 ± .004	.95 ± .002	.87 ± .003	.82 ± .006	.94 ± .002	.92 ± .002
Second-order LINE	.91 ± .002	.91 ± .002	.97 ± .002	.91 ± .002	.89 ± .003	.96 ± .002	.93 ± .002
Walklets	.91 ± .002	.91 ± .002	.97 ± .001	.91 ± .002	.90 ± .004	.97 ± .001	.93 ± .002
First-order LINE	.92 ± .001	.92 ± .001	.98 ± .0005	.92 ± .001	.88 ± .002	.97 ± .0006	.96 ± .002

Table F.5: Decision Tree edge prediction performance in test evaluation on Homo Sapiens. The rows are sorted by balanced accuracy.

	Balanced Accuracy	F1 Score	AUPRC	Accuracy	Recall	AUROC	Precision
GLEE	.51 ± .0006	.02 ± .003	.99 ± .001	.51 ± .0006	.01 ± .001	1.00 ± .0006	.93 ± .011
Node2Vec GloVe	.54 ± .001	.53 ± .001	.52 ± .001	.54 ± .001	.53 ± .002	.54 ± .002	.54 ± .001
DeepWalk GloVe	.55 ± .001	.55 ± .002	.54 ± .001	.55 ± .001	.54 ± .002	.55 ± .001	.55 ± .001
NMFADMM	.59 ± .002	.58 ± .002	.56 ± .002	.59 ± .002	.58 ± .002	.59 ± .003	.59 ± .003
DeepWalk CBOW	.61 ± .002	.60 ± .002	.57 ± .002	.61 ± .002	.58 ± .003	.61 ± .002	.61 ± .002
Node2Vec CBOW	.62 ± .002	.61 ± .002	.58 ± .002	.62 ± .002	.60 ± .002	.62 ± .002	.63 ± .002
Node2Vec SkipGram	.74 ± .003	.73 ± .003	.69 ± .003	.74 ± .003	.72 ± .003	.74 ± .003	.75 ± .004
DeepWalk SkipGram	.75 ± .004	.74 ± .004	.70 ± .004	.75 ± .004	.73 ± .004	.75 ± .004	.76 ± .004
Role2Vec	.76 ± .003	.75 ± .003	.71 ± .004	.76 ± .003	.71 ± .003	.78 ± .004	.79 ± .004
GraRep	.77 ± .002	.77 ± .002	.73 ± .003	.77 ± .002	.75 ± .002	.77 ± .002	.78 ± .002
RandNE	.77 ± .002	.77 ± .002	.73 ± .002	.77 ± .002	.77 ± .002	.78 ± .002	.78 ± .002
Walklets	.78 ± .002	.77 ± .003	.73 ± .003	.78 ± .002	.74 ± .003	.79 ± .002	.80 ± .003
Second-order LINE	.78 ± .004	.78 ± .004	.74 ± .005	.78 ± .004	.75 ± .005	.79 ± .004	.80 ± .004
NetMF	.82 ± .001	.81 ± .001	.77 ± .002	.82 ± .001	.79 ± .002	.82 ± .002	.83 ± .002
First-order LINE	.82 ± .003	.81 ± .003	.77 ± .002	.82 ± .003	.77 ± .004	.83 ± .002	.85 ± .002
HOPE	.82 ± .002	.82 ± .002	.78 ± .002	.82 ± .002	.80 ± .002	.83 ± .002	.84 ± .002

Table F.6: Perceptron edge prediction performance in test evaluation on Homo Sapiens. The rows are sorted by balanced accuracy.

	Balanced Accuracy	F1 Score	AUPRC	Accuracy	Recall	AUROC	Precision
GLEE	.50 ± .	.67 ± .	.99 ± .011	.50 ± .	1.00 ± .	.99 ± .011	.50 ± .
Node2Vec GloVe	.57 ± .002	.56 ± .003	.60 ± .002	.57 ± .002	.55 ± .004	.60 ± .002	.57 ± .002
DeepWalk GloVe	.59 ± .002	.58 ± .002	.63 ± .003	.59 ± .002	.57 ± .003	.63 ± .003	.59 ± .002
NMFADMM	.62 ± .003	.61 ± .004	.64 ± .004	.62 ± .003	.60 ± .005	.65 ± .003	.62 ± .003
DeepWalk SkipGram	.63 ± .027	.63 ± .06	.64 ± .032	.63 ± .027	.64 ± .128	.67 ± .031	.63 ± .031
Node2Vec SkipGram	.64 ± .027	.65 ± .058	.65 ± .033	.64 ± .027	.70 ± .134	.68 ± .031	.63 ± .034
HOPE	.71 ± .069	.76 ± .035	.92 ± .05	.71 ± .069	.94 ± .036	.92 ± .048	.65 ± .06
Node2Vec CBOW	.74 ± .002	.72 ± .002	.82 ± .002	.74 ± .002	.66 ± .005	.79 ± .001	.79 ± .004
DeepWalk CBOW	.74 ± .002	.73 ± .003	.82 ± .002	.74 ± .002	.68 ± .006	.79 ± .002	.78 ± .004
GraRep	.77 ± .026	.75 ± .051	.86 ± .034	.77 ± .026	.73 ± .17	.89 ± .019	.83 ± .098
NetMF	.78 ± .052	.79 ± .025	.83 ± .085	.78 ± .052	.81 ± .119	.88 ± .054	.79 ± .107
Role2Vec	.85 ± .003	.83 ± .004	.93 ± .003	.85 ± .003	.77 ± .005	.93 ± .003	.92 ± .002
RandNE	.85 ± .002	.84 ± .003	.94 ± .001	.85 ± .002	.79 ± .005	.92 ± .002	.91 ± .001
Second-order LINE	.90 ± .002	.89 ± .002	.96 ± .001	.90 ± .002	.87 ± .003	.95 ± .002	.92 ± .002
Walklets	.90 ± .001	.90 ± .001	.96 ± .0009	.90 ± .001	.88 ± .002	.96 ± .001	.91 ± .002
First-order LINE	.91 ± .0007	.90 ± .0008	.97 ± .0004	.91 ± .0007	.85 ± .002	.97 ± .0005	.96 ± .002

F.2 Node-label prediction performance

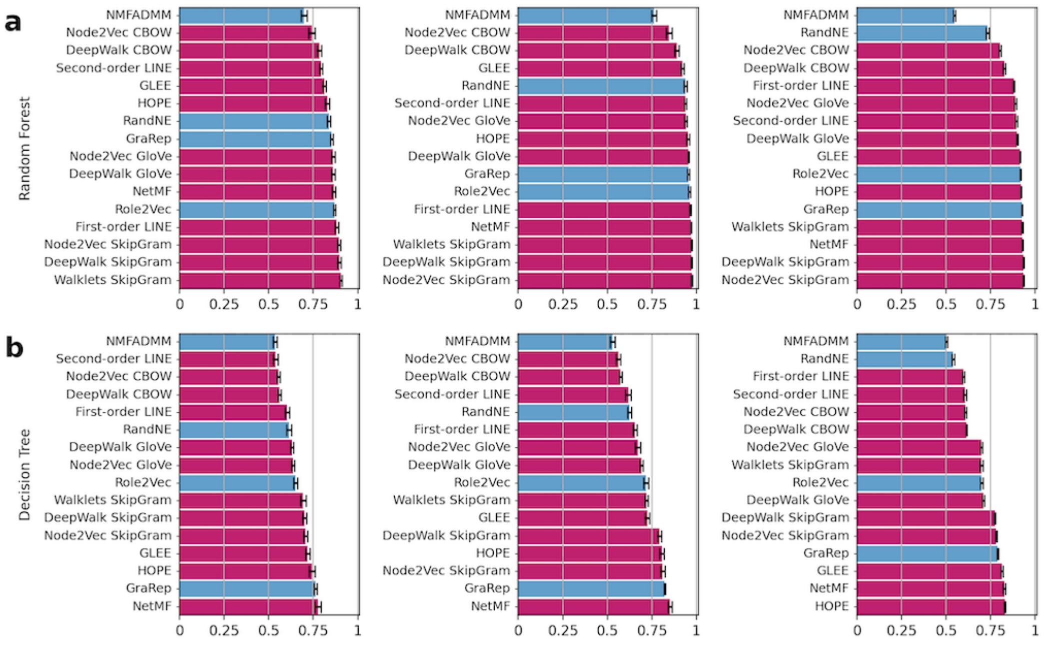


Figure F.8: Average auroc of node label prediction models trained on embedding methods. Results are averaged across ten holdouts. Embedding models are sorted for each task; methods implemented in *GRAPE* are in purple, while methods integrated through the *GRAPE* interface are in cyan. We show random forest and decision tree in row **a** and **b**, respectively. From left to right, CiteSeer, Cora and PubMed Diabetes datasets..

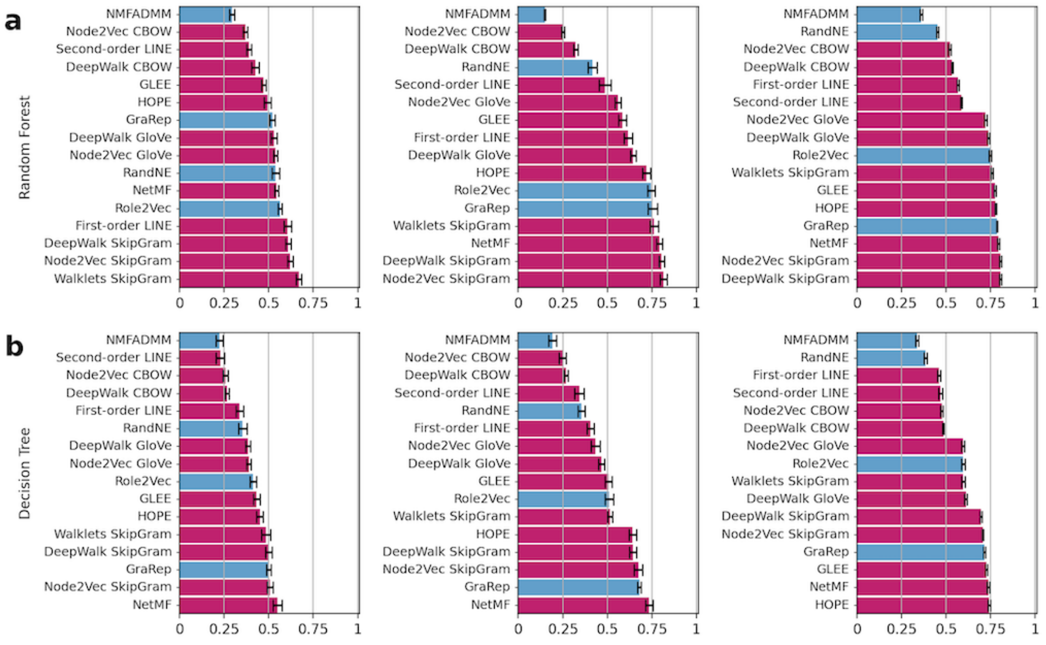


Figure F.9: Average balanced accuracy of node label prediction models trained on embedding methods. Results are averaged across ten holdouts. Embedding models are sorted for each task; methods implemented in *GRAPE* are in purple, while methods integrated through the *GRAPE* interface are in cyan. We show random forest and decision tree in row **a** and **b**, respectively. From left to right, CiteSeer, Cora and PubMed Diabetes datasets..

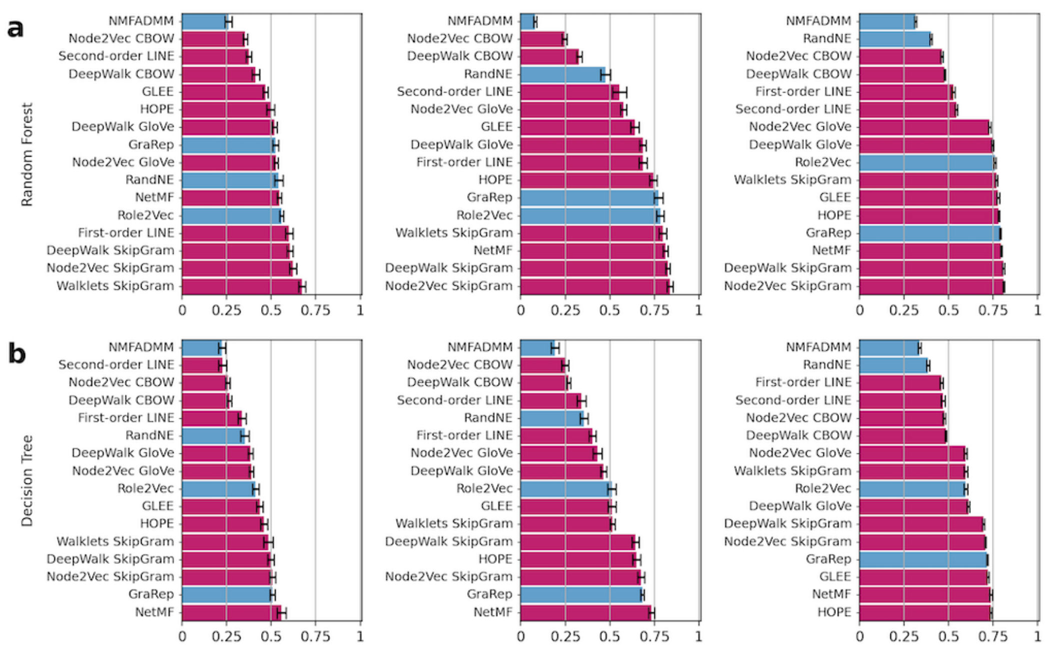


Figure F.10: Average f1 score of node label prediction models trained on embedding methods. Results are averaged across ten holdouts. Embedding models are sorted for each task; methods implemented in *GRAPE* are in purple, while methods integrated through the *GRAPE* interface are in cyan. We show random forest and decision tree in row **a** and **b**, respectively. From left to right, CiteSeer, Cora and PubMed Diabetes datasets..

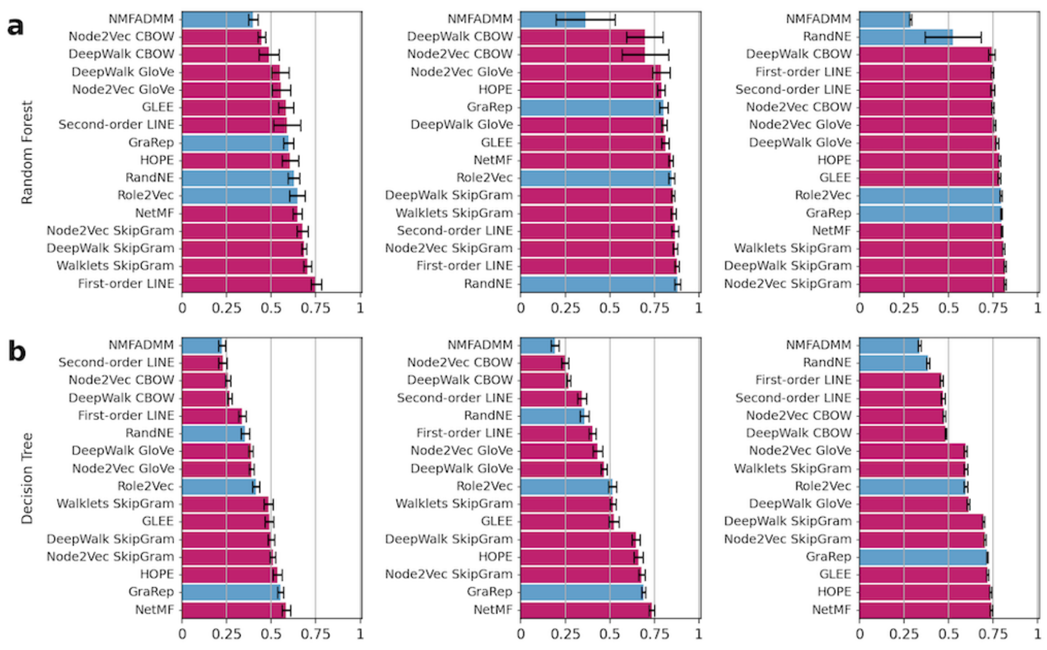


Figure F.11: Average precision of node label prediction models trained on embedding methods. Results are averaged across ten holdouts. Embedding models are sorted for each task; methods implemented in *GRAPE* are in purple, while methods integrated through the *GRAPE* interface are in cyan. We show random forest and decision tree in row **a** and **b**, respectively. From left to right, CiteSeer, Cora and PubMed Diabetes datasets..

