REORDERING GRAPHS FOR NODE EMBEDDING

by
BERKAY DEMİRELLER

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfilment of
the requirements for the degree of Master of Science

Sabancı University
July 2024

REORDERING GRAPHS FOR NODE EMBEDDING

Approved by:

Assoc. Prof. Kamer Kaya .....
(Thesis Supervisor)

Prof. Hüsnü Yenigün .....

Assoc. Prof. Didem Unat .....

Date of Approval:

ABSTRACT

REORDERING GRAPHS FOR NODE EMBEDDING

BERKAY DEMİRELLER

COMPUTER SCIENCE AND ENGINEERING M.Sc. THESIS, JULY 2024

Thesis Supervisor: Assoc. Prof. KAMER KAYA

Keywords: Graphs, Matrix Reordering, Node Embedding, GPU

In today's interconnected world, graphs are widely used to model complex relationships and structures across various domains, from social networks and transportation systems to biological networks and recommendation systems. However, the high dimensionality and intricate connectivity of these graphs pose significant challenges for analysis and processing. Node embedding techniques have emerged as powerful tools to address these challenges by transforming graph nodes into low-dimensional vectors while preserving the inherent structural properties and relationships of the original graph. Despite their effectiveness, node embedding can be an expensive process, particularly for large-scale graphs, due to the substantial computational resources and time required. This thesis aims to improve node embedding frameworks that utilize GPUs by reordering matrices that represent graphs. We propose a probabilistic part-skipping strategy on reordered graphs that eliminates the overhead created by moving parts of the graph into and out of the GPU memory and therefore speeding up the process significantly. The resulting embeddings perform as well as embeddings learned on a randomly ordered graph and in some cases perform significantly better on link prediction tasks. We also present link prediction results after reordering on various graphs obtained from *SuiteSparse* and *The Network Repository*. The results show that the class of reordering algorithms that emphasize the connectivity structure and community information found within the graphs improve the link prediction results regardless of the graph type used.

ÖZET

DÜĞÜM GÖMME İÇİN ÇİZGELERİ YENİDEN SIRALAMA

BERKAY DEMİRELLER

BİLGİSAYAR BİLİMİ VE MÜHENDİSLİĞİ YÜKSEK LİSANS TEZİ, TEMMUZ 2024

Tez Danışmanı: Doç. Dr. KAMER KAYA

Anahtar Kelimeler: Çizgeler, Matris Düzenleme, Düğüm Gömme, GPU

Günümüzün birbirine bağlı dünyasında, çizgeler sosyal ağlardan ulaşım sistemlerine, biyolojik ağlardan öneri sistemlerine kadar çeşitli alanlarda karmaşık ilişkileri ve yapıları modellemek için yaygın olarak kullanılmaktadır. Ancak, bu çizgelerin yüksek boyutluluğu ve karmaşık bağlantıları, analiz ve işleme konusunda önemli zorluklar yaratmaktadır. Düğüm gömme teknikleri, çizge düğümlerini düşük boyutlu vektörlere dönüştürerek orijinal çizgenin yapısal özelliklerini ve ilişkilerini korurken bu zorlukların üstesinden gelmek için güçlü araçlar olarak ortaya çıkmıştır. Etkili olmalarına rağmen, düğüm gömme işlemi, büyük ölçekli çizgeler için önemli hesaplama kaynakları ve zaman gerektirdiğinden pahalı bir süreç olabilir. Bu tez, çizgeleri temsil eden matrislerin yeniden düzenlenmesiyle GPU'ları kullanan düğüm gömme çerçevelerini iyileştirmeyi amaçlamaktadır. Çizgelerin yeniden düzenlendiği olasılıksal bir parça atlama stratejisi öneriyoruz; bu strateji, çizgenin parçalarının GPU belleğine taşınması ve geri alınması sırasında oluşan yükü ortadan kaldırarak süreci önemli ölçüde hızlandırmaktadır. Ortaya çıkan gömmeler, rastgele sıralanmış bir çizgede öğrenilen gömmeler kadar iyi performans göstermekte ve bazı durumlarda bağlantı tahmini görevlerinde önemli ölçüde daha iyi performans göstermektedir. Ayrıca, SuiteSparse ve The Network Repository'den elde edilen çeşitli çizgelerde yeniden düzenleme sonrası bağlantı tahmini sonuçlarını sunuyoruz. Sonuçlar, çizgeler içinde bulunan bağlantı yapısı ve topluluk bilgilerini vurgulayan yeniden düzenleme algoritmaları sınıfının, kullanılan çizge türünden bağımsız olarak bağlantı tahmini sonuçlarını iyileştirdiğini göstermektedir.

ACKNOWLEDGEMENTS

I would like to sincerely thank my family and friends for their unending support, to my uncle Mustafa for always having a place for me in his prayers, to my professor Kamer Hoca for his guidance and for making this journey enjoyable, and to my friend Beyza who offered her valuable time when I needed it.

Dedicated to Mustafa Demireller
Ruhu şad olsun.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

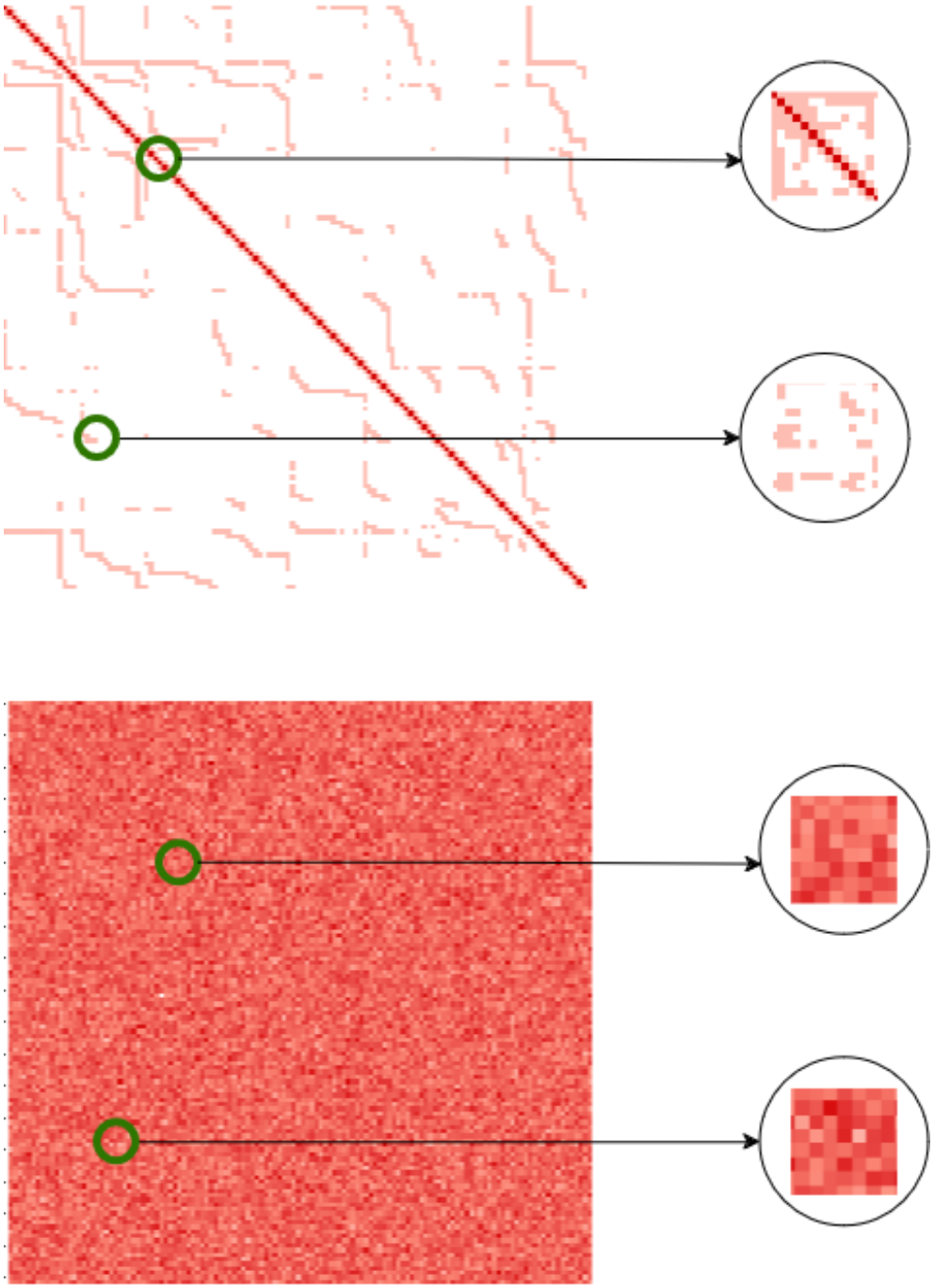
1. INTRODUCTION

Graphs are becoming more and more relevant in today's world. In social networks, graphs represent individuals as nodes and their interactions as edges, facilitating analysis of community structures and influence patterns. In transportation systems, graphs model cities as nodes and roads or routes as edges, optimizing navigation and traffic management. Biological networks use graphs to depict molecular interactions, aiding in the understanding of cellular processes and the development of medical treatments. The use cases are endless and this leads to extensive research on graph processing and analysis. The inherent problem with graph analysis is that graphs are high dimensional structures which makes them very hard to analyse efficiently and accurately. Node embedding is a technique that helps with graph analysis which aims to learn how to represent nodes of a graph with lower dimensional vectors. This technique presents its own set of challenges, it is hard to try and learn how to accurately represent nodes and their connectivity as is and when the constraints of computational resources are factored in the need for research on how to learn node embeddings faster while staying accurate becomes apparent.