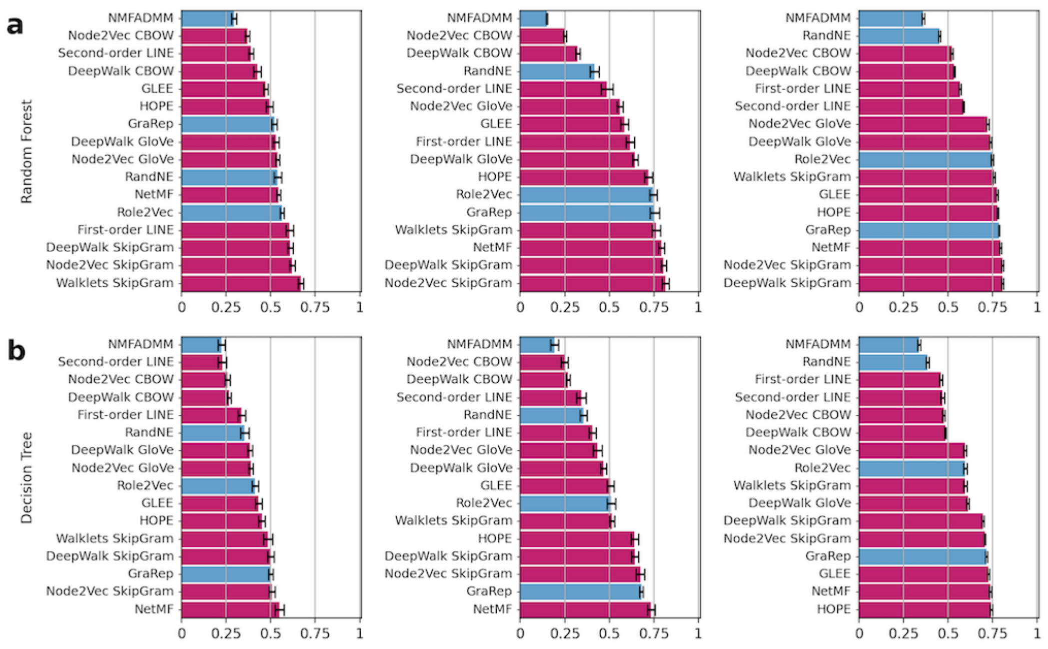


Figure F.12: Average recall of node label prediction models trained on embedding methods. Results are averaged across ten holdouts. Embedding models are sorted for each task; methods implemented in *GRAPE* are in purple, while methods integrated through the *GRAPE* interface are in cyan. We show random forest and decision tree in row **a** and **b**, respectively. From left to right, CiteSeer, Cora and PubMed Diabetes datasets..

Table F.7: Random Forest node-label prediction performance in test evaluation on Cora. The rows are sorted by balanced accuracy.

	AUROC	F1 Score	Balanced Accuracy	Recall	Precision	Accuracy
NMFADMM	.76 ± .015	.08 ± .009	.15 ± .004	.15 ± .004	.36 ± .165	.31 ± .005
Node2Vec CBOW	.85 ± .018	.25 ± .014	.25 ± .01	.25 ± .01	.70 ± .13	.40 ± .009
DeepWalk CBOW	.89 ± .013	.33 ± .016	.32 ± .013	.32 ± .013	.70 ± .102	.48 ± .013
RandNE	.94 ± .008	.47 ± .027	.42 ± .025	.42 ± .025	.88 ± .016	.54 ± .026
Second-order LINE	.94 ± .006	.55 ± .04	.49 ± .034	.49 ± .034	.87 ± .019	.59 ± .026
Node2Vec GloVe	.94 ± .007	.58 ± .016	.56 ± .016	.56 ± .016	.79 ± .048	.68 ± .014
GLEE	.92 ± .008	.64 ± .023	.58 ± .022	.58 ± .022	.81 ± .02	.68 ± .013
First-order LINE	.96 ± .005	.69 ± .023	.62 ± .023	.62 ± .023	.88 ± .012	.70 ± .017
DeepWalk GloVe	.95 ± .003	.68 ± .019	.65 ± .017	.65 ± .017	.81 ± .015	.73 ± .009
HOPE	.95 ± .008	.74 ± .021	.72 ± .022	.72 ± .022	.79 ± .02	.77 ± .019
Role2Vec	.96 ± .006	.78 ± .021	.75 ± .022	.75 ± .022	.85 ± .017	.79 ± .015
GraRep	.96 ± .006	.77 ± .025	.75 ± .026	.75 ± .026	.80 ± .024	.78 ± .023
Walklets	.97 ± .004	.80 ± .021	.76 ± .025	.76 ± .025	.86 ± .015	.81 ± .017
NetMF	.97 ± .003	.81 ± .014	.79 ± .017	.79 ± .017	.84 ± .013	.82 ± .014
DeepWalk SkipGram	.97 ± .003	.83 ± .012	.80 ± .015	.80 ± .015	.85 ± .01	.83 ± .01
Node2Vec SkipGram	.97 ± .004	.84 ± .016	.82 ± .019	.82 ± .019	.87 ± .013	.84 ± .017

Table F.8: Decision Tree node-label prediction performance in test evaluation on Cora. The rows are sorted by balanced accuracy.

	AUROC	F1 Score	Balanced Accuracy	Recall	Precision	Accuracy
NMFADMM	.53 ± .013	.19 ± .023	.19 ± .023	.19 ± .023	.19 ± .023	.23 ± .022
Node2Vec CBOW	.56 ± .012	.25 ± .02	.25 ± .02	.25 ± .02	.25 ± .02	.28 ± .026
DeepWalk CBOW	.57 ± .007	.27 ± .011	.27 ± .012	.27 ± .012	.27 ± .011	.31 ± .014
Second-order LINE	.62 ± .015	.34 ± .025	.34 ± .026	.34 ± .026	.34 ± .026	.38 ± .021
RandNE	.62 ± .011	.36 ± .022	.35 ± .021	.35 ± .021	.36 ± .025	.39 ± .015
First-order LINE	.65 ± .011	.40 ± .019	.41 ± .02	.41 ± .02	.40 ± .019	.43 ± .017
Node2Vec GloVe	.67 ± .015	.43 ± .025	.43 ± .025	.43 ± .025	.43 ± .026	.48 ± .029
DeepWalk GloVe	.69 ± .01	.46 ± .016	.47 ± .017	.47 ± .017	.47 ± .017	.52 ± .016
GLEE	.72 ± .014	.51 ± .023	.51 ± .02	.51 ± .02	.52 ± .027	.55 ± .023
Role2Vec	.72 ± .014	.51 ± .023	.51 ± .024	.51 ± .024	.51 ± .022	.54 ± .022
Walklets	.72 ± .008	.51 ± .015	.51 ± .013	.51 ± .013	.52 ± .018	.55 ± .02
HOPE	.81 ± .012	.65 ± .022	.64 ± .021	.64 ± .021	.66 ± .026	.67 ± .022
DeepWalk SkipGram	.79 ± .011	.64 ± .02	.64 ± .019	.64 ± .019	.65 ± .022	.67 ± .019
Node2Vec SkipGram	.81 ± .013	.67 ± .02	.67 ± .023	.67 ± .023	.68 ± .019	.70 ± .018
GraRep	.82 ± .005	.68 ± .01	.68 ± .01	.68 ± .01	.69 ± .011	.70 ± .008
NetMF	.85 ± .011	.73 ± .017	.74 ± .02	.74 ± .02	.74 ± .015	.75 ± .016

Table F.9: Random Forest node-label prediction performance in test evaluation on CiteSeer. The rows are sorted by balanced accuracy.

	AUROC	F1 Score	Balanced Accuracy	Recall	Precision	Accuracy
NMFADMM	.70 ± .017	.26 ± .019	.29 ± .014	.29 ± .014	.40 ± .027	.35 ± .016
Node2Vec CBOW	.74 ± .018	.35 ± .013	.37 ± .012	.37 ± .012	.45 ± .02	.43 ± .014
Second-order LINE	.79 ± .009	.37 ± .016	.39 ± .015	.39 ± .015	.59 ± .075	.46 ± .016
DeepWalk CBOW	.78 ± .013	.41 ± .021	.43 ± .021	.43 ± .021	.49 ± .056	.49 ± .023
GLEE	.81 ± .01	.47 ± .014	.47 ± .012	.47 ± .012	.58 ± .042	.54 ± .012
HOPE	.83 ± .012	.50 ± .024	.49 ± .02	.49 ± .02	.61 ± .046	.56 ± .02
GraRep	.85 ± .008	.52 ± .017	.52 ± .015	.52 ± .015	.60 ± .027	.58 ± .014
DeepWalk GloVe	.86 ± .01	.52 ± .015	.53 ± .016	.53 ± .016	.55 ± .049	.60 ± .017
Node2Vec GloVe	.86 ± .009	.53 ± .011	.54 ± .011	.54 ± .011	.56 ± .052	.61 ± .011
RandNE	.84 ± .01	.54 ± .024	.54 ± .022	.54 ± .022	.63 ± .033	.60 ± .022
NetMF	.87 ± .009	.55 ± .012	.54 ± .012	.54 ± .012	.65 ± .025	.60 ± .016
Role2Vec	.87 ± .006	.56 ± .012	.56 ± .012	.56 ± .012	.65 ± .043	.63 ± .013
First-order LINE	.88 ± .01	.60 ± .021	.61 ± .021	.61 ± .021	.75 ± .029	.68 ± .022
DeepWalk SkipGram	.90 ± .009	.61 ± .016	.61 ± .015	.61 ± .015	.68 ± .015	.68 ± .015
Node2Vec SkipGram	.90 ± .008	.62 ± .021	.62 ± .016	.62 ± .016	.68 ± .031	.68 ± .013
Walklets	.91 ± .006	.67 ± .02	.67 ± .016	.67 ± .016	.70 ± .023	.72 ± .012

Table F.10: Decision Tree node-label prediction performance in test evaluation on CiteSeer. The rows are sorted by balanced accuracy.

	AUROC	F1 Score	Balanced Accuracy	Recall	Precision	Accuracy
NMFADMM	$.53 \pm .011$	$.22 \pm .019$	$.22 \pm .019$	$.22 \pm .019$	$.23 \pm .02$	$.24 \pm .019$
Second-order LINE	$.54 \pm .014$	$.23 \pm .023$	$.23 \pm .023$	$.23 \pm .023$	$.23 \pm .023$	$.25 \pm .024$
Node2Vec CBOW	$.56 \pm .008$	$.26 \pm .013$	$.26 \pm .013$	$.26 \pm .013$	$.26 \pm .014$	$.27 \pm .016$
DeepWalk CBOW	$.56 \pm .007$	$.27 \pm .012$	$.27 \pm .012$	$.27 \pm .012$	$.27 \pm .013$	$.29 \pm .013$
First-order LINE	$.60 \pm .013$	$.34 \pm .021$	$.34 \pm .022$	$.34 \pm .022$	$.34 \pm .021$	$.36 \pm .023$
RandNE	$.61 \pm .014$	$.35 \pm .024$	$.35 \pm .024$	$.35 \pm .024$	$.35 \pm .024$	$.37 \pm .021$
DeepWalk GloVe	$.63 \pm .008$	$.38 \pm .014$	$.38 \pm .014$	$.38 \pm .014$	$.39 \pm .014$	$.41 \pm .015$
Node2Vec GloVe	$.64 \pm .008$	$.39 \pm .013$	$.39 \pm .013$	$.39 \pm .013$	$.39 \pm .014$	$.42 \pm .017$
Role2Vec	$.65 \pm .011$	$.41 \pm .019$	$.41 \pm .019$	$.41 \pm .019$	$.42 \pm .019$	$.44 \pm .019$
GLEE	$.72 \pm .012$	$.44 \pm .019$	$.43 \pm .017$	$.43 \pm .017$	$.49 \pm .024$	$.48 \pm .017$
HOPE	$.74 \pm .016$	$.46 \pm .021$	$.45 \pm .018$	$.45 \pm .018$	$.54 \pm .025$	$.50 \pm .019$
Walklets	$.69 \pm .016$	$.48 \pm .026$	$.48 \pm .026$	$.48 \pm .026$	$.49 \pm .025$	$.51 \pm .026$
DeepWalk SkipGram	$.70 \pm .011$	$.50 \pm .018$	$.50 \pm .018$	$.50 \pm .018$	$.50 \pm .019$	$.52 \pm .018$
GraRep	$.76 \pm .008$	$.51 \pm .014$	$.50 \pm .013$	$.50 \pm .013$	$.55 \pm .017$	$.54 \pm .012$
Node2Vec SkipGram	$.71 \pm .01$	$.51 \pm .017$	$.51 \pm .016$	$.51 \pm .016$	$.51 \pm .016$	$.53 \pm .019$
NetMF	$.78 \pm .017$	$.56 \pm .025$	$.55 \pm .025$	$.55 \pm .025$	$.58 \pm .025$	$.59 \pm .026$

Table F.11: Random Forest node-label prediction performance in test evaluation on PubMedDiabetes. The rows are sorted by balanced accuracy.

	AUROC	F1 Score	Balanced Accuracy	Recall	Precision	Accuracy
NMFADMM	$.55 \pm .006$	$.31 \pm .006$	$.36 \pm .006$	$.36 \pm .006$	$.29 \pm .005$	$.43 \pm .007$
RandNE	$.73 \pm .008$	$.40 \pm .005$	$.45 \pm .006$	$.45 \pm .006$	$.53 \pm .157$	$.54 \pm .007$
Node2Vec CBOW	$.80 \pm .006$	$.46 \pm .007$	$.52 \pm .008$	$.52 \pm .008$	$.75 \pm .006$	$.62 \pm .009$
DeepWalk CBOW	$.83 \pm .007$	$.48 \pm .005$	$.53 \pm .005$	$.53 \pm .005$	$.74 \pm .018$	$.63 \pm .005$
First-order LINE	$.88 \pm .004$	$.53 \pm .009$	$.57 \pm .006$	$.57 \pm .006$	$.75 \pm .008$	$.66 \pm .006$
Second-order LINE	$.90 \pm .004$	$.54 \pm .006$	$.59 \pm .004$	$.59 \pm .004$	$.75 \pm .011$	$.68 \pm .004$
Node2Vec GloVe	$.89 \pm .005$	$.73 \pm .007$	$.72 \pm .007$	$.72 \pm .007$	$.76 \pm .006$	$.76 \pm .006$
DeepWalk GloVe	$.90 \pm .004$	$.75 \pm .007$	$.74 \pm .006$	$.74 \pm .006$	$.77 \pm .007$	$.77 \pm .006$
Role2Vec	$.92 \pm .003$	$.76 \pm .007$	$.75 \pm .007$	$.75 \pm .007$	$.79 \pm .006$	$.79 \pm .006$
Walklets	$.93 \pm .004$	$.77 \pm .005$	$.76 \pm .005$	$.76 \pm .005$	$.81 \pm .005$	$.80 \pm .005$
GLEE	$.92 \pm .003$	$.78 \pm .007$	$.77 \pm .007$	$.77 \pm .007$	$.79 \pm .007$	$.80 \pm .006$
HOPE	$.92 \pm .003$	$.78 \pm .005$	$.78 \pm .005$	$.78 \pm .005$	$.79 \pm .006$	$.80 \pm .005$
GraRep	$.93 \pm .003$	$.79 \pm .003$	$.79 \pm .003$	$.79 \pm .003$	$.80 \pm .004$	$.81 \pm .003$
NetMF	$.93 \pm .002$	$.80 \pm .004$	$.79 \pm .005$	$.79 \pm .005$	$.80 \pm .004$	$.81 \pm .004$
Node2Vec SkipGram	$.93 \pm .002$	$.81 \pm .005$	$.81 \pm .005$	$.81 \pm .005$	$.82 \pm .006$	$.82 \pm .005$
DeepWalk SkipGram	$.93 \pm .003$	$.81 \pm .006$	$.81 \pm .006$	$.81 \pm .006$	$.82 \pm .006$	$.82 \pm .005$

Table F.12: Decision Tree node-label prediction performance in test evaluation on PubMedDiabetes. The rows are sorted by balanced accuracy.

	AUROC	F1 Score	Balanced Accuracy	Recall	Precision	Accuracy
NMFADMM	.50 ± .006	.34 ± .008	.34 ± .008	.34 ± .008	.34 ± .008	.36 ± .009
RandNE	.54 ± .006	.39 ± .008	.39 ± .008	.39 ± .008	.39 ± .008	.41 ± .009
First-order LINE	.60 ± .007	.46 ± .008	.46 ± .008	.46 ± .008	.46 ± .008	.48 ± .008
Second-order LINE	.60 ± .008	.47 ± .011	.47 ± .011	.47 ± .011	.47 ± .011	.49 ± .012
Node2Vec CBOW	.61 ± .005	.48 ± .007	.48 ± .007	.48 ± .007	.48 ± .007	.50 ± .008
DeepWalk CBOW	.61 ± .004	.48 ± .005	.48 ± .005	.48 ± .005	.48 ± .005	.50 ± .005
Node2Vec GloVe	.70 ± .006	.59 ± .008	.59 ± .008	.59 ± .008	.60 ± .008	.62 ± .008
Role2Vec	.70 ± .008	.60 ± .01	.60 ± .01	.60 ± .01	.60 ± .009	.62 ± .01
Walklets	.70 ± .008	.60 ± .011	.60 ± .011	.60 ± .011	.60 ± .01	.62 ± .009
DeepWalk GloVe	.71 ± .005	.61 ± .007	.61 ± .007	.61 ± .007	.61 ± .007	.63 ± .008
DeepWalk SkipGram	.77 ± .005	.70 ± .006	.70 ± .006	.70 ± .006	.70 ± .005	.71 ± .006
Node2Vec SkipGram	.78 ± .003	.71 ± .005	.71 ± .005	.71 ± .005	.71 ± .005	.72 ± .005
GraRep	.79 ± .004	.72 ± .005	.72 ± .006	.72 ± .006	.72 ± .004	.73 ± .005
GLEE	.81 ± .006	.72 ± .006	.73 ± .006	.73 ± .006	.72 ± .006	.73 ± .006
NetMF	.83 ± .007	.74 ± .008	.74 ± .008	.74 ± .008	.74 ± .008	.75 ± .006
HOPE	.83 ± .005	.74 ± .006	.74 ± .006	.74 ± .006	.74 ± .006	.75 ± .006

Appendix G

Experiments on large real-world graphs: results and information

This section includes detailed information about the data and the software we used to build up the three real world big graphs used in the experiments. In particular Table G.1 reports the pre-processed data we used to construct the graphs. Table G.2 and G.3 show data and scripts used to build up respectively the PheKnowLator Knowledge Graph and the CTD and Wikipedia graphs.

Figures and Tables in this section show the comparison of the experimental results on the above big real-world graphs estimated using different metrics obtained by the Decision Trees trained on Node2vec embeddings generated by *GRAPE* and the other state-of-the-art graph embedding libraries.

Resource	**Link**
Prebuilt CTD	`https://archive.org/download/ctd_20220404/CTD.tar`
Prebuilt PheKnowLator	`https://archive.org/download/pheknowlator_20220411/PheKnowLator.tar`
Prebuilt English Wikipedia	`https://archive.org/download/wikipedia_edge_list.npy/wikipedia_edge_list.npy.gz`

Table G.1: Preprocessed datasets used to build-up the experiments for CTD, the PheKnowLator biomedical KG and Wikipedia.

Resource	Link
PheKnowLator build datasets	`https://github.com/callahantiff/PheKnowLator/wiki/v2-Data-Sources`
PheKnowLator used in experiments	`https://storage.googleapis.com/pheknowlator/archived_builds/release_v3.0.2/build_18OCT2021/data/original_data/downloaded_build_metadata.txt`
Preprocessed PheKnowLator data	`https://storage.googleapis.com/pheknowlator/archived_builds/release_v3.0.2/build_18OCT2021/data/processed_data/preprocessed_build_metadata.txt`
PheKnowLator build scripts	`https://github.com/callahantiff/PheKnowLator/tree/master/builds`
PheKnowLator build logs	`https://storage.googleapis.com/pheknowlator/archived_builds/release_v3.0.2/build_18OCT2021/knowledge_graphs/subclass_builds/inverse_relations/owlnets/pkt_build_log.log`

Table G.2: Ontologies, open link data sources and scripts used for the generation of the PheKnowLator Knowledge Graph.

Resource	Link
Complete CTD dataset	`http://ctdbase.org/reports/`
Dumps of Wikipedia	`https://dumps.wikimedia.org/backup-index.html`
English Wikipedia script	`https://github.com/AnacletoLAB/ensmallen/blob/develop/bindings/python/ensmallen/datasets/wikipedia_automatic_graph_retrieval.py`

Table G.3: Data and script used to construct the CTD and English Wikipedia graphs.

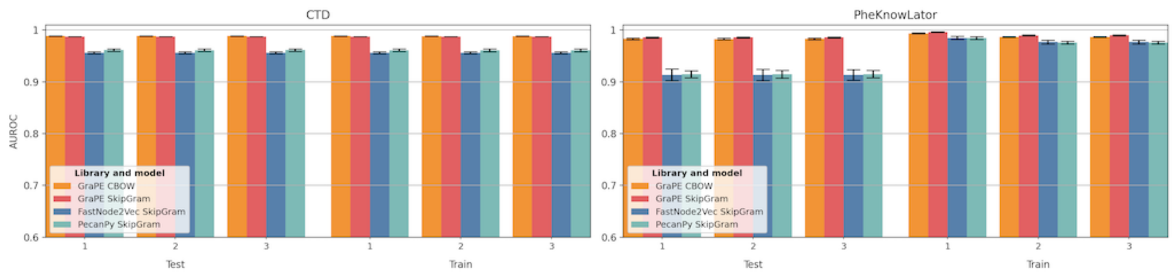


Figure G.1: Average auroc of Decision Tree trained. Results are averaged across ten holdouts. The bar plot on the left is relative to the CTD graph, while the one on the right is relative to the PheKnowLator graph. The number on the horizontal axis represents different unbalance rates of existing and non-existing edges considered during the evaluation: 1 means a balanced rate of existing and non-existing edges, 2 means 1 : 2 and 3 means 1 : 3. All embedding methods achieve worse performance with the increasing unbalance rates, but *GRAPE* implementations consistently outperform the other GenSim-based libraries.

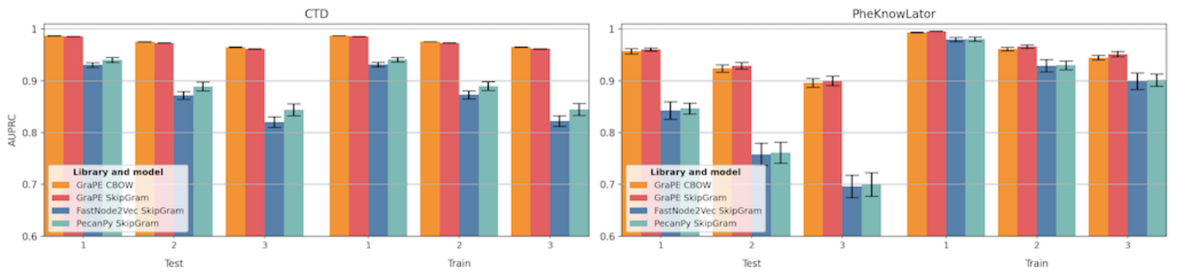


Figure G.2: Average auprc of Decision Tree trained. Results are averaged across ten holdouts. The bar plot on the left is relative to the CTD graph, while the one on the right is relative to the PheKnowLator graph. The number on the horizontal axis represents different unbalance rates of existing and non-existing edges considered during the evaluation: 1 means a balanced rate of existing and non-existing edges, 2 means 1 : 2 and 3 means 1 : 3. All embedding methods achieve worse performance with the increasing unbalance rates, but *GRAPE* implementations consistently outperform the other GenSim-based libraries.

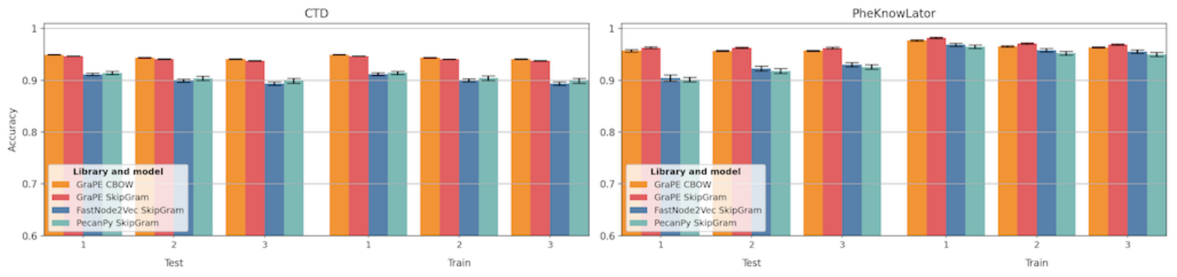


Figure G.3: Average accuracy of Decision Tree trained. Results are averaged across ten holdouts. The bar plot on the left is relative to the CTD graph, while the one on the right is relative to the PheKnowLator graph. The number on the horizontal axis represents different unbalance rates of existing and non-existing edges considered during the evaluation: 1 means a balanced rate of existing and non-existing edges, 2 means 1 : 2 and 3 means 1 : 3. All embedding methods achieve worse performance with the increasing unbalance rates, but *GRAPE* implementations consistently outperform the other GenSim-based libraries.

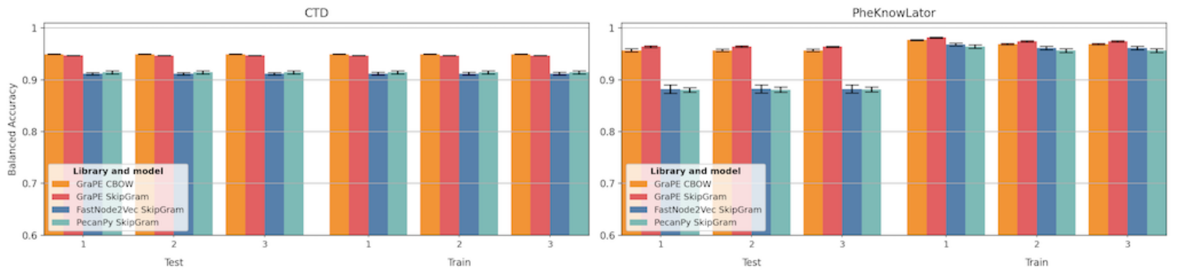


Figure G.4: Average balanced accuracy of Decision Tree trained. Results are averaged across ten holdouts. The bar plot on the left is relative to the CTD graph, while the one on the right is relative to the PheKnowLator graph. The number on the horizontal axis represents different unbalance rates of existing and non-existing edges considered during the evaluation: 1 means a balanced rate of existing and non-existing edges, 2 means 1 : 2 and 3 means 1 : 3. All embedding methods achieve worse performance with the increasing unbalance rates, but *GRAPE* implementations consistently outperform the other GenSim-based libraries.

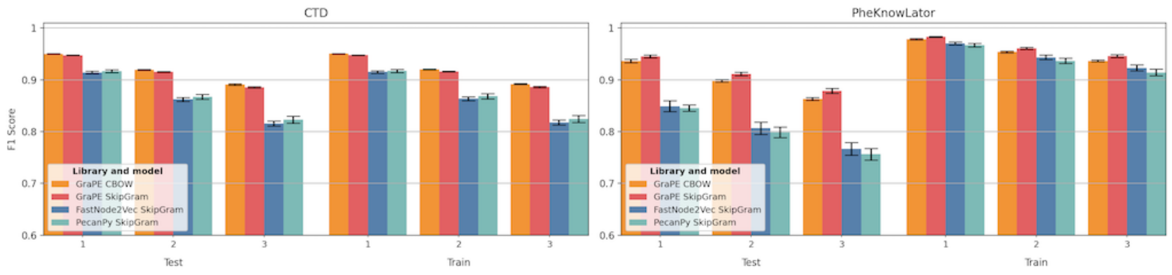


Figure G.5: Average f1 score of Decision Tree trained. Results are averaged across ten holdouts. The bar plot on the left is relative to the CTD graph, while the one on the right is relative to the PheKnowLator graph. The number on the horizontal axis represents different unbalance rates of existing and non-existing edges considered during the evaluation: 1 means a balanced rate of existing and non-existing edges, 2 means 1 : 2 and 3 means 1 : 3. All embedding methods achieve worse performance with the increasing unbalance rates, but *GRAPE* implementations consistently outperform the other GenSim-based libraries.

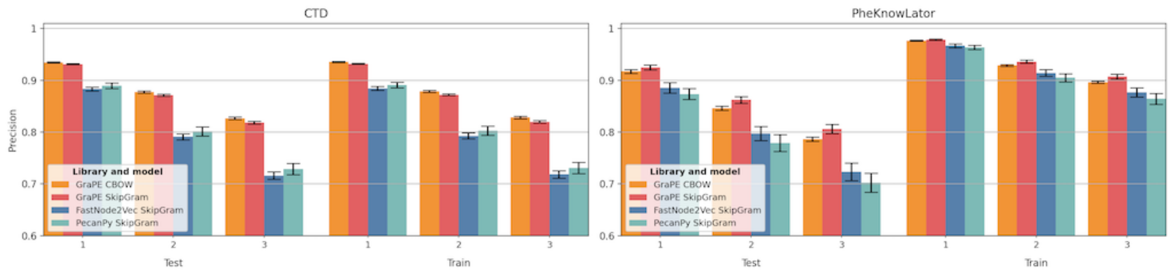


Figure G.6: Average precision of Decision Tree trained. Results are averaged across ten holdouts. The bar plot on the left is relative to the CTD graph, while the one on the right is relative to the PheKnowLator graph. The number on the horizontal axis represents different unbalance rates of existing and non-existing edges considered during the evaluation: 1 means a balanced rate of existing and non-existing edges, 2 means 1 : 2 and 3 means 1 : 3. All embedding methods achieve worse performance with the increasing unbalance rates, but *GRAPE* implementations consistently outperform the other GenSim-based libraries.

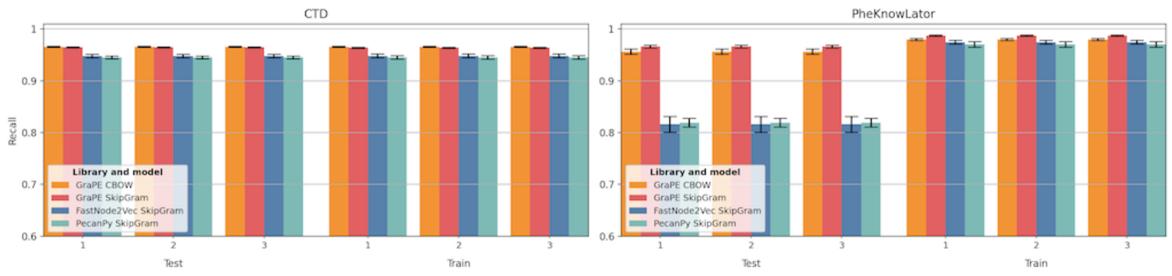


Figure G.7: Average recall of Decision Tree trained. Results are averaged across ten holdouts. The bar plot on the left is relative to the CTD graph, while the one on the right is relative to the PheKnowLator graph. The number on the horizontal axis represents different unbalance rates of existing and non-existing edges considered during the evaluation: 1 means a balanced rate of existing and non-existing edges, 2 means 1 : 2 and 3 means 1 : 3. All embedding methods achieve worse performance with the increasing unbalance rates, but *GRAPE* implementations consistently outperform the other GenSim-based libraries.