Graph/matrix (re)ordering is a technique used to rearrange the rows and columns of a matrix, which can be an adjacency matrix of a graph, to achieve a more desirable structure. This can enhance computational efficiency, improve numerical stability, and optimize memory usage in various mathematical and computational applications. Matrix reordering in the context of graph theory involves the systematic rearrangement of the rows and columns of an adjacency matrix with the goal of enhancing computational efficiency and improving the performance of various graph algorithms. In real-world applications, the adjacency matrices of graphs are typically sparse, containing far fewer edges than the total possible number of connections. Sparse matrices often exhibit large regions with few or no edges, and regions which are almost dense. Some reordering algorithms can exacerbate these by creating matrix zones that are particularly sparse/dense. To illustrate, consider an adjacency matrix (as in Figure 1.1) representing a social network, where each row and column correspond to individuals in the network, and each entry indicates

whether a connection exists between two individuals. In this study, we will exploit these regions in various ways.

Figure 1.1 Figure illustrating the effects of reordering on sub-regions of a graph. Both adjacency matrices belong to the same graph. Regions colored with darker shades are denser relative to the overall density of the respective matrix than regions colored with a lighter shade.



Our **hypothesis** is the following: *there is an implicit relation between graph reordering and embedding.* That is reordering graphs can be exploited for *node embedding*, where the goal is to learn low-dimensional representations of nodes in a graph

that capture their structural and relational properties. That is a vector with a given dimension is generated for each node. In large-scale graphs, the order in which nodes are presented to an embedding algorithm can significantly influence the quality of the resulting embeddings. This is because the process of node embedding relies on the sequential processing of nodes and their neighborhoods to generate vectors that encode their positions and relationships within the graph. If nodes that are structurally important or closely related in the graph are processed consecutively, the embedding algorithm can more effectively capture the underlying structure of the graph.

When embedding large-scale graphs, GPUs are typically the preferred choice due to their processing speed. However, the limited memory capacity of GPUs poses a challenge, as it is often not feasible to fit both the embedding matrix and the entire graph into memory simultaneously. To address this, embedding pipelines may use multiple GPUs to partition the matrix across them. Alternatively, *if only a single GPU is available*, which is the main case studied in this thesis, the graph and its corresponding embedding matrix must be processed in batches, with data being transferred in and out of the GPU as needed. For a given GPU-based embedding tool, we focus on improving two different quality metrics.

- *The accuracy of the embedding for a specific task*: If the adjacency matrix is ordered randomly, the embedding algorithm might process a batch containing weakly connected nodes that are not similar to each other in the overall graph topology. This leads to a learning process in which the positive information, i.e., edges in between these nodes, do not accurately suit the true network structure. However, if the matrix is reordered such that strongly connected nodes, or nodes central to the network's structure, are processed in succession, the batches used in the embedding process can more accurately capture the local and global properties of the graph. To this end, in Section 3.4, we use various reordering algorithms using different strategies with different focuses to reorder the rows and columns of a matrix on medium- to large-scale graphs and evaluate the effects of reordering on the accuracy.

- *The runtime of the embedding*: The batching approach also introduces a significant bottleneck; transferring the embedding data between the GPU and main memory becomes increasingly time-consuming as the graph size and the latent dimension of the embedding, i.e., size of each vector generated for the nodes, grow. Transferring the sparse(r), possibly empty, regions and their corresponding parts of the embedding matrix to the GPU incurs (almost) an unnecessary overhead. These regions usually contribute little to the learning

process. In many cases, they represent small, isolated parts of the graph that have minimal impact on the overall embedding quality and may even be considered as noise. After reordering graphs and creating these regions/batches, we also try to speed up the process of embedding by introducing different sampling schedules that probabilistically eliminate these regions from the embedding process and, therefore eliminate the execution time overhead created by carrying these regions into the GPU. Section 3.3 focuses on the runtime dimension of this thesis.

This work aims to **contribute** to the literature in following ways: We aim to show that there is indeed a relation between graph reordering and embedding by providing results of downstream tasks using embeddings learned from randomly ordered graphs and graphs reordered by matrix reordering algorithms. We show that link prediction models and node embedding algorithms perform better when reordered graphs are used to learn embeddings instead of randomly ordered graphs. We also show that by reordering graphs we can better exploit the sparsity of real-world graphs and accelerate the embedding learning process without losing information and lowering the quality of the final embedding vectors by sparingly including certain parts of a graph in the learning process instead of treating every part equally. To show these findings are not dependent on specific node embedding algorithms, we present the results by using embeddings obtained from the work of Zhu, Xu, Tang & Qu (2019), GraphVite, and our in-house solution GOSH by Akyildiz, Alabsi Aljundi & Kaya (2020).

The rest of this thesis is organized as follows: Chapter 2 focuses on the background of node embedding and graph reordering and examines works that try to use these two concepts together in some capacity. Chapter 3 explains graph types and reordering algorithms used in this work and details the experiments performed by defining the intention behind each decision for both embedding quality and embedding speed. Chapter 4 presents the results of the experiments and Chapter 5 concludes this thesis. The appendix contains the visuals for the reordered states of the graphs used in this thesis.

2. BACKGROUND

2.1 Notation

A graph $G = (V, E)$ is a data structure that consists of a set of nodes V, which represent the entities in a dataset. The specific meaning of these entities can vary depending on the problem context. Additionally, the graph includes a set of edges $E \subseteq (V \times V)$ that represent the relationships between these entities. The embedding of a graph G is represented by a matrix $\mathbf{M}$ with dimensions $|V| \times d$ where $|V|$ is the number of nodes in the graph and d is the dimensionality of embeddings. Each row $\mathbf{M}[i]$ corresponds to the embedding of the node $i \in V$ where each value of $\mathbf{M}[i][j]$ tries to capture a feature of the node i.

A graph reordering is a permutation $\sigma : V \to V$ that reassigns each node in the graph to a new position. The reordered graph, denoted as G_σ maintains the same structure as G but with nodes rearranged according to σ

2.2 Related Work

2.2.1 Graph Embedding

Graph embedding is the process of obtaining d dimensional vectors to represent the structure and information of nodes of a graph with numerical values while minimiz-

ing the loss of information. These vectors are then used for various downstream machine learning tasks such as but not limited to link prediction, node classification and community detection that aim to understand and model the interaction inside real-life data.

The challenges of graph embedding depend on what the input and the output of the embedding algorithm will be. Many different types of graphs represent different insights about the data. For example, a graph can be used to show how data points are connected, in which case the embedding should preserve the connection information. If the graph is a knowledge graph then in addition to the connection information, the embedding should also preserve what the connections represent for each data point. As mentioned previously, depending on the task the output of the process also changes. If the task is related to singular nodes, then obtaining a node embedding where similar nodes are represented as similar vectors can be enough. If however, the task requires information on node pairs, subgraphs or edges then a different type of output (e.g. edge embeddings or whole-graph embeddings) may be more suitable (Cai, Zheng & Chang, 2018). For this research, this section will only detail previous work done on node embeddings with structural graphs (Cai et al., 2018).

2.2.1.1 Node Embedding

Node embedding techniques aim to represent each node in a graph as a low-dimensional vector while preserving the graph's structural properties and node attributes. The primary challenge in node embedding is to encode the complex relationships and dependencies between nodes in a way that meaningful patterns and similarities are retained in the embedding space. This process involves capturing both local structures (e.g., immediate neighbours of a node) and global structures (e.g., communities or clusters within the graph).

One of the critical problems in node embedding is the preservation of various types of proximities. Proximity in this context refers to the notion of similarity or closeness between nodes, which can be defined in multiple ways:

First Order Proximity: First-order proximity is the direct connection between two nodes. The first-order proximity of two nodes u and v being high means that u and v are directly connected by an edge. LINE (Tang, Qu, Wang, Zhang, Yan & Mei, 2015) models first-order proximity by defining an objective function that directly

measures the similarity between the embeddings of connected node pairs. LINE's first-order proximity model focuses on capturing the direct connections between nodes. The objective is to maximize the likelihood of observed edges and minimize the likelihood of non-existent edges in the graph. To make the optimization computationally feasible, LINE employs negative sampling, which approximates the objective function. For each observed edge (u, v), several non-edge node pairs are generated. The embedding vector $\mathbf{M}[u]$ is updated using stochastic gradient descent to maximize the probability of observed edges while decreasing the probability of non-edges. By defining a probabilistic model for edge existence and using negative sampling to optimize the embeddings, LINE effectively ensures that nodes with direct connections in the graph are embedded close to each other.

Second Order Proximity: First-order proximity is needed to ensure that the local structures within the graph are preserved as much as possible, but it doesn't account for global structures and similarities. Many real-world graphs are considered sparse, which means local connectivity of nodes doesn't provide much insight into the graph and many nodes may be considered similar without sharing an edge. Second-order proximity considers the similarity of nodes based on their shared neighbours. It has been shown that even if two nodes u and v do not share an edge, they may have similar roles in the structure if they share many common neighbours. (Dash, 2008; Jin, Girvan & Newman, 2001; Liben-nowell & Kleinberg, 2003) The second-order proximity also has the advantage of being applicable to both directed and undirected graphs whereas by definition first-order proximity can only be applied to undirected graphs. SDNE (Wang, Cui & Zhu, 2016) uses a semi-supervised deep learning framework that consists of a deep autoencoder network. This network is designed to capture both first-order and second-order structures of the graph. The layered structure of the network also allows it to capture the nature of highly non-linear graphs more effectively. The unsupervised component of the model is designed to preserve the second-order proximity while the supervised component makes use of the small amount of information available on the pairwise similarities of pairs of nodes to preserve first-order proximity.

LINE (Tang et al., 2015) also has a model that utilizes second-order proximity. The authors treat each vertex as a specific context and assume vertices with similar distributions over contexts are similar. This means each vertex has two roles for the embedding process, one as itself and the other being a context for other vertices. With this information, the probability distribution used in LINE's first-order proximity model is adapted so that instead of working over the probability of edges and non-edges, the model instead concerns itself with the probability of a node u generating a context node v. With this tweak to the distribution, the objective becomes

maximizing the likelihood of observing the context node given a target node or minimizing the distance between the distribution of contexts given the target node and their empirical probability distribution.

DeepWalk (Perozzi, Al-Rfou & Skiena, 2014) is another method that learns node embeddings by utilizing both first-order and second-order proximities. DeepWalk learns node embeddings by performing random walks on graphs and treating the resulting sequences of nodes as sentences in a language model. This approach enables the model to capture both first-order and second-order proximities through the context provided by random walks. Due to the nature of random walks, the sequences include directly connected nodes, implicitly preserving first-order structures. DeepWalk explicitly captures second-order structures by integrating SkipGram (Mikolov, Chen, Corrado & Dean, 2013), a model that maximizes the probability of a node's neighbours within a certain window in the walk sequence, into the embedding framework. For each node in a walk sequence, the skip-gram model considers the surrounding nodes within a defined context window. This means that not only direct neighbours but also nodes that are a few steps away contribute to the embedding process.

node2vec (Grover & Leskovec, 2016) is an algorithm that extends the DeepWalk method by introducing a more flexible way to generate random walks, which can capture both first-order and second-order structures more effectively. It does this by using a biased random walk strategy that balances between breadth-first search (BFS) which helps more with preserving local neighbourhoods (first-order proximity), and depth-first search (DFS) which provides a better understanding of macro structures within the graph (second-order proximity). node2vec achieves this balance. Similar to DeepWalk (Perozzi et al., 2014), the first-order proximity is captured implicitly thanks to the nature of random walks. The balance of BFS and DFS is achieved by two parameters, return parameter p and in-out parameter$q$, introduced into the random walking process. Quoting Grover & Leskovec (2016), p "controls the likelihood of immediately revisiting a node in the walk" while q "controls the likelihood of walking to nodes". This means with a higher value of p the search behaves more like BFS and a higher value of q results in a random walk process that shows DFS-like behavior. After a node is processed during the random walk, the probability of visiting a node in the next step is determined by a bias function π which uses p and q as its inputs.

2.2.2 Graph Reordering

Graph reordering is a transformation technique applied to matrices, where the rows and/or columns of the matrix are rearranged according to a specific criterion or algorithm. The primary goal of matrix reordering is to enhance the structure of the matrix to reveal certain properties, improve computational efficiency, or facilitate more effective data analysis and visualization. Different approaches to matrix reordering can cater to specific goals, such as minimizing bandwidth, enhancing sparsity patterns, or clustering similar elements.

- Approximate Minimum Degree Ordering (AMD): An efficient way of solving linear systems $\mathbf{Ax} = \mathbf{b}$ by factorizing $\mathbf{A}$. One factorization method known as the Cholesky Factorizaiton factors $\mathbf{A}$ into $\mathbf{LL}^\top$ where $\mathbf{L}$ is a lower triangular matrix with positive diagonal entries and $\mathbf{L}^\top$ is the transpose of $\mathbf{L}$. If the fill-in, i.e., the introduction of nonzero elements in positions that were zero in the original matrix, during factorization is minimal then solvers usually perform better. The minimum degree algorithm is a heuristic used in sparse matrix computations to reduce fill-in during matrix factorization. It achieves this by iteratively selecting and eliminating the node (or variable) with the smallest degree, thereby aiming to maintain sparsity and optimize computational efficiency.

  The minimum degree algorithm is a heuristic used in sparse matrix computations to reduce fill-in during matrix factorization. It achieves this by iteratively selecting and eliminating the node with the smallest degree, thereby aiming to maintain sparsity and optimize computational efficiency. Directly computing the exact minimum degree ordering can be computationally expensive for large matrices. **Approximate Minimum Degree Ordering (AMD)** Amestoy, Davis & Duff (2004) offer a more efficient approximation that balances computational cost and reordering quality. AMD achieves this by using approximation techniques to update degrees and select pivots.

- Batched Order By Attachment (BOBA): Drescher, Awad, Porumbescu & Owens (2023) propose a graph reordering technique designed to improve the efficiency of graph algorithms by reducing the number of cache misses during computation. This method focuses on reordering the vertices of a graph to enhance data locality. BOBA is based on the idea that reordering vertices by their connectivity in batches can significantly reduce the frequency of cache misses. Indeed, grouping vertices that are frequently accessed together ensures that they are loaded into the cache simultaneously. Highly connected vertices (those with higher degrees) are processed earlier, ensuring that their adjacent vertices are more likely to be accessed soon after by maximizing the likelihood

that adjacent vertices are accessed sequentially.

- The Reverse Cuthill-McKee Algorithm (RCM): (George & Liu, 1981) is a variance of the Cuthill-McKee (Cuthill & McKee, 1969) algorithm which is a graph reordering technique used primarily to reduce the bandwidth of sparse matrices. Using a variation of the BFS algorithm, RCM gives labels to neighbors of each vertex $i \in V$ until each one is labeled. The first vertex used in the labeling process has severe implications on the final output of the RCM algorithm. Selecting a node with a high eccentricity (shortest path distance from the farthest other node in the graph) allows RCM to perform better in most cases but it also means it can get very expensive to execute for certain graphs.

- Rabbit Order: Arai, Shiokawa, Yamamuro, Onizuka & Iwamura (2016) propose a method that aims to improve the end-to-end (reordering followed by graph analysis) performance by providing a reordering that reduces fill-in, achieves a low bandwidth and short reordering time. It improves locality by producing dense non-zero blocks by detecting and using communities and it parallelizes the community detection to achieve fast reordering times.

- SlashBurn (SBURN): SBURN by Lim, Kang & Faloutsos (2014) challenges the conventional block-diagonal view of real-world graphs and introduces a new approach to graph compression that leverages the presence of hubs and super-hubs. This methodology aims to improve the understanding and processing of complex graph structures, moving beyond the limitations of traditional community detection methods. The algorithm diverges from the traditional view of graphs as collections of dense communities or cliques. Instead, it models graphs as collections of hubs (highly connected nodes) and spokes (nodes connected to hubs). SBURN aims to place hubs in a way that edges form patterns that are easier to compress.