Table G.4: Decision Tree edge prediction performance in train evaluation on CTD with unbalance rate 1

library & model	Accuracy	Precision	F1 Score	AUPRC	Balanced Accuracy	Recall	AUROC
FastNode2Vec SG	.91 ± .0027	.88 ± .0036	.92 ± .0025	.93 ± .0042	.91 ± .0027	.95 ± .0036	.96 ± .002
GRAPE CBOW	.95 ± .0003	.94 ± .0011	.95 ± .0003	.99 ± .0002	.95 ± .0003	.97 ± .0011	.99 ± .0001
GRAPE SG	.95 ± .0003	.93 ± .0011	.95 ± .0003	.99 ± .0001	.95 ± .0003	.96 ± .001	.99 ± .0001
PecanPy SG	.91 ± .0031	.89 ± .0054	.92 ± .0028	.94 ± .0047	.91 ± .0031	.95 ± .0031	.96 ± .0027

Table G.5: Decision Tree edge prediction performance in train evaluation on CTD with unbalance rate 2

library & model	Accuracy	Precision	F1 Score	AUPRC	Balanced Accuracy	Recall	AUROC
FastNode2Vec SG	.90 ± .003	.79 ± .0058	.86 ± .0037	.87 ± .0075	.91 ± .0027	.95 ± .0036	.96 ± .002
GRAPE CBOW	.94 ± .0006	.88 ± .0019	.92 ± .0007	.98 ± .0003	.95 ± .0003	.97 ± .0011	.99 ± .0001
GRAPE SG	.94 ± .0006	.87 ± .0019	.92 ± .0007	.97 ± .0001	.95 ± .0003	.96 ± .001	.99 ± .0001
PecanPy SG	.90 ± .0041	.80 ± .0087	.87 ± .0049	.89 ± .0084	.91 ± .0031	.95 ± .0031	.96 ± .0027

Table G.6: Decision Tree edge prediction performance in train evaluation on CTD with unbalance rate 3

library & model	Accuracy	Precision	F1 Score	AUPRC	Balanced Accuracy	Recall	AUROC
FastNode2Vec SG	.89 ± .0033	.72 ± .0072	.82 ± .0048	.82 ± .0102	.91 ± .0027	.95 ± .0036	.96 ± .002
GRAPE CBOW	.94 ± .0007	.83 ± .0026	.89 ± .0012	.97 ± .0004	.95 ± .0003	.97 ± .0011	.99 ± .0001
GRAPE SG	.94 ± .0007	.82 ± .0025	.89 ± .0011	.96 ± .0002	.95 ± .0003	.96 ± .001	.99 ± .0001
PecanPy SG	.90 ± .0047	.73 ± .0109	.82 ± .0067	.84 ± .0114	.91 ± .0031	.95 ± .0031	.96 ± .0027

Table G.7: Decision Tree edge prediction performance in test evaluation on CTD with unbalance rate 1

library & model	Accuracy	Precision	F1 Score	AUPRC	Balanced Accuracy	Recall	AUROC
FastNode2Vec SG	.91 ± .0026	.88 ± .0036	.91 ± .0024	.93 ± .0043	.91 ± .0026	.95 ± .0035	.96 ± .002
GRAPE CBOW	.95 ± .0004	.93 ± .0011	.95 ± .0003	.99 ± .0002	.95 ± .0004	.97 ± .0011	.99 ± .0001
GRAPE SG	.95 ± .0003	.93 ± .0011	.95 ± .0003	.99 ± .0001	.95 ± .0003	.96 ± .001	.99 ± .0001
PecanPy SG	.91 ± .0031	.89 ± .0054	.92 ± .0028	.94 ± .0047	.91 ± .0031	.95 ± .003	.96 ± .0026

Table G.8: Decision Tree edge prediction performance in test evaluation on CTD with unbalance rate 2

library & model	Accuracy	Precision	F1 Score	AUPRC	Balanced Accuracy	Recall	AUROC
FastNode2Vec SG	.90 ± .003	.79 ± .0058	.86 ± .0037	.87 ± .0076	.91 ± .0026	.95 ± .0035	.96 ± .002
GRAPE CBOW	.94 ± .0006	.88 ± .002	.92 ± .0007	.98 ± .0003	.95 ± .0004	.97 ± .0011	.99 ± .0001
GRAPE SG	.94 ± .0006	.87 ± .0019	.92 ± .0007	.97 ± .0001	.95 ± .0003	.96 ± .001	.99 ± .
PecanPy SG	.90 ± .0041	.80 ± .0089	.87 ± .005	.89 ± .0085	.91 ± .0031	.95 ± .003	.96 ± .0027

Table G.9: Decision Tree edge prediction performance in test evaluation on CTD with unbalance rate 3

library & model	Accuracy	Precision	F1 Score	AUPRC	Balanced Accuracy	Recall	AUROC
FastNode2Vec SG	.89 ± .0033	.72 ± .0072	.82 ± .0047	.82 ± .0102	.91 ± .0026	.95 ± .0035	.96 ± .002
GRAPE CBOW	.94 ± .0007	.83 ± .0025	.89 ± .0011	.96 ± .0004	.95 ± .0003	.97 ± .0011	.99 ± .0001
GRAPE SG	.94 ± .0007	.82 ± .0025	.89 ± .0011	.96 ± .0002	.95 ± .0003	.96 ± .001	.99 ± .0001
PecanPy SG	.90 ± .0047	.73 ± .0109	.82 ± .0067	.84 ± .0115	.91 ± .0031	.95 ± .003	.96 ± .0026

Table G.10: Decision Tree edge prediction performance in train evaluation on PheKnowLator with unbalance rate 1

library & model	Accuracy	Precision	F1 Score	AUPRC	Balanced Accuracy	Recall	AUROC
FastNode2Vec SG	.97 ± .0029	.97 ± .0036	.97 ± .0027	.98 ± .0043	.97 ± .0029	.97 ± .0037	.98 ± .003
GRAPE CBOW	.98 ± .0013	.98 ± .0009	.98 ± .0012	.99 ± .0008	.98 ± .0012	.98 ± .002	.99 ± .0009
GRAPE SG	.98 ± .0012	.98 ± .0014	.98 ± .0011	10 ± .0005	.98 ± .0012	.99 ± .0015	10 ± .0005
PecanPy SG	.96 ± .0035	.96 ± .0038	.97 ± .0033	.98 ± .004	.96 ± .0034	.97 ± .0057	.98 ± .0025

Table G.11: Decision Tree edge prediction performance in train evaluation on PheKnowLator with unbalance rate 2

library & model	Accuracy	Precision	F1 Score	AUPRC	Balanced Accuracy	Recall	AUROC
FastNode2Vec SG	.96 ± .0033	.91 ± .0065	.94 ± .0043	.93 ± .0116	.96 ± .0032	.97 ± .0037	.98 ± .0037
GRAPE CBOW	.97 ± .0012	.93 ± .0016	.95 ± .0016	.96 ± .0033	.97 ± .0013	.98 ± .002	.99 ± .0008
GRAPE SG	.97 ± .0014	.94 ± .0032	.96 ± .0019	.97 ± .0036	.97 ± .0013	.99 ± .0015	.99 ± .0012
PecanPy SG	.95 ± .0038	.91 ± .0078	.94 ± .0049	.93 ± .0085	.96 ± .0037	.97 ± .0057	.98 ± .0028

Table G.12: Decision Tree edge prediction performance in train evaluation on PheKnowLator with unbalance rate 3

library & model	Accuracy	Precision	F1 Score	AUPRC	Balanced Accuracy	Recall	AUROC
FastNode2Vec SG	.96 ± .0035	.88 ± .009	.92 ± .0057	.90 ± .0159	.96 ± .0032	.97 ± .0037	.98 ± .0036
GRAPE CBOW	.96 ± .001	.90 ± .002	.94 ± .0018	.94 ± .0045	.97 ± .0013	.98 ± .002	.99 ± .0008
GRAPE SG	.97 ± .0016	.91 ± .0048	.95 ± .0027	.95 ± .0051	.97 ± .0014	.99 ± .0015	.99 ± .0012
PecanPy SG	.95 ± .004	.86 ± .0104	.91 ± .0064	.90 ± .0118	.96 ± .0037	.97 ± .0057	.98 ± .0029

Table G.13: Decision Tree edge prediction performance in test evaluation on PheKnowLator with unbalance rate 1

library & model	Accuracy	Precision	F1 Score	AUPRC	Balanced Accuracy	Recall	AUROC
FastNode2Vec SG	.90 ± .0062	.89 ± .0101	.85 ± .0105	.84 ± .0167	.88 ± .0082	.82 ± .0152	.91 ± .0112
GRAPE CBOW	.96 ± .0024	.92 ± .0037	.94 ± .0036	.96 ± .0053	.96 ± .003	.96 ± .0053	.98 ± .0017
GRAPE SG	.96 ± .0018	.92 ± .0047	.95 ± .0026	.96 ± .0032	.96 ± .0017	.97 ± .0027	.99 ± .0011
PecanPy SG	.90 ± .0042	.87 ± .0104	.85 ± .0063	.85 ± .0105	.88 ± .0046	.82 ± .0086	.91 ± .0065

Table G.14: Decision Tree edge prediction performance in test evaluation on PheKnowLator with unbalance rate 2

library & model	Accuracy	Precision	F1 Score	AUPRC	Balanced Accuracy	Recall	AUROC
FastNode2Vec SG	.92 ± .0046	.80 ± .0134	.81 ± .0117	.76 ± .0215	.88 ± .0081	.82 ± .0152	.91 ± .0108
GRAPE CBOW	.96 ± .0011	.85 ± .004	.90 ± .0026	.92 ± .0072	.96 ± .0024	.96 ± .0053	.98 ± .0016
GRAPE SG	.96 ± .0015	.86 ± .0066	.91 ± .0033	.93 ± .0063	.96 ± .0012	.97 ± .0027	.99 ± .0012
PecanPy SG	.92 ± .0048	.78 ± .0161	.80 ± .0101	.76 ± .0201	.88 ± .0053	.82 ± .0086	.91 ± .0073

Table G.15: Decision Tree edge prediction performance in test evaluation on PheKnowLator with unbalance rate 3

library & model	Accuracy	Precision	F1 Score	AUPRC	Balanced Accuracy	Recall	AUROC
FastNode2Vec SG	.93 ± .004	.72 ± .0169	.77 ± .0121	.70 ± .0218	.88 ± .0076	.82 ± .0152	.91 ± .0102
GRAPE CBOW	.96 ± .001	.79 ± .0042	.86 ± .0029	.90 ± .0085	.96 ± .0025	.96 ± .0053	.98 ± .0017
GRAPE SG	.96 ± .0017	.81 ± .0088	.88 ± .0046	.90 ± .0092	.96 ± .001	.97 ± .0027	.99 ± .0013
PecanPy SG	.93 ± .0044	.70 ± .0184	.76 ± .0113	.70 ± .0225	.88 ± .0047	.82 ± .0086	.92 ± .0069

Appendix H

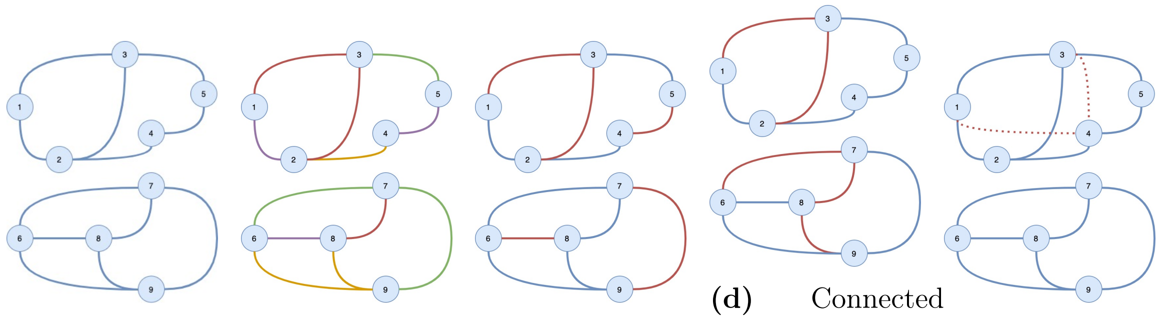
Other *GRAPE* utilities.

H.1 Graph analysis and graph reporting

GRAPE implements fast algorithms to analyze the overall characteristics of the graph, including Breadth and Depth-first search, Dijkstra, Tarjan's strongly connected components, efficient Diameter computation, spanning arborescence and connected components, approximated vertex cover, triads counting, transitivity, clustering coefficient and triangles counting, Betweenness and stress centrality, Closeness and harmonic centrality, as well as optimized implementations for algebraic set graph-operations and node and edge filters.

H.2 Construction of train and test graphs for edge prediction

To generate train and test samples for the evaluation of edge prediction task, *GRAPE* makes available Kfold, Monte Carlo and Connected Monte Carlo techniques. All methods may be stratified relatively to user-provided edge types. We show examples in figure H.1 on a graph with two components.



(a) Original graph **(b)** Kfolds **(c)** Monte Carlo **(d)** Connected Monte Carlo **(e)** Negative edges

Figure H.1: Different holdouts on a graph with multiple components: From left to right, A) the graph considered, composed of two connected components, B) K-folds edge holdouts (we represent each validation fold with a different colour), C) Monte Carlo Holdout, sampling random edges for the validation set (in red) that may generate new connected components and D) Connected Monte Carlo Holdout, sampling random edges for the validation set (in red) without generating new connected components. Finally, in picture E), we show in red as dotted, the generated negative edges.

H.2.1 K-folds

As per the normal Kfolds, this mechanism splits the graph edges into k folds to be used for cross-validating a model on a link prediction task (figure H.1b). We suggest using this method when the graph is either composed of a single connected component with high density or, if multiple connected components exist, each component has a high density to avoid creating new components when the procedure randomly removes edges.

H.2.2 Monte Carlo

This method randomly samples the edges, but multiple holdouts may share the same edges (figure H.1c). The training graph generated with this method, as for the aforementioned k-folds method, may contain more components than the original graph.

H.2.3 Connected Monte Carlo

Often, in link prediction tasks, it is assumed a closed word hypothesis: that is, components that are not connected do not have unknown edges connecting them. By using the Connected Monte Carlo holdouts guarantees that the training graph always has the same number of connected components of the original graph without creating new ones by reserving a set of edges forming a spanning arborescence for the training set and sampling the test set edges only from the remaining edges (figure H.1d). The connected Monte Carlo holdout avoids introducing a negative bias when the task assumes a closed word hypothesis. For instance, a link prediction model working on the closed word assumption would not predict links between the different components that, for instance, a simple Monte Carlo holdout may generate.

H.2.4 Negative edge sampling

To train edge prediction models, it is common practice to generate negative edges, that is edges that do not exist in the graph. We generate such edges by sampling their vertices from the same connected components, to avoid easily predictable negative edges. Indeed, from our experimental studies we noted that negative edges between two graph connected components have odd embeddings that make it easy to identify them, causing a positive bias in the model performance (figure H.1e). To sample massive amounts of source and destination nodes we use a SIMD-vectorized version of xorshift, detailed in the section B. Edges may be sampled by either following a uniform, or more preferably, a scale-free distribution. The latter has been shown to introduce less covariate-shift and therefore make the sampled negative edges a meaningful task, while often the uniform sampling can sample trivially false edges.

Appendix I

Additional ALPINE results

In this chapter we briefly report the model parameters used in the ALPINE experiments and the results obtained in the node-label and edge predictions.

I.1 Node embedding models parameters

In this section we succinctly report the parameters employed to train all the considered node embedding models within the ALPINE experiments.