- Gray: Gray, as described by Zhao, Xia, Li, Zhao, Zheng & Ren (2020), is an algorithm designed to leverage the insights gained from examining the sparsity patterns within sub-matrices of the original matrix. Similar to Rabbit (Arai et al., 2016), the algorithm clusters nodes with comparable edge distributions using a density-based reordering method. These clusters of similar nodes are then individually analyzed to establish their performance limits. Finally, the sub-matrices undergo a bitmap-based reordering to enhance locality, tailored to the specific requirements identified by their performance bounds.

2.2.3 Graph Reordering and Node Embedding

There are various works combining graph reordering and node embedding. These works mainly try to exploit the performance gains obtained by reordering graphs in a way that allows for more efficient memory access patterns. Coleman, Segarra, Shrivastava & Smola (2021) address the challenge of performing efficient nearest neighbor search in large graphs. The main idea is to optimize the memory layout of the graph to enhance cache efficiency, thereby speeding up the search process. The first step is to compute the embeddings for each node in the graph. Using the computed embeddings, an initial ordering of the nodes is created. Nodes with similar embeddings (hence similar structural properties) are placed close to each other in this initial order. The core of the graph reordering algorithm takes the initial node ordering and further refines it to optimize cache efficiency. Zhao, Rong, Yu, Huang & Zhang (2020) propose Deep Order Network (*DON-RL*), a learning-based approach to find optimal graph orderings to improve the performance of graph-based algorithms by reordering the graph in a way that enhances computational efficiency. This is achieved through calculating node embeddings and then learning an optimal ordering based on these embeddings. The learning objective is usually minimizing a metric that can offer insight on the quality of the reordering such as fill-in or bandwidth.

Processing graphs that are reordered usually results in increased performance for the graph analysis step, but the reordering process itself may introduce a significant amount of overhead to the end-to-end runtime depending on the application. Using this fact as motivation, Balaji & Lucia (2018) compile lightweight reordering algorithms in their work and test them together with different analysis techniques to identify the characteristics of these applications. By linking the performance of lightweight algorithms to the structural properties of the input graph, the authors also propose a way to determine whether an input graph would benefit from reordering in the context of the analysis.

2.2.4 Embedding Quality

In addition to the efficiency of applications, the embedding quality is also a very important factor when it comes to graph analysis. If embedding quality is overlooked then the overall analysis would suffer in terms of performance. In light of this, Dehghan, Kamiński, Kraiński, Pralat & Theberge (2021a) shows a systematic

analysis on embedding robustness. Performing several attacks that utilize the label and connectivity information of nodes on the input data, like edge addition and rewiring, the authors show that evaluated embedding models are sensitive to certain types of attacks, especially for downstream tasks such as node classification.

Approaching the problem of embedding quality from a different perspective, Dehghan, Kamiński, Kraiński, Pralat & Theberge (2021b) aim to provide a thorough evaluation of various node embedding algorithms on machine learning tasks such as node classification, community detection, or link prediction. They evaluate models on how sensitive they are to statistics like network size, degree distribution, noise levels, maximum degree and community sizes. The authors conclude that node2vec is the best possible choice if the embedding algorithm has to be chosen before the analysis starts but also state that the best possible approach is to generate several embeddings and compare them using a framework before deciding.

Traditional methods usually have an underlying method that require hyper-parameters which needs to be set before the process of embedding starts (e.g. DeepWalk (Perozzi et al., 2014) requires random walk lengths to be set beforehand). Abu-El-Haija, Perozzi, Al-Rfou & Alemi (2018) explore a novel approach to node embeddings that leverages an attention mechanism to improve the quality of embeddings by automatically learning optimal hyper-parameters.

2.2.5 Embedding Speed

Since it has been established that node embedding is a very time-consuming process, various works propose different approaches to node embedding intending to reduce execution time. Neighbourhood Based Node Embeddings (NBNE) (Tiago Pimentel, 2018) borrows the idea of sentence generation from (Grover & Leskovec, 2016; Perozzi et al., 2014) and modifies it to use the neighbourhood of a node to generate sentences instead of random walks. By limiting the number of neighbours that can be selected for the sentence and using permutations of neighbourhoods for each round, the authors create a controlled environment where the trade-off between training time and dataset size is dependent on an input value that dictates the number of these permutations. This makes it so that even if the time complexities are similar, NBNE usually is faster than node2vec (Grover & Leskovec, 2016) and DeepWalk (Perozzi et al., 2014).

The MERIT framework (Che, Liu, Wang & Liu, 2023) introduces a multi-level graph

embedding refinement approach designed to enhance both the quality and speed of embeddings. Unlike traditional methods such as node2vec (Grover & Leskovec, 2016) and DeepWalk (Perozzi et al., 2014), MERIT leverages a hierarchical strategy, refining embeddings iteratively across multiple levels of the graph's structure. By decomposing the graph into smaller subgraphs and performing localized refinements, MERIT achieves a balance between computational efficiency and embedding quality. This hierarchical refinement allows the framework to handle large-scale graphs more effectively, making it significantly faster while maintaining high-quality embeddings.

GraphVite (Zhu et al., 2019) is designed to address the time-consuming nature of node embedding processes by utilizing GPU acceleration to efficiently handle large-scale graphs. By offloading computationally intensive tasks to GPUs, GraphVite leverages the parallel processing capabilities of modern GPUs to significantly accelerate the embedding process. It supports multiple embedding algorithms, enabling users to choose the most suitable method for their specific requirements. The framework divides the embedding computations into smaller, parallelizable tasks that can be executed simultaneously on the GPU, drastically reducing the time needed for large-scale graph embeddings. This approach allows GraphVite to process graphs with millions or billions of nodes and edges much faster than traditional CPU-based methods. The library applies to various tasks such as link prediction, node classification, and community detection, providing a balance between execution time and embedding quality. By improving computational efficiency and scalability through GPU acceleration, GraphVite demonstrates practical enhancements in embedding speed, making it a useful tool for working with extensive graph datasets.

GOSH (Akyildiz et al., 2020) is another framework that utilizes the graph coarsening process like MERIT (Che et al., 2023). GOSH proposes a framework that allows embedding arbitrarily large graphs on a single GPU without being constrained by device memory limitations. This is achieved by learning embeddings for smaller graphs and mapping the learned embeddings to the next level of coarsened graphs until the whole embedding matrix is obtained. When the graph and its corresponding matrix do not fit into the device memory, GOSH obtains positive edge samples before copying data into GPU memory and copies the relative portions of the graph and the corresponding embedding matrix, allowing for batched processing.

3. REORDERING GRAPHS FOR NODE EMBEDDING

In this chapter, we delve into the core of our research on reordering graphs to enhance node embedding. Building upon the foundations laid in the previous chapters, we now present a comprehensive analysis of the reordering algorithms employed, the experimental framework, and the resultant effects on embedding performance both in terms of speed and quality. Our investigation aims to systematically explore how different reordering strategies can impact both the efficiency and quality of node embeddings, aiming to offer deeper insights into optimizing graph representation learning.

This chapter is organized as follows: First, in Section 3.1 detailed explanations of used reordering algorithms are given. Then, Section 3.2 details the graph types used in the experiments. Experiment details on embedding speed with reordered graphs and problem definition are given in Section 3.3. Finally, problem definition and experiment details for experiments on embedding quality are given in Section 3.4.

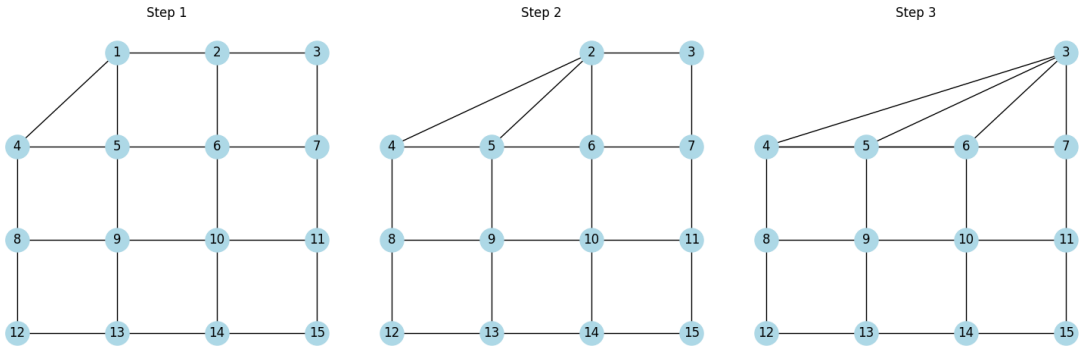
3.1 Reordering Algorithms

As has been established in Section 2.2, there are multiple ways of reordering graphs that make it easier to process them for different kinds of tasks. What follows is a detailed explanation of each of the algorithms used for this research.

- Approximate Minimum Degree Ordering (AMD): AMD is an improvement on Minimum Degree Ordering, which was explained briefly in Section 2.2. To explain AMD in detail, a brief explanation of elimination and quotient graphs is needed. During factorization of a matrix $\mathbf{M}$, represented by a graph $G = (V, E)$ a nonzero pivot is selected from V at each iteration. An elimination graph (an example can be seen in Figure 3.1) is a sub-graph $G^n = (V^n, E^n)$,

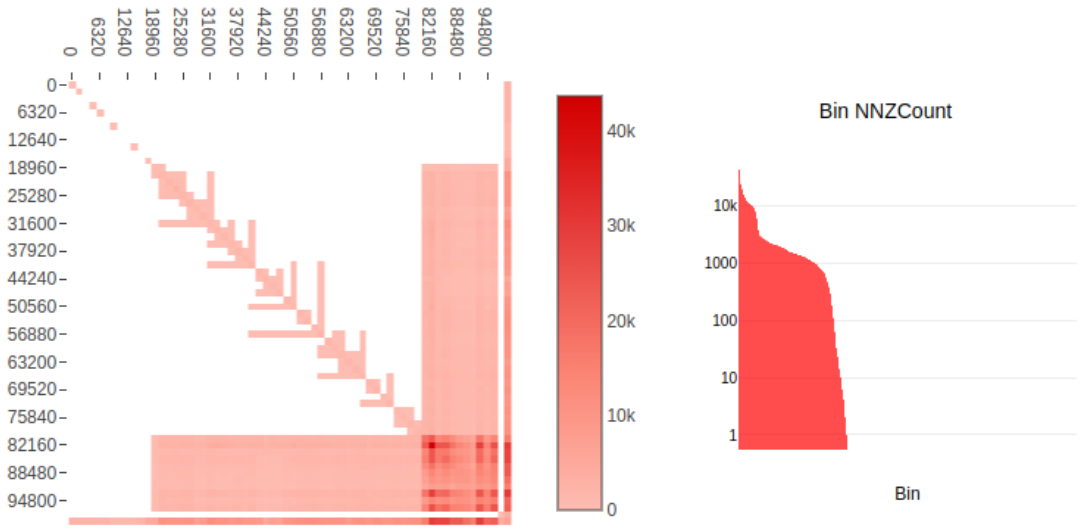
that has not been factorized yet, of G where n pivots are already selected and eliminated. A quotient graph is a structure that efficiently represents elimination graphs as a combination of sets of eliminated nodes, uneliminated nodes, and edges within corresponding sets of nodes.

Figure 3.1 Elimination process in an Elimination Graph



If two nodes in the elimination graph are neighbors and they also have identical neighbor sets they are called *indistinguishable*, as described by George & Liu (1989). When a node is eliminated from the graph, these nodes stay indistinguishable. Using this fact, AMD defines a tight upper bound on the degree of a node i which is the sum of the weights of its adjacent nodes minus a value related to overlapping sets of neighbors of i's already eliminated neighbors. Algorithm 1 provides the details for this process and Figure 3.2 shows an example of an AMD ordered graph.

Figure 3.2 AMD ordered state of a social graph



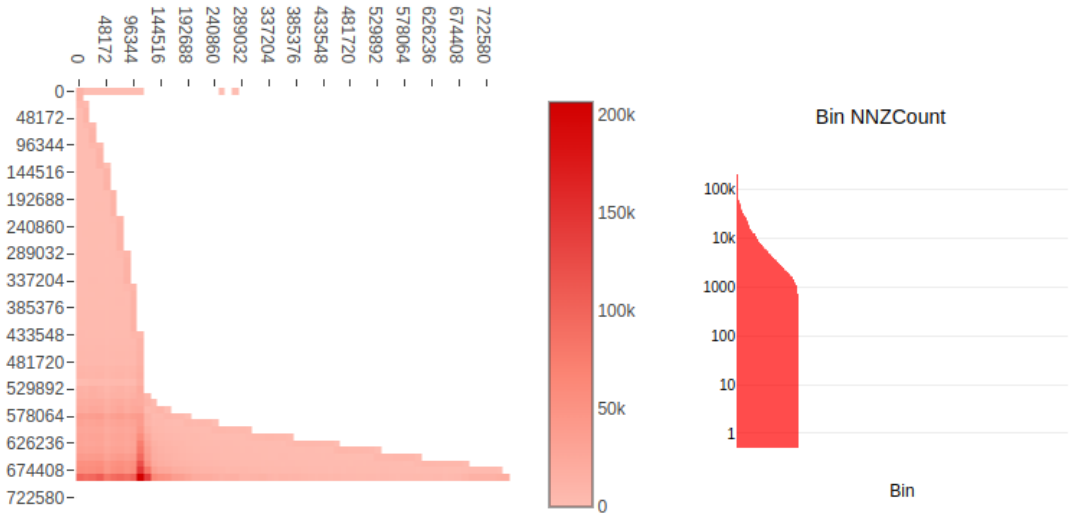
Algorithm 1 ApproximateMinimumDegree(G)

Input: $G = (V, E)$: input graph
Output: σ: Reordering sequence

```
Initialize the degree of each node
Initialize the reordering sequence σ = []
Initialize a priority queue Q with nodes sorted by degree
while not all nodes are reordered do
  Select the node v with the approximate minimum degree from Q
  Add v to the reordering sequence σ
  Remove v from the graph
  for each neighbor u of v do
    Update the degree of u in Q
  end for
end while
return Reordering sequence σ
```

- Batched Order By Attachment (BOBA): Instead of directly working on the degree values of individual nodes, BOBA takes a different approach to node embedding and is inspired by **Preferential Attachment** defined by Albert & Barabási (2002). By analyzing the expected value of a theoretical measure that scores permutations and defining an upper bound for this measure, the authors show ordering nodes by their attachment time in the preferential attachment model is a good selection for ordering nodes. In the permuted graph G_σ built by BOBA, a vertex v appears in G_σ in the order it would be seen in a preferential attachment scenario. What follows is a brief explanation of preferential attachment, algorithm details on BOBA in Algorithm 2 and an example graph in Figure 3.3.

Figure 3.3 BOBA ordered state of a social graph



Preferential attachment or the Barabási-Albert (BA) model is used to try and explain or understand the formation of scale-free networks, which are networks characterized by a few highly connected nodes (hubs) and many nodes with fewer connections. The key idea behind preferential attachment is that new nodes are more likely to connect to existing nodes that already have a high degree (i.e., many connections). In this model, a new node is connected to an existing node with a probability proportional to the degree of the existing node.

- Reverse Cuthill-McKee (RCM): RCM is a fairly straightforward reordering algorithm. It is a degree-based reordering algorithm, meaning a node is reordered concerning some rule relating to its degree. RCM aims to reduce the *bandwidth* of the matrix. A brief explanation of the concept of bandwidth is needed before the algorithm is detailed. Bandwidth can be informally described as a measure of how far the non-zero elements are from the diagonal. Mathematically, it can be described as follows:

  – The lower bandwidth of a matrix $\mathbf{M}$, denoted by l is the maximum number of sub-diagonals containing non-zero elements:

  $$l = \max_{i,j}\{i - j | a_{ij} \neq 0\}$$

  – The upper bandwidth of a matrix $\mathbf{M}$, denoted by u is the maximum

Algorithm 2 BatchedOrderByAttachment($G, batch\_size$)

Input: $G = (V, E)$: input graph
Input: $batch\_size$: size of each batch
Output: σ: Reordering sequence

```
Initialize the reordering sequence σ = []
Initialize a set U of unprocessed nodes with all nodes in G
while U is not empty do
  Initialize an empty batch B = []
  for i from 1 to batch_size do
    if U is empty then
      break
    end if
    Select a node v from U based on attachment criteria (e.g., highest degree)
    Add v to the batch B
    Remove v from U
  end for
  Add batch B to the reordering sequence σ
  for each node v in batch B do
    for each neighbor u of v in U do
      Update the attachment criteria of u
    end for
  end for
end while
return Reordering sequence σ
```