Parameter	Value
random_state	42
embedding_size	100
epochs	100
learning_rate	0.05
learning_rate_decay	0.9
use_scale_free_distribution	True
dtype	f32

Table I.1: Parameters used for node embedding model *Second-order LINE*

Parameter	Value
random_state	42
embedding_size	100
epochs	100
learning_rate	0.05
learning_rate_decay	0.9
use_scale_free_distribution	True
dtype	f32

Table I.2: Parameters used for node embedding model *First-order LINE*

Parameter	Value
embedding_size	100
dtype	u8
window_size	2

Table I.3: Parameters used for node embedding model *Degree-based WINE*

Parameter	Value
dtype	u8
window_size	2

Table I.4: Parameters used for node embedding model *Node-label-based WINE*

Parameter	Value
dtype	u8

Table I.5: Parameters used for node embedding model *Node-label-based SPINE*

Parameter	Value
embedding_size	100
dtype	u8

Table I.6: Parameters used for node embedding model *Degree-based SPINE*

Parameter	Value
random_state	42
embedding_size	100
alpha	0.75
epochs	30
clipping_value	6.0
walk_length	128
iterations	10
window_size	5
max_neighbours	100
learning_rate	0.001
learning_rate_decay	0.9
dtype	f32

Table I.7: Parameters used for node embedding model *DeepWalk GloVe*

Parameter	Value
random_state	42
embedding_size	100
epochs	30
clipping_value	6.0
number_of_negative_samples	10
walk_length	128
iterations	10
window_size	5
max_neighbours	100
learning_rate	0.01
learning_rate_decay	0.9
use_scale_free_distribution	True
dtype	f32

Table I.8: Parameters used for node embedding model *DeepWalk CBOW*

Parameter	Value
random_state	42
embedding_size	100
epochs	30
clipping_value	6.0
number_of_negative_samples	10
walk_length	128
iterations	10
window_size	5
max_neighbours	100
learning_rate	0.01
learning_rate_decay	0.9
use_scale_free_distribution	True
dtype	f32

Table I.9: Parameters used for node embedding model *DeepWalk SkipGram*

Parameter	Value
random_state	42
embedding_size	100
epochs	30
walk_length	128
iterations	10
window_size	4
return_weight	1.0
explore_weight	1.0
max_neighbours	100
learning_rate	0.001
learning_rate_decay	0.9
alpha	0.75
use_scale_free_distribution	True
dtype	f32

Table I.10: Parameters used for node embedding model *Walklets GloVe*

Parameter	Value
random_state	42
embedding_size	100
epochs	30
clipping_value	6.0
number_of_negative_samples	10
walk_length	128
iterations	10
window_size	4
return_weight	1.0
explore_weight	1.0
max_neighbours	100
learning_rate	0.01
learning_rate_decay	0.9
use_scale_free_distribution	True
dtype	f32

Table I.11: Parameters used for node embedding model *Walklets CBOW*

Parameter	Value
random_state	42
embedding_size	100
epochs	30
clipping_value	6.0
number_of_negative_samples	10
walk_length	128
iterations	10
window_size	4
return_weight	1.0
explore_weight	1.0
max_neighbours	100
learning_rate	0.01
learning_rate_decay	0.9
use_scale_free_distribution	True
dtype	f32

Table I.12: Parameters used for node embedding model *Walklets SkipGram*

Parameter	Value
embedding_size	100
metric	Neighbours Intersection size

Table I.13: Parameters used for node embedding model *HOPE*

Parameter	Value
random_state	42
embedding_size	100
walk_length	128
iterations	10
window_size	10
max_neighbours	100

Table I.14: Parameters used for node embedding model *NetMF*

I.2 Node-label prediction

In this section we succinctly report the average and standard deviations of all the considered node embedding models within the ALPINE node-label prediction experiments.

I.2.1 Random forest parameters

The edge prediction model considered for the ALPINE node-label prediction experiment is a Random Forest Classifier. The parameters used to train this random forest are reported in table I.15.

Parameter	Value
random_state	42
n_estimators	1000
criterion	gini
max_depth	10
min_samples_split	2
min_samples_leaf	1
max_features	sqrt
bootstrap	True
n_jobs	24
class_weight	balanced

Table I.15: Parameters used for classifier model *Random Forest Classifier*

I.2.2 Performance

Method	**Balanced Accuracy**	**F1 Score**	**Precision**	**Recall**	**AUROC**
DW CBOW	0.67 ± 0.06	0.72 ± 0.05	0.84 ± 0.02	0.67 ± 0.06	0.95 ± 0.01
DW GloVe	0.77 ± 0.03	0.79 ± 0.02	0.82 ± 0.02	0.77 ± 0.03	0.96 ± 0.01
DW SG	0.83 ± 0.03	0.84 ± 0.02	0.86 ± 0.02	0.83 ± 0.03	0.97 ± 0.00
Degree SPINE	0.74 ± 0.02	0.75 ± 0.02	0.77 ± 0.02	0.74 ± 0.02	0.94 ± 0.01
Degree WINE	0.68 ± 0.01	0.69 ± 0.02	0.71 ± 0.01	0.68 ± 0.01	0.92 ± 0.01
HOPE	0.74 ± 0.02	0.72 ± 0.02	0.73 ± 0.02	0.74 ± 0.02	0.94 ± 0.01
LINE (1st)	0.73 ± 0.09	0.77 ± 0.07	0.84 ± 0.01	0.73 ± 0.09	0.95 ± 0.02
LINE (2nd)	0.42 ± 0.05	0.46 ± 0.06	0.78 ± 0.03	0.42 ± 0.05	0.85 ± 0.02
Label SPINE	0.68 ± 0.03	0.63 ± 0.03	0.69 ± 0.03	0.68 ± 0.03	0.90 ± 0.01
Label WINE	0.71 ± 0.02	0.71 ± 0.02	0.74 ± 0.02	0.71 ± 0.02	0.94 ± 0.01
NetMF	0.80 ± 0.02	0.81 ± 0.02	0.83 ± 0.02	0.80 ± 0.02	0.97 ± 0.00
Walklets CBOW	0.80 ± 0.02	0.82 ± 0.02	0.85 ± 0.02	0.80 ± 0.02	0.97 ± 0.00
Walklets GloVe	0.60 ± 0.07	0.67 ± 0.06	0.86 ± 0.01	0.60 ± 0.07	0.94 ± 0.01
Walklets SG	0.69 ± 0.08	0.74 ± 0.07	0.86 ± 0.02	0.69 ± 0.08	0.95 ± 0.01

Table I.16: Node-label prediction performance of Random Forest model on different node embedding of the Cora graph The value reported are the average and standard deviations across 10 stratified Monte Carlo holdouts.

Method	**Balanced Accuracy**	**F1 Score**	**Precision**	**Recall**	**AUROC**
DW CBOW	0.57 ± 0.02	0.57 ± 0.02	0.61 ± 0.02	0.57 ± 0.02	0.84 ± 0.02
DW GloVe	0.65 ± 0.03	0.65 ± 0.03	0.67 ± 0.03	0.65 ± 0.03	0.89 ± 0.01
DW SG	0.67 ± 0.04	0.67 ± 0.04	0.68 ± 0.03	0.67 ± 0.04	0.90 ± 0.01
Degree SPINE	0.55 ± 0.02	0.55 ± 0.02	0.63 ± 0.02	0.55 ± 0.02	0.84 ± 0.01
Degree WINE	0.46 ± 0.02	0.46 ± 0.02	0.57 ± 0.03	0.46 ± 0.02	0.81 ± 0.01
HOPE	0.51 ± 0.01	0.51 ± 0.01	0.62 ± 0.02	0.51 ± 0.01	0.82 ± 0.01
LINE (1st)	0.58 ± 0.08	0.58 ± 0.08	0.61 ± 0.06	0.58 ± 0.08	0.84 ± 0.04
LINE (2nd)	0.34 ± 0.03	0.33 ± 0.03	0.44 ± 0.05	0.34 ± 0.03	0.70 ± 0.02
Label SPINE	0.28 ± 0.10	0.25 ± 0.12	0.37 ± 0.15	0.28 ± 0.10	0.65 ± 0.09
Label WINE	0.48 ± 0.04	0.50 ± 0.04	0.64 ± 0.03	0.48 ± 0.04	0.81 ± 0.02
NetMF	0.56 ± 0.02	0.57 ± 0.02	0.65 ± 0.02	0.56 ± 0.02	0.86 ± 0.01
Walklets CBOW	0.60 ± 0.02	0.61 ± 0.03	0.64 ± 0.03	0.60 ± 0.02	0.88 ± 0.02
Walklets GloVe	0.58 ± 0.06	0.59 ± 0.06	0.62 ± 0.05	0.58 ± 0.06	0.83 ± 0.03
Walklets SG	0.62 ± 0.06	0.62 ± 0.06	0.64 ± 0.05	0.62 ± 0.06	0.85 ± 0.03

Table I.17: Node-label prediction performance of Random Forest model on different node embedding of the CiteSeer graph The value reported are the average and standard deviations across 10 stratified Monte Carlo holdouts.

Method	Balanced Accuracy	F1 Score	Precision	Recall	AUROC
DW CBOW	0.79 ± 0.01	0.79 ± 0.01	0.79 ± 0.01	0.79 ± 0.01	0.92 ± 0.00
DW GloVe	0.77 ± 0.01	0.77 ± 0.01	0.77 ± 0.00	0.77 ± 0.01	0.91 ± 0.00
DW SG	0.82 ± 0.01	0.81 ± 0.01	0.81 ± 0.00	0.82 ± 0.01	0.93 ± 0.00
Degree SPINE	0.77 ± 0.01	0.77 ± 0.01	0.77 ± 0.01	0.77 ± 0.01	0.91 ± 0.00
Degree WINE	0.63 ± 0.00	0.60 ± 0.01	0.66 ± 0.00	0.63 ± 0.00	0.83 ± 0.00
HOPE	0.78 ± 0.01	0.76 ± 0.01	0.76 ± 0.01	0.78 ± 0.01	0.91 ± 0.00
LINE (1st)	0.71 ± 0.10	0.71 ± 0.09	0.73 ± 0.07	0.71 ± 0.10	0.88 ± 0.05
LINE (2nd)	0.48 ± 0.01	0.48 ± 0.01	0.52 ± 0.01	0.48 ± 0.01	0.69 ± 0.01
Label SPINE	0.26 ± 0.05	0.20 ± 0.04	0.23 ± 0.14	0.26 ± 0.05	0.48 ± 0.03
Label WINE	0.67 ± 0.06	0.66 ± 0.07	0.67 ± 0.07	0.67 ± 0.06	0.80 ± 0.05
NetMF	0.81 ± 0.01	0.80 ± 0.00	0.80 ± 0.00	0.81 ± 0.01	0.93 ± 0.00
Walklets CBOW	0.81 ± 0.01	0.81 ± 0.01	0.81 ± 0.01	0.81 ± 0.01	0.93 ± 0.00
Walklets GloVe	0.55 ± 0.01	0.55 ± 0.01	0.58 ± 0.01	0.55 ± 0.01	0.76 ± 0.01
Walklets SG	0.77 ± 0.02	0.77 ± 0.01	0.78 ± 0.01	0.77 ± 0.02	0.91 ± 0.01

Table I.18: Node-label prediction performance of Random Forest model on different node embedding of the PubMed Diabetes graph The value reported are the average and standard deviations across 10 stratified Monte Carlo holdouts.

I.3 Edge prediction

In this section we succinctly report the average and standard deviations of all the considered node embedding models within the ALPINE edge prediction experiments. In particular, for each node embedding model, we report the performance for all explored edge embedding models.

I.3.1 Perceptron parameters

The edge prediction model considered for the ALPINE edge prediction experiment is a Perceptron. The parameters used to train this perceptron are reported in table I.19.

Parameter	Value
random_state	42
number_of_epochs	100
number_of_edges_per_mini_batch	4096
learning_rate	0.01
first_order_decay_factor	0.9
second_order_decay_factor	0.999
avoid_false_negatives	False
use_scale_free_distribution	True

Table I.19: Parameters used for edge prediction model *Perceptron*

I.3.2 Edge prediction performance

Edge embedding	Accuracy	F1 Score	AUROC	AUPRC
Add	0.50 ± 0.03	0.49 ± 0.17	0.50 ± 0.01	0.55 ± 0.01
Concatenate	0.50 ± 0.02	0.49 ± 0.18	0.50 ± 0.01	0.50 ± 0.02
Cosine	0.87 ± 0.02	0.87 ± 0.03	0.94 ± 0.01	0.95 ± 0.01
Euclidean	0.83 ± 0.04	0.80 ± 0.05	0.89 ± 0.03	0.92 ± 0.02
Hadamard	0.88 ± 0.02	0.87 ± 0.03	0.95 ± 0.01	0.95 ± 0.01
L1	0.87 ± 0.02	0.86 ± 0.03	0.94 ± 0.01	0.95 ± 0.01
L2	0.88 ± 0.02	0.87 ± 0.03	0.94 ± 0.01	0.95 ± 0.01
Maximum	0.87 ± 0.02	0.86 ± 0.03	0.94 ± 0.02	0.95 ± 0.01
Minimum	0.87 ± 0.02	0.86 ± 0.03	0.94 ± 0.02	0.95 ± 0.01
Sub	0.50 ± 0.07	0.46 ± 0.26	0.50 ± 0.00	0.45 ± 0.00

Table I.20: Edge prediction performance of Perceptron model on First-order LINE of the *Homo sapiens* graph The value reported are the average and standard deviations across 10 connected Monte Carlo holdouts.

Edge embedding	Accuracy	F1 Score	AUROC	AUPRC
Add	0.50 ± 0.01	0.45 ± 0.09	0.50 ± 0.00	0.51 ± 0.00
Concatenate	0.50 ± 0.00	0.47 ± 0.08	0.50 ± 0.00	0.50 ± 0.00
Cosine	0.81 ± 0.04	0.79 ± 0.06	0.89 ± 0.03	0.91 ± 0.03
Euclidean	0.59 ± 0.03	0.54 ± 0.05	0.62 ± 0.03	0.65 ± 0.04
Hadamard	0.81 ± 0.04	0.79 ± 0.06	0.89 ± 0.04	0.90 ± 0.04
L1	0.73 ± 0.05	0.69 ± 0.07	0.81 ± 0.04	0.83 ± 0.05
L2	0.72 ± 0.05	0.69 ± 0.08	0.80 ± 0.05	0.82 ± 0.06
Maximum	0.71 ± 0.04	0.68 ± 0.06	0.77 ± 0.05	0.79 ± 0.06
Minimum	0.69 ± 0.04	0.66 ± 0.04	0.75 ± 0.04	0.77 ± 0.05
Sub	0.50 ± 0.00	0.50 ± 0.02	0.50 ± 0.00	0.49 ± 0.00

Table I.21: Edge prediction performance of Perceptron model on DeepWalk CBOW of the *Homo sapiens* graph The value reported are the average and standard deviations across 10 connected Monte Carlo holdouts.

Edge embedding	Accuracy	F1 Score	AUROC	AUPRC
Add	0.50 ± 0.00	0.01 ± 0.05	0.54 ± 0.01	0.56 ± 0.01
Concatenate	0.50 ± 0.00	0.01 ± 0.06	0.53 ± 0.01	0.53 ± 0.02
Cosine	0.61 ± 0.01	0.60 ± 0.02	0.67 ± 0.01	0.67 ± 0.01
Euclidean	0.63 ± 0.01	0.67 ± 0.00	0.70 ± 0.02	0.71 ± 0.02
Hadamard	0.50 ± 0.01	0.23 ± 0.21	0.52 ± 0.02	0.53 ± 0.00
L1	0.68 ± 0.01	0.70 ± 0.01	0.75 ± 0.01	0.74 ± 0.02
L2	0.70 ± 0.01	0.71 ± 0.01	0.76 ± 0.01	0.76 ± 0.01
Maximum	0.57 ± 0.02	0.59 ± 0.03	0.61 ± 0.01	0.63 ± 0.01
Minimum	0.55 ± 0.00	0.44 ± 0.05	0.57 ± 0.02	0.58 ± 0.01
Sub	0.50 ± 0.02	0.51 ± 0.08	0.50 ± 0.00	0.47 ± 0.01

Table I.22: Edge prediction performance of Perceptron model on Degree-based SPINE of the *Homo sapiens* graph The value reported are the average and standard deviations across 10 connected Monte Carlo holdouts.

Edge embedding	Accuracy	F1 Score	AUROC	AUPRC
Add	0.51 ± 0.01	0.41 ± 0.11	0.50 ± 0.01	0.52 ± 0.01
Concatenate	0.50 ± 0.01	0.41 ± 0.11	0.50 ± 0.01	0.50 ± 0.01
Cosine	0.85 ± 0.04	0.83 ± 0.05	0.93 ± 0.01	0.95 ± 0.01
Euclidean	0.69 ± 0.03	0.68 ± 0.03	0.76 ± 0.03	0.79 ± 0.03
Hadamard	0.84 ± 0.03	0.82 ± 0.04	0.92 ± 0.01	0.92 ± 0.01
L1	0.82 ± 0.03	0.79 ± 0.05	0.91 ± 0.02	0.92 ± 0.01
L2	0.80 ± 0.03	0.76 ± 0.05	0.86 ± 0.01	0.85 ± 0.05
Maximum	0.82 ± 0.03	0.79 ± 0.05	0.90 ± 0.02	0.91 ± 0.02
Minimum	0.81 ± 0.03	0.78 ± 0.04	0.89 ± 0.02	0.90 ± 0.01
Sub	0.50 ± 0.02	0.51 ± 0.06	0.50 ± 0.00	0.48 ± 0.01

Table I.23: Edge prediction performance of Perceptron model on DeepWalk SkipGram of the *Homo sapiens* graph The value reported are the average and standard deviations across 10 connected Monte Carlo holdouts.