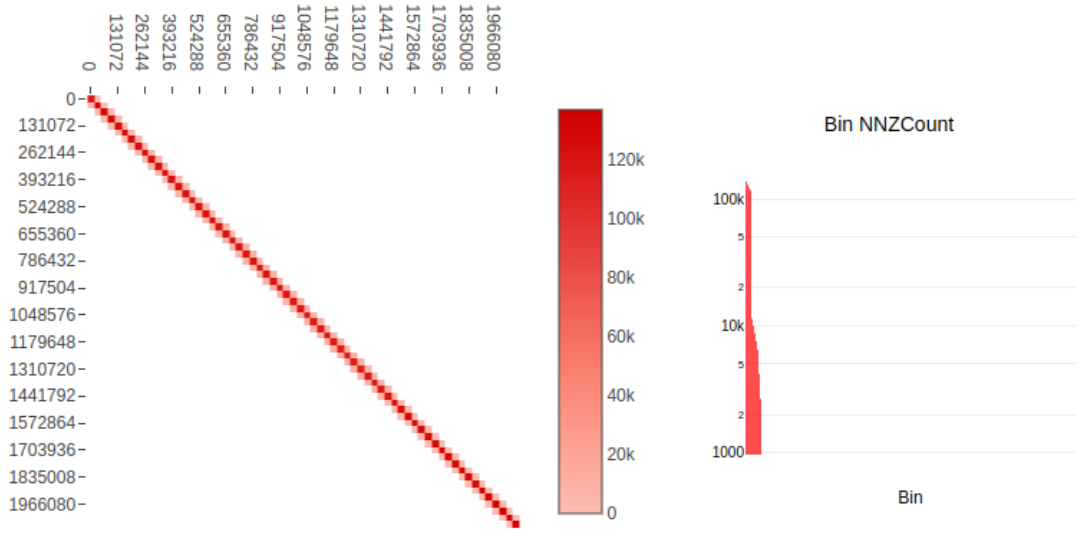
number of super-diagonals containing non-zero elements:

$$l = \max_{i,j}\{j - i | a_{ij} \neq 0\}$$

– Then, the bandwidth of $\mathbf{M}$ is calculated as:

$$Bandwidth = l + u + 1$$

Figure 3.4 RCM ordered state of a Delaunay graph



Following the explanation of the bandwidth of a matrix, Algorithm 3 shows the steps for RCM and Figure 3.4 shows an example of an RCM ordered graph.

Since every vertex is processed in some order, the selection of the initial vertex is important. Since it is out of scope for this section, the selection process will not be detailed. For completeness, various selection strategies and their details can be found in (George & Liu, 1981) and (George & Liu, 1979).

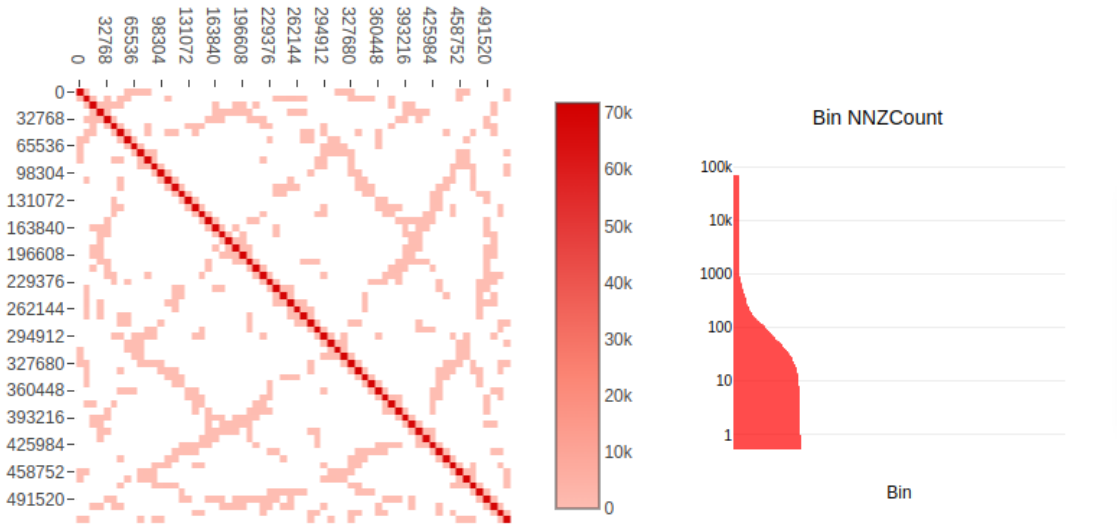
- Rabbit Order: One of the reasons why graph reordering is a widely researched area is to improve the locality of a matrix $\mathbf{M}$ representing a graph G so that during the end-to-end analysis process more efficient memory access patterns are observed. As shown by Arai et al. (2016), most graph analysis algorithms show similar patterns seen in sparse matrix-vector multiplication. When the matrix processed in sparse matrix-vector multiplication shows poor locality, the algorithm suffers greatly in terms of execution time.

Algorithm 3 ReverseCuthill-McKeeAlgorithm(G)

Input: $G = (V, E)$: input graph
Output: σ: Reordering sequence

```
Initialize the reordering sequence σ = []
Select a starting node s with the minimum degree
Initialize a queue Q and enqueue the starting node s
Mark s as visited
while Q is not empty do
   Dequeue a node v from Q
   Add v to the reordering sequence σ
   Get the list of unvisited neighbors of v sorted by increasing degree
   for each neighbor u in the sorted list do
      if u is not visited then
         Enqueue u to Q
         Mark u as visited
      end if
   end for
end while
Reverse the reordering sequence σ
return Reordering sequence σ
```

Figure 3.5 Rabbit ordered state of a random geometric graph



Arai et al. (2016) show in their that if there are dense blocks of nonzero values in $\mathbf{M}$, cache misses during the sparse matrix-vector multiplication process are minimized. They also show that constructing dense blocks of nonzero values in $\mathbf{M}$ is possible by capturing community structures in G. Furthering the concept of communities within graphs, Rabbit Order as seen in Algorithm 4 and Figure 3.5 detects hierarchical communities in G where each community

has inner communities that are denser than their parent community. By using hierarchical community ordering Rabbit Order improves locality, and by using parallel processing it decreases execution time considerably.

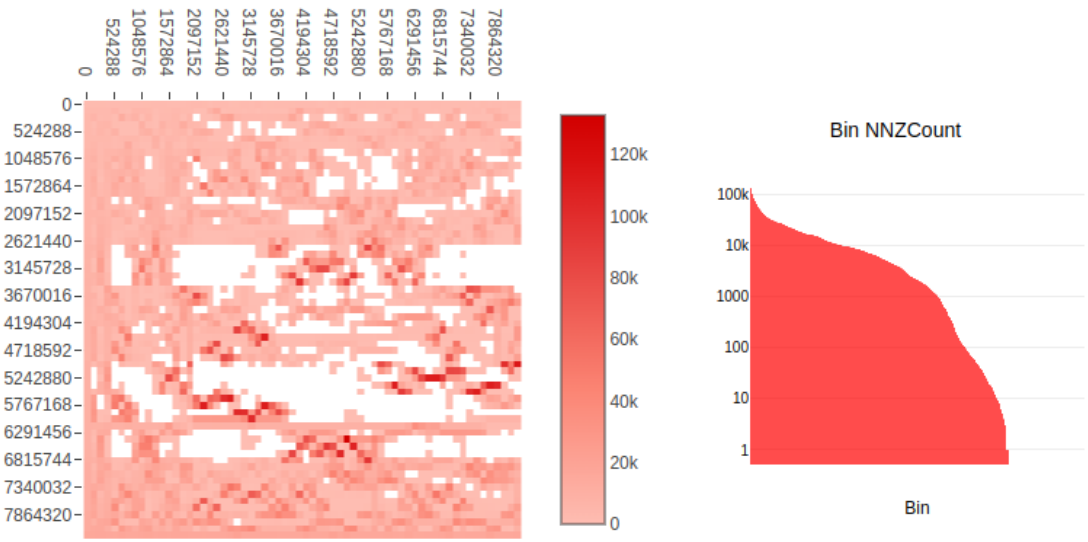
Algorithm 4 RabbitOrderingAlgorithm(G)

Input: $G = (V, E)$: input graph
Output: σ: Reordering sequence

$G = (V, E)$: input graph
Step 1 : Hierarchical Community Detection
$C = \text{CommunityDetection}(G)$
C: hierarchical communities represented as a dendrogram
Step 2 : Generate New Ordering from Dendrogram
$\sigma = \text{OrderingGeneration}(C)$
σ: new vertex ordering where vertices in the same community are co-located
return σ