Edge embedding	Accuracy	F1 Score	AUROC	AUPRC
Add	0.50 ± 0.01	0.24 ± 0.21	0.50 ± 0.02	0.51 ± 0.01
Concatenate	0.50 ± 0.01	0.28 ± 0.19	0.50 ± 0.01	0.50 ± 0.01
Cosine	0.70 ± 0.01	0.69 ± 0.01	0.76 ± 0.02	0.78 ± 0.02
Euclidean	0.62 ± 0.02	0.63 ± 0.01	0.67 ± 0.03	0.70 ± 0.03
Hadamard	0.75 ± 0.01	0.73 ± 0.03	0.82 ± 0.01	0.82 ± 0.01
L1	0.70 ± 0.01	0.69 ± 0.01	0.76 ± 0.01	0.76 ± 0.02
L2	0.69 ± 0.01	0.69 ± 0.01	0.75 ± 0.01	0.76 ± 0.02
Maximum	0.69 ± 0.01	0.69 ± 0.01	0.75 ± 0.01	0.74 ± 0.01
Minimum	0.68 ± 0.01	0.68 ± 0.01	0.75 ± 0.01	0.73 ± 0.01
Sub	0.50 ± 0.02	0.49 ± 0.14	0.50 ± 0.00	0.48 ± 0.01

Table I.24: Edge prediction performance of Perceptron model on DeepWalk GloVe of the *Homo sapiens* graph The value reported are the average and standard deviations across 10 connected Monte Carlo holdouts.

Edge embedding	Accuracy	F1 Score	AUROC	AUPRC
Add	0.50 ± 0.01	0.32 ± 0.31	0.50 ± 0.02	0.54 ± 0.02
Concatenate	0.50 ± 0.01	0.36 ± 0.30	0.50 ± 0.01	0.52 ± 0.02
Cosine	0.75 ± 0.04	0.76 ± 0.04	0.83 ± 0.04	0.86 ± 0.03
Euclidean	0.55 ± 0.01	0.57 ± 0.04	0.58 ± 0.02	0.60 ± 0.01
Hadamard	0.61 ± 0.03	0.57 ± 0.09	0.65 ± 0.04	0.64 ± 0.03
L1	0.61 ± 0.03	0.60 ± 0.07	0.66 ± 0.05	0.64 ± 0.05
L2	0.58 ± 0.02	0.53 ± 0.09	0.62 ± 0.05	0.59 ± 0.05
Maximum	0.54 ± 0.02	0.52 ± 0.16	0.56 ± 0.03	0.55 ± 0.02
Minimum	0.64 ± 0.03	0.58 ± 0.10	0.71 ± 0.04	0.73 ± 0.03
Sub	0.50 ± 0.02	0.49 ± 0.06	0.50 ± 0.00	0.47 ± 0.01

Table I.25: Edge prediction performance of Perceptron model on Degree-based WINE of the *Homo sapiens* graph The value reported are the average and standard deviations across 10 connected Monte Carlo holdouts.

Edge embedding	Accuracy	F1 Score	AUROC	AUPRC
Add	0.50 ± 0.02	0.50 ± 0.06	0.50 ± 0.01	0.54 ± 0.01
Concatenate	0.50 ± 0.02	0.50 ± 0.07	0.50 ± 0.01	0.49 ± 0.01
Cosine	0.84 ± 0.05	0.82 ± 0.08	0.92 ± 0.03	0.93 ± 0.03
Euclidean	0.77 ± 0.09	0.71 ± 0.14	0.83 ± 0.09	0.86 ± 0.08
Hadamard	0.85 ± 0.06	0.82 ± 0.08	0.92 ± 0.04	0.94 ± 0.03
L1	0.83 ± 0.06	0.80 ± 0.09	0.92 ± 0.04	0.93 ± 0.03
L2	0.84 ± 0.06	0.82 ± 0.08	0.92 ± 0.03	0.93 ± 0.03
Maximum	0.82 ± 0.08	0.80 ± 0.10	0.89 ± 0.07	0.90 ± 0.06
Minimum	0.82 ± 0.07	0.80 ± 0.10	0.89 ± 0.07	0.90 ± 0.06
Sub	0.51 ± 0.07	0.49 ± 0.17	0.50 ± 0.00	0.46 ± 0.01

Table I.26: Edge prediction performance of Perceptron model on Second-order LINE of the *Homo sapiens* graph The value reported are the average and standard deviations across 10 connected Monte Carlo holdouts.

Edge embedding	Accuracy	F1 Score	AUROC	AUPRC
Add	0.50 ± 0.02	0.49 ± 0.06	0.50 ± 0.02	0.55 ± 0.01
Concatenate	0.50 ± 0.02	0.49 ± 0.07	0.50 ± 0.01	0.50 ± 0.02
Cosine	0.89 ± 0.02	0.88 ± 0.03	0.95 ± 0.01	0.95 ± 0.01
Euclidean	0.84 ± 0.04	0.81 ± 0.06	0.91 ± 0.03	0.93 ± 0.02
Hadamard	0.89 ± 0.02	0.89 ± 0.03	0.96 ± 0.01	0.96 ± 0.01
L1	0.88 ± 0.03	0.87 ± 0.03	0.95 ± 0.01	0.96 ± 0.01
L2	0.89 ± 0.02	0.88 ± 0.03	0.95 ± 0.01	0.96 ± 0.01
Maximum	0.88 ± 0.03	0.88 ± 0.03	0.95 ± 0.02	0.96 ± 0.01
Minimum	0.88 ± 0.03	0.88 ± 0.03	0.95 ± 0.02	0.96 ± 0.01
Sub	0.52 ± 0.09	0.49 ± 0.21	0.50 ± 0.00	0.45 ± 0.00

Table I.27: Edge prediction performance of Perceptron model on First-order LINE of the *Mus musculus* graph The value reported are the average and standard deviations across 10 connected Monte Carlo holdouts.

Edge embedding	Accuracy	F1 Score	AUROC	AUPRC
Add	0.50 ± 0.01	0.47 ± 0.06	0.50 ± 0.00	0.51 ± 0.00
Concatenate	0.50 ± 0.00	0.47 ± 0.07	0.50 ± 0.00	0.50 ± 0.00
Cosine	0.83 ± 0.04	0.81 ± 0.05	0.90 ± 0.03	0.92 ± 0.03
Euclidean	0.60 ± 0.03	0.57 ± 0.06	0.64 ± 0.04	0.67 ± 0.05
Hadamard	0.83 ± 0.04	0.82 ± 0.05	0.91 ± 0.03	0.91 ± 0.03
L1	0.75 ± 0.05	0.72 ± 0.05	0.82 ± 0.05	0.84 ± 0.05
L2	0.74 ± 0.05	0.72 ± 0.07	0.82 ± 0.05	0.83 ± 0.06
Maximum	0.71 ± 0.04	0.68 ± 0.05	0.78 ± 0.05	0.79 ± 0.05
Minimum	0.72 ± 0.05	0.69 ± 0.05	0.78 ± 0.05	0.80 ± 0.06
Sub	0.50 ± 0.00	0.50 ± 0.01	0.50 ± 0.00	0.49 ± 0.00

Table I.28: Edge prediction performance of Perceptron model on DeepWalk CBOW of the *Mus musculus* graph The value reported are the average and standard deviations across 10 connected Monte Carlo holdouts.

Edge embedding	**Accuracy**	**F1 Score**	**AUROC**	**AUPRC**
Add	0.50 ± 0.00	0.22 ± 0.32	0.51 ± 0.05	0.54 ± 0.04
Concatenate	0.50 ± 0.00	0.23 ± 0.32	0.51 ± 0.04	0.52 ± 0.03
Cosine	0.65 ± 0.01	0.64 ± 0.01	0.72 ± 0.01	0.71 ± 0.01
Euclidean	0.65 ± 0.01	0.68 ± 0.01	0.74 ± 0.02	0.75 ± 0.02
Hadamard	0.51 ± 0.01	0.32 ± 0.20	0.52 ± 0.01	0.53 ± 0.01
L1	0.71 ± 0.01	0.73 ± 0.01	0.79 ± 0.01	0.78 ± 0.02
L2	0.73 ± 0.01	0.74 ± 0.01	0.81 ± 0.01	0.81 ± 0.01
Maximum	0.58 ± 0.01	0.55 ± 0.06	0.62 ± 0.01	0.64 ± 0.01
Minimum	0.56 ± 0.00	0.49 ± 0.02	0.59 ± 0.01	0.59 ± 0.01
Sub	0.50 ± 0.01	0.49 ± 0.06	0.50 ± 0.00	0.47 ± 0.01

Table I.29: Edge prediction performance of Perceptron model on Degree-based SPINE of the *Mus musculus* graph The value reported are the average and standard deviations across 10 connected Monte Carlo holdouts.

Edge embedding	**Accuracy**	**F1 Score**	**AUROC**	**AUPRC**
Add	0.50 ± 0.02	0.43 ± 0.15	0.50 ± 0.01	0.52 ± 0.01
Concatenate	0.50 ± 0.01	0.44 ± 0.14	0.50 ± 0.01	0.50 ± 0.01
Cosine	0.87 ± 0.04	0.85 ± 0.05	0.95 ± 0.01	0.96 ± 0.01
Euclidean	0.73 ± 0.02	0.71 ± 0.03	0.79 ± 0.03	0.82 ± 0.03
Hadamard	0.85 ± 0.03	0.83 ± 0.05	0.94 ± 0.01	0.94 ± 0.00
L1	0.84 ± 0.03	0.82 ± 0.05	0.93 ± 0.02	0.94 ± 0.01
L2	0.81 ± 0.04	0.78 ± 0.06	0.89 ± 0.01	0.87 ± 0.04
Maximum	0.83 ± 0.03	0.81 ± 0.04	0.91 ± 0.02	0.92 ± 0.01
Minimum	0.83 ± 0.03	0.81 ± 0.04	0.92 ± 0.02	0.93 ± 0.01
Sub	0.50 ± 0.01	0.50 ± 0.04	0.50 ± 0.00	0.47 ± 0.00

Table I.30: Edge prediction performance of Perceptron model on DeepWalk SkipGram of the *Mus musculus* graph The value reported are the average and standard deviations across 10 connected Monte Carlo holdouts.

Edge embedding	**Accuracy**	**F1 Score**	**AUROC**	**AUPRC**
Add	0.50 ± 0.01	0.37 ± 0.26	0.50 ± 0.01	0.52 ± 0.01
Concatenate	0.50 ± 0.01	0.36 ± 0.25	0.50 ± 0.01	0.50 ± 0.01
Cosine	0.72 ± 0.01	0.72 ± 0.01	0.79 ± 0.01	0.81 ± 0.01
Euclidean	0.64 ± 0.02	0.65 ± 0.02	0.71 ± 0.02	0.73 ± 0.02
Hadamard	0.78 ± 0.02	0.77 ± 0.02	0.86 ± 0.02	0.85 ± 0.02
L1	0.74 ± 0.01	0.73 ± 0.01	0.82 ± 0.01	0.82 ± 0.01
L2	0.73 ± 0.01	0.73 ± 0.01	0.80 ± 0.01	0.81 ± 0.01
Maximum	0.72 ± 0.01	0.71 ± 0.01	0.79 ± 0.01	0.78 ± 0.01
Minimum	0.72 ± 0.01	0.72 ± 0.01	0.79 ± 0.01	0.78 ± 0.01
Sub	0.50 ± 0.02	0.48 ± 0.10	0.50 ± 0.00	0.48 ± 0.01

Table I.31: Edge prediction performance of Perceptron model on DeepWalk GloVe of the *Mus musculus* graph The value reported are the average and standard deviations across 10 connected Monte Carlo holdouts.

Edge embedding	Accuracy	F1 Score	AUROC	AUPRC
Add	0.50 ± 0.01	0.36 ± 0.30	0.51 ± 0.03	0.54 ± 0.02
Concatenate	0.50 ± 0.02	0.44 ± 0.28	0.49 ± 0.02	0.52 ± 0.03
Cosine	0.79 ± 0.04	0.79 ± 0.04	0.85 ± 0.03	0.88 ± 0.03
Euclidean	0.56 ± 0.02	0.59 ± 0.04	0.60 ± 0.02	0.60 ± 0.01
Hadamard	0.62 ± 0.03	0.61 ± 0.10	0.67 ± 0.05	0.65 ± 0.04
L1	0.62 ± 0.03	0.63 ± 0.07	0.68 ± 0.05	0.65 ± 0.06
L2	0.59 ± 0.03	0.56 ± 0.10	0.63 ± 0.07	0.60 ± 0.07
Maximum	0.54 ± 0.02	0.53 ± 0.16	0.57 ± 0.03	0.56 ± 0.02
Minimum	0.66 ± 0.03	0.60 ± 0.09	0.73 ± 0.04	0.75 ± 0.03
Sub	0.50 ± 0.01	0.51 ± 0.03	0.50 ± 0.00	0.46 ± 0.00

Table I.32: Edge prediction performance of Perceptron model on Degree-based WINE of the *Mus musculus* graph The value reported are the average and standard deviations across 10 connected Monte Carlo holdouts.

Edge embedding	Accuracy	F1 Score	AUROC	AUPRC
Add	0.50 ± 0.02	0.50 ± 0.05	0.50 ± 0.01	0.54 ± 0.02
Concatenate	0.50 ± 0.02	0.50 ± 0.05	0.50 ± 0.01	0.50 ± 0.01
Cosine	0.84 ± 0.06	0.82 ± 0.08	0.93 ± 0.04	0.94 ± 0.03
Euclidean	0.76 ± 0.10	0.70 ± 0.15	0.83 ± 0.09	0.86 ± 0.09
Hadamard	0.85 ± 0.06	0.83 ± 0.09	0.93 ± 0.04	0.94 ± 0.03
L1	0.83 ± 0.07	0.80 ± 0.10	0.93 ± 0.04	0.93 ± 0.03
L2	0.85 ± 0.06	0.82 ± 0.09	0.93 ± 0.03	0.94 ± 0.03
Maximum	0.82 ± 0.08	0.79 ± 0.11	0.89 ± 0.07	0.90 ± 0.07
Minimum	0.82 ± 0.08	0.80 ± 0.11	0.89 ± 0.07	0.90 ± 0.07
Sub	0.49 ± 0.06	0.47 ± 0.14	0.50 ± 0.00	0.46 ± 0.01

Table I.33: Edge prediction performance of Perceptron model on Second-order LINE of the *Mus musculus* graph The value reported are the average and standard deviations across 10 connected Monte Carlo holdouts.

Edge embedding	Accuracy	F1 Score	AUROC	AUPRC
Add	0.49 ± 0.02	0.54 ± 0.04	0.50 ± 0.03	0.54 ± 0.03
Concatenate	0.51 ± 0.02	0.55 ± 0.05	0.50 ± 0.02	0.49 ± 0.03
Cosine	0.87 ± 0.03	0.87 ± 0.04	0.94 ± 0.03	0.94 ± 0.03
Euclidean	0.81 ± 0.05	0.77 ± 0.07	0.88 ± 0.05	0.90 ± 0.04
Hadamard	0.88 ± 0.03	0.87 ± 0.04	0.94 ± 0.03	0.94 ± 0.03
L1	0.87 ± 0.04	0.85 ± 0.04	0.94 ± 0.03	0.94 ± 0.03
L2	0.87 ± 0.04	0.87 ± 0.04	0.94 ± 0.03	0.94 ± 0.03
Maximum	0.86 ± 0.03	0.85 ± 0.04	0.94 ± 0.03	0.94 ± 0.03
Minimum	0.86 ± 0.03	0.85 ± 0.04	0.94 ± 0.03	0.94 ± 0.03
Sub	0.56 ± 0.04	0.62 ± 0.07	0.50 ± 0.00	0.45 ± 0.01

Table I.34: Edge prediction performance of Perceptron model on First-order LINE of the *Saccharomyces cerevisiae* graph The value reported are the average and standard deviations across 10 connected Monte Carlo holdouts.