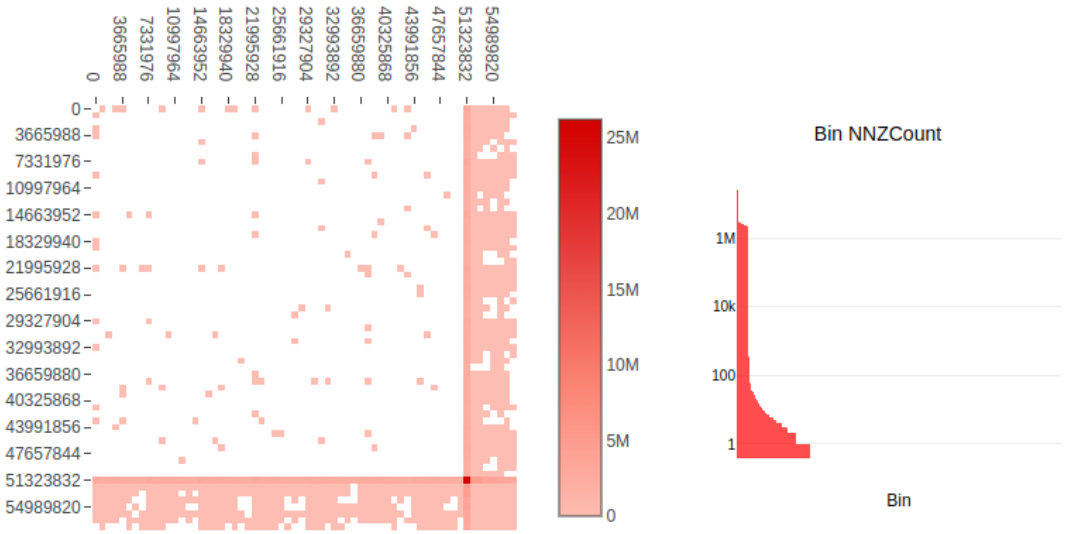
- Gray Ordering: Another ordering algorithm aiming to improve the performance of Sparse Matrix-Vector Multiplication based applications, Gray by Zhao et al. (2020) is an algorithm trying to exploit insights obtained by analysing sparsity structures of sub-matrices inside the original matrix. Like Rabbit (Arai et al., 2016), nodes that have similar edge distributions are grouped together in a density-based reordering. Then, each of these similar groups of nodes is inspected separately to determine their performance bounds. Finally, the sub-matrices are reordered with a bitmap-based reordering to improve locality based on the needs of the sub-matrix determined by its performance bound. An example of a Gray ordered graph can be seen in Figure 3.6.

Figure 3.6 Gray ordered state of a Delaunay graph



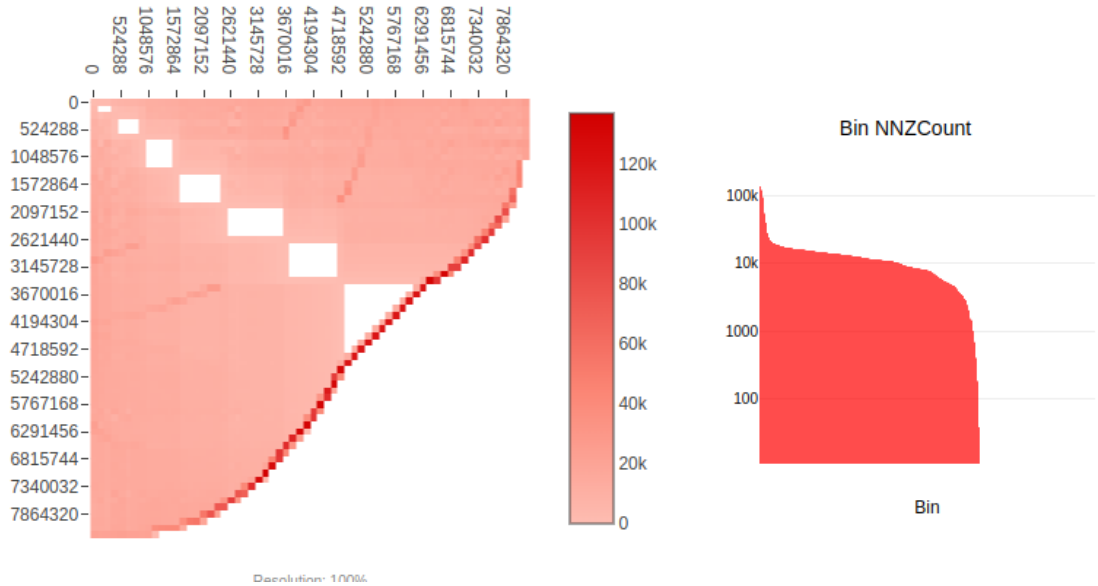
- Partitioning Tool for Hypergraphs (PaToH): PaToH by Çatalyürek & Aykanat (2011) is a hypergraph partitioning tool designed to enhance computational performance by efficiently distributing data across multiple processors. It utilizes a multilevel partitioning approach, which includes coarsening the hypergraph, partitioning the coarsened hypergraph, and refining the partition to minimize communication costs and balance the computational load. This method is particularly effective for complex hypergraphs with edges connecting multiple nodes, but it can also be applied to regular graphs by treating each edge as a hyperedge connecting two nodes. In this context, a "cut" refers to a hyperedge that connects nodes in different partitions. Cuts lead to increased communication costs between processors which means minimizing cuts is essential to reduce inter-processor communication. It is possible to use the partitioning information obtained by running PaToH together with a regular graph to reorder the rows or columns of the matrix representing the graph, as can be seen in Figure 3.7.

Figure 3.7 PaToH ordered state of a social graph



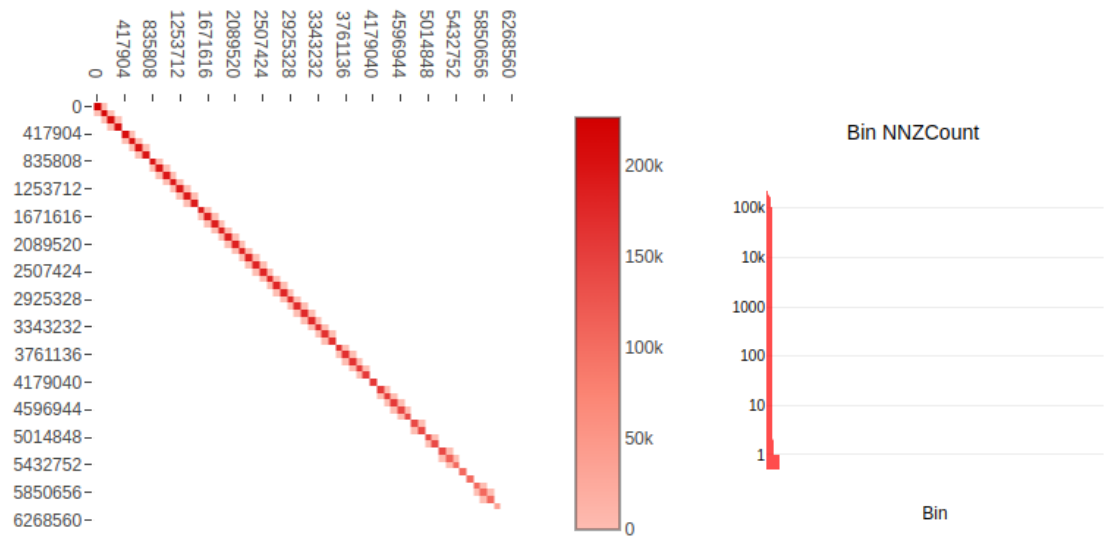
- DynaDeg: DynaDeg ordering algorithm takes inspiration from minimum degree ordering algorithms. It finds a node with a minimum degree, updates the reordering structure, removes the vertex from the original graph, and updates the degrees of its neighbors. An example of a DynaDeg ordered graph can be seen in Figure 3.8

Figure 3.8 DynaDeg ordered state of a Delaunay graph



- SlashBurn (SBURN): The SlashBurn algorithm is a graph compression technique designed to efficiently handle large-scale graphs by leveraging their underlying hub-and-spoke structure. Unlike traditional methods that focus on dense communities or cliques, SlashBurn identifies and removes highly connected nodes, or "hubs," which simplifies the graph into smaller, more manageable components (an example of this can be seen in Figure 3.9). This process, termed "slashing," is followed by a "burning" step, where the remaining nodes, now forming smaller connected components, are further analyzed and compressed.