Edge embedding	**Accuracy**	**F1 Score**	**AUROC**	**AUPRC**
Add	0.49 ± 0.01	0.57 ± 0.02	0.50 ± 0.01	0.51 ± 0.01
Concatenate	0.50 ± 0.01	0.56 ± 0.03	0.50 ± 0.01	0.50 ± 0.01
Cosine	0.84 ± 0.03	0.83 ± 0.04	0.92 ± 0.02	0.93 ± 0.02
Euclidean	0.63 ± 0.02	0.57 ± 0.04	0.65 ± 0.04	0.70 ± 0.04
Hadamard	0.82 ± 0.05	0.80 ± 0.07	0.91 ± 0.04	0.90 ± 0.04
L1	0.77 ± 0.04	0.74 ± 0.06	0.84 ± 0.04	0.85 ± 0.04
L2	0.73 ± 0.08	0.70 ± 0.14	0.83 ± 0.05	0.83 ± 0.05
Maximum	0.73 ± 0.04	0.72 ± 0.04	0.80 ± 0.05	0.80 ± 0.05
Minimum	0.73 ± 0.04	0.72 ± 0.04	0.79 ± 0.05	0.80 ± 0.05
Sub	0.50 ± 0.00	0.51 ± 0.01	0.50 ± 0.00	0.49 ± 0.00

Table I.35: Edge prediction performance of Perceptron model on DeepWalk CBOW of the *Saccharomyces cerevisiae* graph The value reported are the average and standard deviations across 10 connected Monte Carlo holdouts.

Edge embedding	**Accuracy**	**F1 Score**	**AUROC**	**AUPRC**
Add	0.50 ± 0.01	0.37 ± 0.30	0.49 ± 0.04	0.52 ± 0.03
Concatenate	0.50 ± 0.02	0.38 ± 0.29	0.49 ± 0.03	0.50 ± 0.03
Cosine	0.60 ± 0.07	0.66 ± 0.02	0.59 ± 0.20	0.61 ± 0.16
Euclidean	0.73 ± 0.02	0.74 ± 0.01	0.81 ± 0.02	0.81 ± 0.02
Hadamard	0.51 ± 0.03	0.47 ± 0.10	0.53 ± 0.02	0.54 ± 0.00
L1	0.77 ± 0.01	0.78 ± 0.01	0.85 ± 0.01	0.84 ± 0.02
L2	0.78 ± 0.01	0.79 ± 0.01	0.86 ± 0.01	0.85 ± 0.01
Maximum	0.61 ± 0.02	0.63 ± 0.01	0.67 ± 0.01	0.69 ± 0.00
Minimum	0.58 ± 0.01	0.52 ± 0.06	0.63 ± 0.02	0.62 ± 0.01
Sub	0.52 ± 0.01	0.54 ± 0.02	0.50 ± 0.00	0.46 ± 0.01

Table I.36: Edge prediction performance of Perceptron model on Degree-based SPINE of the *Saccharomyces cerevisiae* graph The value reported are the average and standard deviations across 10 connected Monte Carlo holdouts.

Edge embedding	**Accuracy**	**F1 Score**	**AUROC**	**AUPRC**
Add	0.51 ± 0.02	0.48 ± 0.10	0.50 ± 0.01	0.52 ± 0.01
Concatenate	0.50 ± 0.01	0.47 ± 0.10	0.50 ± 0.01	0.51 ± 0.01
Cosine	0.85 ± 0.02	0.83 ± 0.03	0.94 ± 0.01	0.94 ± 0.01
Euclidean	0.64 ± 0.01	0.61 ± 0.02	0.69 ± 0.01	0.72 ± 0.01
Hadamard	0.82 ± 0.02	0.79 ± 0.04	0.90 ± 0.01	0.90 ± 0.01
L1	0.81 ± 0.02	0.79 ± 0.03	0.89 ± 0.01	0.89 ± 0.01
L2	0.74 ± 0.04	0.70 ± 0.08	0.78 ± 0.03	0.77 ± 0.06
Maximum	0.78 ± 0.02	0.76 ± 0.03	0.86 ± 0.01	0.87 ± 0.01
Minimum	0.78 ± 0.02	0.75 ± 0.03	0.86 ± 0.01	0.87 ± 0.01
Sub	0.51 ± 0.00	0.52 ± 0.02	0.50 ± 0.00	0.49 ± 0.00

Table I.37: Edge prediction performance of Perceptron model on DeepWalk SkipGram of the *Saccharomyces cerevisiae* graph The value reported are the average and standard deviations across 10 connected Monte Carlo holdouts.

Edge embedding	**Accuracy**	**F1 Score**	**AUROC**	**AUPRC**
Add	0.50 ± 0.01	0.36 ± 0.15	0.50 ± 0.02	0.52 ± 0.02
Concatenate	0.50 ± 0.02	0.48 ± 0.08	0.50 ± 0.02	0.50 ± 0.02
Cosine	0.68 ± 0.04	0.69 ± 0.04	0.75 ± 0.04	0.74 ± 0.05
Euclidean	0.61 ± 0.03	0.62 ± 0.04	0.65 ± 0.05	0.66 ± 0.04
Hadamard	0.73 ± 0.02	0.73 ± 0.01	0.81 ± 0.02	0.78 ± 0.02
L1	0.73 ± 0.02	0.74 ± 0.01	0.80 ± 0.02	0.77 ± 0.03
L2	0.72 ± 0.02	0.74 ± 0.01	0.79 ± 0.02	0.77 ± 0.02
Maximum	0.72 ± 0.01	0.72 ± 0.01	0.78 ± 0.01	0.76 ± 0.02
Minimum	0.71 ± 0.01	0.72 ± 0.01	0.78 ± 0.01	0.75 ± 0.01
Sub	0.51 ± 0.01	0.57 ± 0.04	0.50 ± 0.00	0.48 ± 0.01

Table I.38: Edge prediction performance of Perceptron model on DeepWalk GloVe of the *Saccharomyces cerevisiae* graph The value reported are the average and standard deviations across 10 connected Monte Carlo holdouts.

Edge embedding	**Accuracy**	**F1 Score**	**AUROC**	**AUPRC**
Add	0.50 ± 0.02	0.44 ± 0.24	0.50 ± 0.02	0.56 ± 0.04
Concatenate	0.50 ± 0.02	0.46 ± 0.23	0.49 ± 0.02	0.55 ± 0.04
Cosine	0.80 ± 0.02	0.80 ± 0.02	0.85 ± 0.02	0.86 ± 0.01
Euclidean	0.59 ± 0.05	0.62 ± 0.05	0.65 ± 0.04	0.64 ± 0.02
Hadamard	0.66 ± 0.05	0.61 ± 0.13	0.70 ± 0.08	0.68 ± 0.07
L1	0.67 ± 0.05	0.66 ± 0.06	0.75 ± 0.07	0.72 ± 0.07
L2	0.62 ± 0.05	0.59 ± 0.09	0.68 ± 0.07	0.65 ± 0.07
Maximum	0.54 ± 0.03	0.59 ± 0.10	0.56 ± 0.05	0.55 ± 0.03
Minimum	0.65 ± 0.06	0.57 ± 0.15	0.70 ± 0.10	0.73 ± 0.09
Sub	0.52 ± 0.02	0.54 ± 0.04	0.50 ± 0.00	0.45 ± 0.00

Table I.39: Edge prediction performance of Perceptron model on Degree-based WINE of the *Saccharomyces cerevisiae* graph The value reported are the average and standard deviations across 10 connected Monte Carlo holdouts.

Edge embedding	**Accuracy**	**F1 Score**	**AUROC**	**AUPRC**
Add	0.49 ± 0.02	0.53 ± 0.05	0.49 ± 0.03	0.53 ± 0.02
Concatenate	0.50 ± 0.03	0.53 ± 0.05	0.49 ± 0.02	0.49 ± 0.01
Cosine	0.86 ± 0.04	0.84 ± 0.05	0.93 ± 0.02	0.94 ± 0.02
Euclidean	0.78 ± 0.05	0.73 ± 0.08	0.85 ± 0.05	0.88 ± 0.04
Hadamard	0.86 ± 0.04	0.84 ± 0.05	0.94 ± 0.02	0.94 ± 0.02
L1	0.85 ± 0.04	0.83 ± 0.05	0.94 ± 0.02	0.94 ± 0.02
L2	0.86 ± 0.04	0.84 ± 0.05	0.94 ± 0.02	0.94 ± 0.02
Maximum	0.84 ± 0.04	0.82 ± 0.05	0.92 ± 0.03	0.92 ± 0.03
Minimum	0.84 ± 0.04	0.82 ± 0.05	0.92 ± 0.03	0.92 ± 0.03
Sub	0.53 ± 0.03	0.58 ± 0.05	0.50 ± 0.00	0.46 ± 0.01

Table I.40: Edge prediction performance of Perceptron model on Second-order LINE of the *Saccharomyces cerevisiae* graph The value reported are the average and standard deviations across 10 connected Monte Carlo holdouts.

Bibliography

[1] Louis Abraham. *fastnode2vec*. 2020. DOI: `10.5281/zenodo.3902632`. URL: `https://doi.org/10.5281/zenodo.3902632`.

[2] Nesreen K Ahmed et al. "role2vec: Role-based network embeddings". In: *Proc. DLG KDD* (2019), pp. 1–7.

[3] Mehdi Ali et al. "PyKEEN 1.0: A Python Library for Training and Evaluating Knowledge Graph Embeddings". In: *Journal of Machine Learning Research* 22.82 (2021), pp. 1–6. URL: `http://jmlr.org/papers/v22/20-825.html`.

[4] Katrin Amunts et al. "BigBrain: An Ultrahigh-Resolution 3D Human Brain Model". In: *Science* 340.6139 (2013), pp. 1472–1475.

[5] Edward Anderson et al. *LAPACK users' guide.* SIAM, 1999.

[6] Sören Auer et al. "Dbpedia: A nucleus for a web of open data". In: *The semantic web.* Springer, 2007, pp. 722–735.

[7] Lars Backstrom et al. "Four degrees of separation". In: *Proceedings of the 4th Annual ACM Web Science Conference.* 2012, pp. 33–42.

[8] David A Bader and Kamesh Madduri. "Parallel algorithms for evaluating centrality indices in real-world networks". In: *2006 International Conference on Parallel Processing (ICPP'06).* IEEE. 2006, pp. 539–550.

[9] Bahman Bahmani, Kaushik Chakrabarti, and Dong Xin. "Fast personalized pagerank on mapreduce". In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data.* 2011, pp. 973–984.

[10] Mukesh Bansal et al. "How to infer gene networks from expression profiles". In: *Molecular systems biology* 3.1 (2007).

[11] Mikhail Belkin and Partha Niyogi. "Laplacian eigenmaps for dimensionality reduction and data representation". In: *Neural computation* 15.6 (2003), pp. 1373–1396.

[12] James Bergstra et al. "Algorithms for hyper-parameter optimization". In: *Advances in neural information processing systems* 24 (2011).

[13] David Blackman and Sebastiano Vigna. "Scrambled linear pseudorandom number generators". In: *arXiv preprint arXiv:1805.01407* (2018).

[14] The Editorial Board. "Shifting habitats". In: *Nature Climate Change* 10.5 (May 2020), pp. 377–377. ISSN: 1758-6798. DOI: `10.1038/s41558-020-0789-x`. URL: `https://doi.org/10.1038/s41558-020-0789-x`.

[15] Paolo Boldi et al. "Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks". In: *Proceedings of the 20th international conference on World Wide Web*. Ed. by Sadagopan Srinivasan et al. ACM Press, 2011, pp. 587–596.

[16] Antoine Bordes et al. "Translating embeddings for modeling multi-relational data". In: *Advances in neural information processing systems* 26 (2013).

[17] Dhruba Borthakur. "The hadoop distributed file system: Architecture and design". In: *Hadoop Project Website* 11.2007 (2007), p. 21.

[18] Aydin Buluç and Kamesh Madduri. "Parallel breadth-first search on distributed memory systems". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 2011, pp. 1–12.

[19] Aydin Buluç et al. "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks". In: *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. 2009, pp. 233–244.

[20] Federico Busato et al. "Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus". In: *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE. 2018, pp. 1–7.

[21] Andrey Bychkov and Vsevolod Nikolskiy. "Rust Language for Supercomputing Applications". In: *Russian Supercomputing Days*. Springer. 2021, pp. 391–403.

[22] TJ Callahan. *PheKnowLator*. 2019. DOI: `10.5281/zenodo.3401437`. URL: `https://doi.org/10.5281/zenodo.3401437`.

[23] TJ Callahan et al. "A Framework for Automated Construction of Heterogeneous Large-Scale Biomedical Knowledge Graphs". In: *Bio-Ontologies COSI - Intelligent Systems for Molecular Biology*. 2020. DOI: `10.1101/2020.04.30.071407`.

[24] Shaosheng Cao, Wei Lu, and Qiongkai Xu. "Grarep: Learning graph representations with global structural information". In: *Proceedings of the 24th ACM international on conference on information and knowledge management*. 2015, pp. 891–900.

[25] Chen, Shen Wei. *GraphEmbedding*. `https://github.com/shenweichen/GraphEmbedding`. Online; accessed 21 July 2021. 2021.

[26] Ara Cho et al. "WormNet v3: a network-assisted hypothesis-generating server for Caenorhabditis elegans". In: *Nucleic acids research* 42.W1 (2014), W76–W82.

[27] Fan RK Chung. *Spectral graph theory*. Vol. 92. American Mathematical Soc., 1997.

[28] Simona Cocco, Remi Monasson, and Martin Weigt. "From principal component to direct coupling analysis of coevolution in proteins: Low-eigenvalue modes are needed for structure prediction". In: *PLoS computational biology* 9.8 (2013), e1003176.

[29] Gene Ontology Consortium. "The Gene Ontology (GO) database and informatics resource". In: *Nucleic acids research* 32.suppl_1 (2004), pp. D258–D261.

[30] G. A. Constantinides, P. Y. K. Cheung, and W. Luk. "Synthesis of Saturation Arithmetic Architectures". In: *ACM Trans. Des. Autom. Electron. Syst.* 8.3 (July 2003), pp. 334–354. ISSN: 1084-4309. DOI: `10.1145/785411.785415`. URL: `https://doi.org/10.1145/785411.785415`.

[31] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2022.

[32] Gabor Csardi and Tamas Nepusz. "The igraph software package for complex network research". In: *InterJournal* Complex Systems (2006), p. 1695. URL: `https://igraph.org`.

[33] Allan Peter Davis et al. "Comparative Toxicogenomics Database (CTD): update 2021". In: *Nucleic Acids Research* 49.D1 (Oct. 2020), pp. D1138–D1143. ISSN: 0305-1048. DOI: `10.1093/nar/gkaa891`. eprint: `https://academic.oup.com/nar/article-pdf/49/D1/D1138/35364751/gkaa891.pdf`. URL: `https://doi.org/10.1093/nar/gkaa891`.

[34] Wouter De Nooy, Andrej Mrvar, and Vladimir Batagelj. *Exploratory social network analysis with Pajek*. Cambridge University Press, 2011.

[35] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[36] Jack Dongarra. “Report on the Fujitsu Fugaku system”. In: *University of Tennessee-Knoxville Innovative Computing Laboratory, Tech. Rep. ICLUT-20-06* (2020).

[37] J. Duch and A. Arenas. “Community identification using Extremal Optimization Phys”. In: *Rev. E* 72 (2005), p. 027104.

[38] Peter Elias. “Efficient storage and retrieval by content and address of static files”. In: *Journal of the ACM (JACM)* 21.2 (1974), pp. 246–260.

[39] Peter Elias. “On binary representations of monotone sequences”. In: *Proc. sixth princeton conference on information sciences and systems.* 1972, pp. 54–57.

[40] Peter Elias. “Universal codeword sets and representations of the integers”. In: *IEEE transactions on information theory* 21.2 (1975), pp. 194–203.

[41] Federico Errica et al. “A fair comparison of graph neural networks for graph classification”. In: *Proceedings of the 8th International Conference on Learning Representations (ICLR).* 2020.

[42] Jason Evans. “A scalable concurrent malloc (3) implementation for FreeBSD”. In: *Proc. of the bsdcan conference, ottawa, canada.* 2006.

[43] Matthias Fey and Jan E. Lenssen. “Fast Graph Representation Learning with PyTorch Geometric”. In: *ICLR Workshop on Representation Learning on Graphs and Manifolds.* 2019.

[44] Steven Fortune and James Wyllie. “Parallelism in random access machines”. In: *Proceedings of the tenth annual ACM symposium on Theory of computing.* 1978, pp. 114–118.

[45] Jessie Frazelle. “Chip measuring contest: The benefits of purpose-built chips”. In: *Queue* 19.5 (2021), pp. 5–21.

[46] Jean-loup Gailly and Mark Adler. “GNU gzip”. In: *GNU Operating System* (1992).

[47] Erich Gamma et al. *Design Patterns: Elements of reusable object-oriented software.* Addison-Wesley Reading, Massachusetts, 1995.

[48] Lise Getoor. “Link-based classification”. In: *Advanced methods for knowledge discovery from complex data.* Springer, 2005, pp. 189–207.

[49] Kwang-Il Goh et al. “The human disease network”. In: *Proceedings of the National Academy of Sciences* 104.21 (2007), pp. 8685–8690.

[50] J. Gonzalez et al. “GraphX: Graph Processing in a Distributed Dataflow Framework”. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14).* Broomfield, CO: USENIX Association, 2014, pp. 599–613. URL: `https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez`.

[51] Daniele Grattarola and Cesare Alippi. “Graph neural networks in TensorFlow and keras with spektral”. In: *IEEE Computational Intelligence Magazine* 16.1 (2021), pp. 99–106.

[52] Aditya Grover and Jure Leskovec. “node2vec: Scalable feature learning for networks”. In: *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining.* 2016, pp. 855–864.

[53] Thomas R Gruber. “A translation approach to portable ontology specifications”. In: *Knowledge acquisition* 5.2 (1993), pp. 199–220.

[54] Nicola Guarino and Pierdaniele Giaretta. “Ontologies and knowledge bases”. In: *Towards very large knowledge bases* (1995), pp. 1–2.

[55] A. Hagberg, D. Schult, and P. Swart. “Exploring Network Structure, Dynamics, and Function using NetworkX”. In: *Proceedings of the 7th Python in Science Conference, Pasadena, CA USA.* 2008.

[56] Aric Hagberg, Pieter Swart, and Daniel S Chult. *Exploring network structure, dynamics, and function using NetworkX*. Tech. rep. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[57] William L Hamilton. "Graph representation learning". In: *Synthesis Lectures on Artifical Intelligence and Machine Learning* 14.3 (2020), pp. 1–159.

[58] Weihua Hu et al. "Open graph benchmark: Datasets for machine learning on graphs". In: *Advances in neural information processing systems* 33 (2020), pp. 22118–22133.

[59] Jung-Chang Huang and Tau Leng. "Generalized loop-unrolling: a method for program speedup". In: *Proceedings 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology. ASSET'99 (Cat. No. PR00122)*. IEEE. 1999, pp. 244–248.

[60] Paul Jaccard. "The distribution of the flora in the alpine zone. 1". In: *New phytologist* 11.2 (1912), pp. 37–50.

[61] H. Jeong et al. "Lethality and centrality in protein networks". In: *arXiv preprint cond-mat/0105306* (2001).

[62] Rudolf Kadlec, Ondrej Bajgar, and Jan Kleindienst. "Knowledge Base Completion: Baselines Strike Back". In: *Proceedings of the 2nd Workshop on Representation Learning for NLP*. Vancouver, Canada: Association for Computational Linguistics, Aug. 2017, pp. 69–74. DOI: 10.18653/v1/W17-2609. URL: https://aclanthology.org/W17-2609.

[63] Donald Knuth. "Seminumerical algorithms". In: *The art of computer programming* 2 (1981).

[64] Sebastian Köhler et al. "The Human Phenotype Ontology in 2021". In: *Nucleic Acids Research* 49.D1 (Dec. 2020), pp. D1207–D1217. ISSN: 0305-1048. DOI: 10.1093/nar/gkaa1043. eprint: https://academic.oup.com/nar/article-pdf/49/D1/D1207/35364524/gkaa1043.pdf. URL: https://doi.org/10.1093/nar/gkaa1043.

[65] Richard A Kronmal and Arthur V Peterson Jr. "On the alias method for generating random variables from a discrete distribution". In: *The American Statistician* 33.4 (1979), pp. 214–218.

[66] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. "Numba: A llvm-based python jit compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 2015, pp. 1–6.

[67] Daniel Lee and H Sebastian Seung. "Algorithms for non-negative matrix factorization". In: *Advances in neural information processing systems* 13 (2000).

[68] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. "SIMD compression and the intersection of sorted integers". In: *Software: Practice and Experience* 46.6 (2016), pp. 723–749.

[69] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of massive data sets*. Cambridge university press, 2020.

[70] Jure Leskovec and Rok Sosič. "SNAP: A General-Purpose Network Analysis and Graph-Mining Library". In: *ACM Transactions on Intelligent Systems and Technology (TIST)* 8.1 (2016), p. 1.

[71] Zhiyuan Lin et al. "Mmap: Fast billion-scale graph computation on a pc via memory mapping". In: *2014 IEEE International Conference on Big Data (Big Data)*. IEEE. 2014, pp. 159–164.

[72] Renming Liu and Arjun Krishnan. "PecanPy: a fast, efficient and parallelized Python implementation of node2vec". In: *Bioinformatics* 37.19 (2021), pp. 3377–3379.

[73] Renming Liu et al. "Supervised learning is an accurate method for network-based gene classification". In: *Bioinformatics* 36.11 (2020), pp. 3457–3465.

[74] Yucheng Low et al. "GraphLab: A New Framework for Parallel Machine Learning". In: *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*. UAI'10. Catalina Island, CA: AUAI Press, 2010, pp. 340–349.

[75] William R. Mark et al. "Cg: A System for Programming Graphics Hardware in a C-like Language". In: *ACM SIGGRAPH 2003 Papers*. SIGGRAPH '03. San Diego, California: Association for Computing Machinery, 2003, pp. 896–907. ISBN: 1581137095. DOI: 10.1145/1201775.882362. URL: https://doi.org/10.1145/1201775.882362.

[76] George Marsaglia. "Xorshift RNGs". In: *Journal of Statistical Software, Articles* 8.14 (2003), pp. 1–6. ISSN: 1548-7660. DOI: 10.18637/jss.v008.i14. URL: https://www.jstatsoft.org/v008/i14.

[77] V. Martinez, F. Berzal, and JC Cubero. "A survey of link prediction in complex networks". In: *ACM Computing Surveys* 49.6 (2017). DOI: 10.1145/3012704.

[78] Ulrich Meyer and Peter Sanders. "Parallel shortest path for arbitrary graphs". In: *European Conference on Parallel Processing*. Springer. 2000, pp. 461–470.

[79] Tomas Mikolov et al. "Distributed Representations of Words and Phrases and their Compositionality". In: *Advances in Neural Information Processing Systems*. Ed. by C. J. C. Burges et al. Vol. 26. Curran Associates, Inc., 2013. URL: https://proceedings.neurips.cc/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf.

[80] Tomas Mikolov et al. "Efficient Estimation of Word Representations in Vector Space". In: *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2013. URL: http://arxiv.org/abs/1301.3781.

[81] Tomas Mikolov et al. "Efficient estimation of word representations in vector space". In: *arXiv preprint arXiv:1301.3781* (2013).

[82] Alan Mislove et al. "Growth of the flickr social network". In: *Proceedings of the first workshop on Online social networks*. ACM. 2008, pp. 25–30.

[83] Christopher J Mungall et al. "The Monarch Initiative: an integrative data and analytic platform connecting phenotypes to genotypes across species". In: *Nucleic acids research* 45.D1 (2017), pp. D712–D722.

[84] Galileo Namata et al. "Query-driven active surveying for collective classification". In: *10th International Workshop on Mining and Learning with Graphs*. Vol. 8. 2012.

[85] Gonzalo Navarro and Eliana Providel. "Fast, small, simple rank/select on bitmaps". In: *International Symposium on Experimental Algorithms*. Springer. Springer, 2012, pp. 295–306.

[86] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation". In: *ACM Sigplan notices* 42.6 (2007), pp. 89–100.

[87] Natalya F Noy, Deborah L McGuinness, et al. *Ontology development 101: A guide to creating your first ontology*. 2001.

[88] Nvidia Nvidia. *H100 Tensor Core GPU Architecture*. 2022.

[89] Mingdong Ou et al. "Asymmetric Transitivity Preserving Graph Embedding". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 1105–1114. ISBN: 9781450342322.

DOI: 10.1145/2939672.2939751. URL: https://doi.org/10.1145/2939672.2939751.

[90] Mingdong Ou et al. “Asymmetric transitivity preserving graph embedding”. In: *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 2016, pp. 1105–1114.

[91] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[92] Jeffrey Pennington, Richard Socher, and Christopher D Manning. “Glove: Global vectors for word representation”. In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 1532–1543.

[93] Jeffrey M Perkel. “Why scientists are turning to Rust”. In: *Nature* 588.7836 (2020), pp. 185–186.

[94] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. “Deepwalk: Online learning of social representations”. In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2014, pp. 701–710.

[95] Bryan Perozzi et al. “Don’t Walk, Skip! Online Learning of Multi-Scale Network Embeddings”. In: *Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017*. ASONAM ’17. Sydney, Australia: Association for Computing Machinery, 2017, pp. 258–265. ISBN: 9781450349932. DOI: 10.1145/3110025.3110086. URL: https://doi.org/10.1145/3110025.3110086.

[96] Bryan Perozzi et al. “Don’t walk, skip! online learning of multi-scale network embeddings”. In: *Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017*. 2017, pp. 258–265.

[97] Giulio Ermanno Pibiri. “Dynamic Elias-Fano Encoding”. PhD thesis. Master’s Thesis, University of Pisa, Pisa, Italy, 2014.

[98] Giulio Ermanno Pibiri and Rossano Venturini. “Dynamic Elias-Fano Representation”. In: *28th Annual symposium on combinatorial pattern matching (CPM 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017.

[99] Jiezhong Qiu et al. “Network embedding as matrix factorization: Unifying deepwalk, line, pte, and node2vec”. In: *Proceedings of the eleventh ACM international conference on web search and data mining*. 2018, pp. 459–467.

[100] Ranger, Matt. *CSRGraph*. https://github.com/VHRanger/CSRGraph. Online; accessed 21 July 2021. 2021.

[101] Justin T. Reese et al. “KG-COVID-19: A Framework to Produce Customized Knowledge Graphs for COVID-19 Response”. In: *Patterns* 2.1 (Jan. 2021), p. 100155. ISSN: 2666-3899. URL: https://www.sciencedirect.com/science/article/pii/S2666389920302038.

[102] Radim Řehůřek and Petr Sojka. “Software Framework for Topic Modelling with Large Corpora”. English. In: *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. http://is.muni.cz/publication/884893/en. Valletta, Malta: ELRA, May 2010, pp. 45–50.

[103] Peter N Robinson and S Mundlos. “The human phenotype ontology”. In: *Clinical genetics* 77.6 (2010), pp. 525–534.

[104] Ryan Rossi and Nesreen Ahmed. “The network data repository with interactive graph analytics and visualization”. In: *Twenty-ninth AAAI conference on artificial intelligence*. 2015.

[105] Ryan A. Rossi and Nesreen K. Ahmed. “The Network Data Repository with Interactive Graph Analytics and Visualization”. In: *AAAI*. 2015. URL: http://networkrepository.com.

[106] Benedek Rozemberczki, Oliver Kiss, and Rik Sarkar. “Karate Club: An API Oriented Open-source Python Framework for Unsupervised Learning on Graphs”. In: *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM ’20)*. ACM. 2020, pp. 3125–3132.

[107] Yousef Saad. “Parallel iterative methods for sparse linear systems”. In: *Studies in Computational Mathematics*. Vol. 8. Elsevier, 2001, pp. 423–440.

[108] Michel F Sanner et al. “Python: a programming language for software integration and development”. In: *J Mol Graph Model* 17.1 (1999), pp. 57–61.

[109] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. “Evaluating the cost of atomic operations on modern architectures”. In: *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE. 2015, pp. 445–456.

[110] Prithviraj Sen et al. “Collective classification in network data”. In: *AI magazine* 29.3 (2008), pp. 93–93.

[111] Nino Shervashidze et al. “Weisfeiler-lehman graph kernels.” In: *Journal of Machine Learning Research* 12.9 (2011).

[112] Rohit Singh, Jinbo Xu, and Bonnie Berger. “Global alignment of multiple protein interaction networks with application to functional orthology detection”. In: *PNAS* 105.35 (2008), pp. 12763–12768.

[113] Damian Smedley et al. “PhenoDigm: analyzing curated annotations to associate animal models with human diseases”. en. In: *Database (Oxford)* 2013 (May 2013), bat025.

[114] Apache Spark. “Apache spark”. In: *Retrieved January* 17.1 (2018), p. 2018.

[115] Chris Stark et al. “BioGRID: a general repository for interaction datasets”. In: *Nucleic acids research* 34.suppl_1 (2006), pp. D535–D539.

[116] Dennis L Sun and Cedric Fevotte. “Alternating direction method of multipliers for non-negative matrix factorization with the beta-divergence”. In: *2014 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE. 2014, pp. 6201–6205.

[117] Damian Szklarczyk et al. “STRING v11: protein–protein association networks with increased coverage, supporting functional discovery in genome-wide experimental datasets”. In: *Nucleic Acids Research* 47.D1 (2018), pp. D607–D613. ISSN: 0305-1048. DOI: `10.1093/nar/gky1131`. eprint: `https://academic.oup.com/nar/article-pdf/47/D1/D607/27437323/gky1131.pdf`. URL: `https://doi.org/10.1093/nar/gky1131`.

[118] Damian Szklarczyk et al. “The STRING database in 2021: customizable protein–protein networks, and functional characterization of user-uploaded gene/measurement sets”. In: *Nucleic acids research* 49.D1 (2021), pp. D605–D612.

[119] Jian Tang et al. “LINE: Large-scale Information Network Embedding.” In: *WWW*. ACM. 2015.

[120] Lei Tang and Huan Liu. “Relational learning via latent social dimensions”. In: *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2009, pp. 817–826.

[121] Sara Tomiolo and David Ward. “Species migrations and range shifts: A synthesis of causes and consequences”. In: *Perspectives in plant ecology, evolution and systematics* 33 (2018), pp. 62–77.

[122] Leo Torres, Kevin S Chan, and Tina Eliassi-Rad. “GLEE: Geometric Laplacian eigenmap embedding”. In: *Journal of Complex Networks* 8.2 (2020), cnaa007.

[123] Deepak R Unni et al. “Biolink Model: A universal schema for knowledge graphs in clinical, biomedical, and translational science”. In: *Clinical and Translational Science* (2022).

[124] Laurens Van der Maaten and Geoffrey Hinton. “Visualizing data using t-SNE.” In: *Journal of machine learning research* 9.11 (2008).

[125] Sebastiano Vigna. “Quasi-succinct indices”. In: *Proceedings of the sixth ACM international conference on Web search and data mining*. 2013, pp. 83–92.

[126] Michael D Vose. “A linear algorithm for generating random numbers with a given distribution”. In: *IEEE Transactions on software engineering* 17.9 (1991), pp. 972–975.

[127] Minjie Wang et al. “Deep graph library: Towards efficient and scalable deep learning on graphs”. In: *ICLR workshop on representation learning on graphs and manifolds*. 2019.

[128] Naigang Wang et al. “Training deep neural networks with 8-bit floating point numbers”. In: *Advances in neural information processing systems* 31 (2018).

[129] Max Welling and Thomas N Kipf. “Semi-supervised classification with graph convolutional networks”. In: *J. International Conference on Learning Representations (ICLR 2017)*. 2016.

[130] Andy B Yoo, Morris A Jette, and Mark Grondona. “Slurm: Simple linux utility for resource management”. In: *Workshop on job scheduling strategies for parallel processing*. Springer. 2003, pp. 44–60.

[131] D Zhang et al. “Network Representation Learning: A Survey.” In: *IEEE Transactions on Big Data* 1 (2020), pp. 3–28.

[132] Denghui Zhang et al. “Efficient parallel translating embedding for knowledge graphs”. In: *Proceedings of the International Conference on Web Intelligence*. 2017, pp. 460–468.

[133] Ziwei Zhang et al. “Billion-scale network embedding with iterative random projection”. In: *2018 IEEE International Conference on Data Mining (ICDM)*. IEEE. 2018, pp. 787–796.