Figure 3.9 SBURN ordered state of a road graph



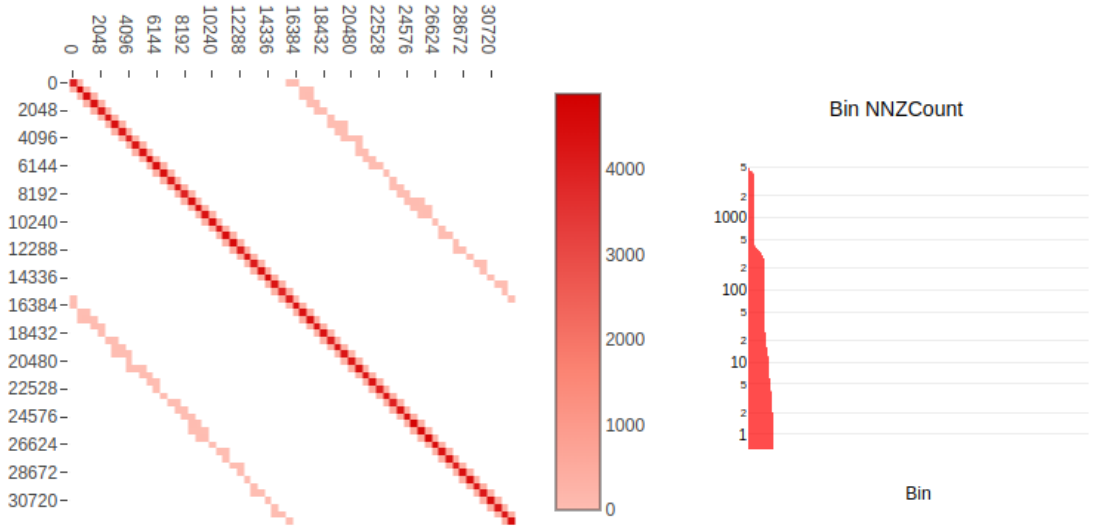
3.2 Graph Types

Understanding the diverse types of graphs is pivotal for grasping the complexities and nuances involved in reordering graphs for node embedding. Graphs serve as powerful tools to model relationships and interactions within various datasets, ranging from social networks to biological systems, and their structural differences can significantly impact the performance and outcomes of embedding algorithms. Each graph type, with its unique set of properties and characteristics, presents distinct challenges and opportunities for effective reordering and embedding. This subsection delves into several prevalent graph types, explaining in detail their traits and the implications for node embedding processes. Through this exploration, we aim to establish a solid foundation for understanding how various graph structures can influence embedding techniques and outcomes.

3.2.1 Random Geometric Graphs

First introduced by Gilbert (1961), random geometric graphs (RGGs) are a type of random graph where nodes are placed in a geometric space, typically a Euclidean space, and edges are formed based on the distance between these nodes (Figure 3.10). Nodes are distributed uniformly at random within a specified geometric space, often a unit square, unit disk, or higher-dimensional analogs. Two parameters, r and x, control the construction of an RGG.

Figure 3.10 Natural state of a random geometric graph



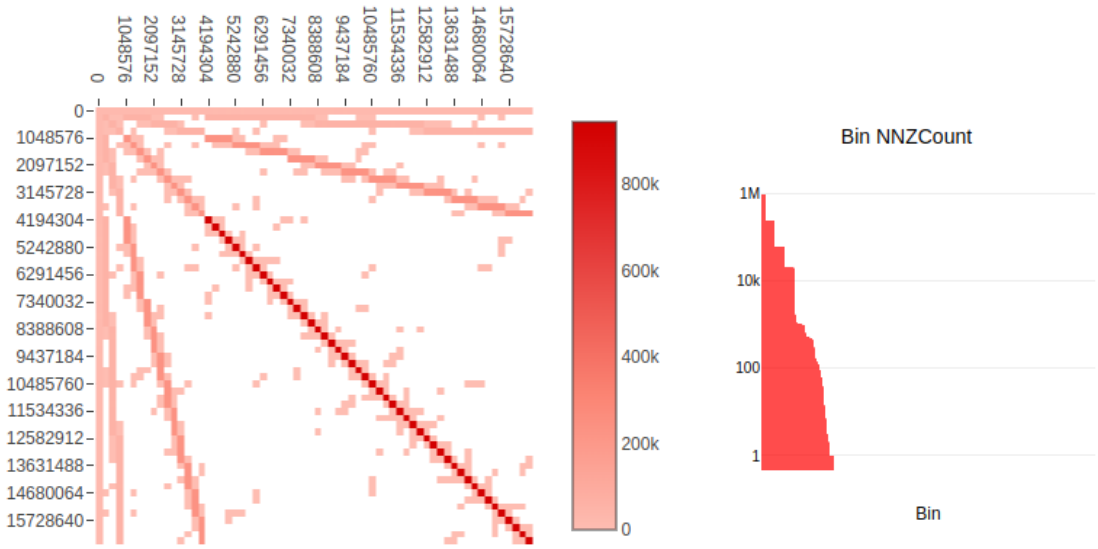
Formally, an RGG has 2^x vertices and if u and v are vertices in an RGG, an edge between them forms if the Euclidian distance between them is below r. For a unit square r is approximately proportional to $\sqrt{\frac{\log n}{n}}$ where n is the total number of vertices. Some use cases in real life include:

- Modelling communication networks in constrained areas such as wireless sensor networks or radio stations, as seen in the work by Gilbert (1961).
- Estrada, Meloni, Sheerin & Moreno (2016) use RGGs to inspect spatial properties of disease propagation on populations.
- In robotics and autonomous systems research, RGGs help in planning and understanding the movement and communication of swarms of robots or drones within a physical space. Karaman & Frazzoli (2011) makes connections between sampling-based path planning algorithms and the theory of random geometric graphs.
- Modeling various natural systems where spatial relationships are crucial, such as neural networks in the brain. Ajazi, Fioralba (2018) have done a comprehensive study on how random geometric graphs can be applied to understand neuronal network structures.

3.2.2 Delaunay Graphs

Delaunay graphs, also known as Delaunay triangulations, are another important type of geometric graph with significant applications in various fields. In a Delaunay graph, nodes are placed in a geometric space, and edges are formed such that no node lies inside the circumcircle of any triangle in the graph (Figure 3.11).

Figure 3.11 Natural state of a Delaunay graph



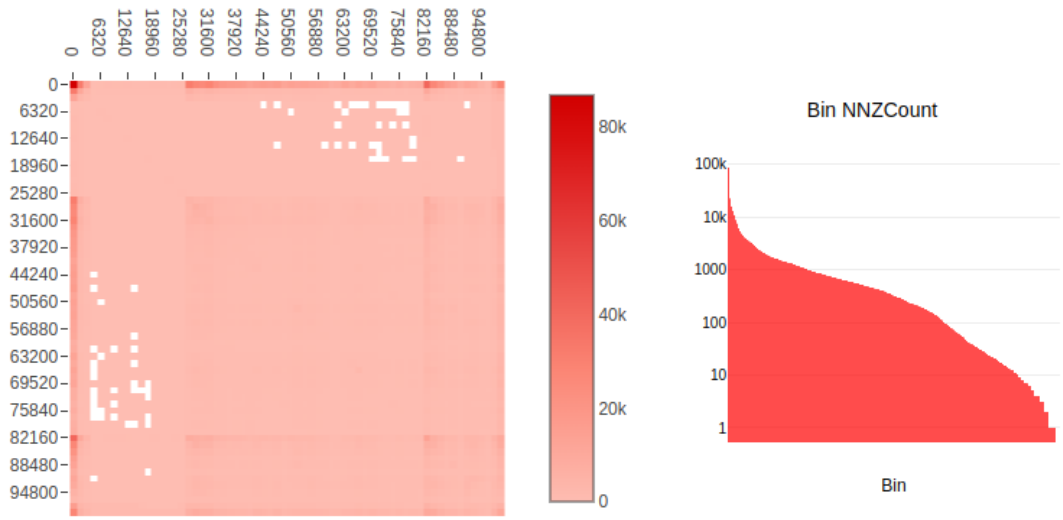
The construction of a Delaunay graph is closely related to the Voronoi diagram, with the Delaunay graph being the dual of the Voronoi diagram. Formally, given a set of points $P = \{p_1, p_2, ..., p_n\}$ in a 2D Euclidean plane, for each subset of three points in P a triangle is formed and an edge is formed between p_i and p_j if and only if there is a circumcircle of some triangle containing p_i and p_j that does not contain any other points from P within it. Some real-life use cases include:

- Martinez Santa, Martínez & Gómez (2014) use Delaunay graphs and Voronoi diagrams to craft a prediction algorithm "applicable for mobile robots with range sensors".
- In the work of Kang, Kim & Shin (2006), Delaunay triangulations are used for efficient terrain rendering for video games.
- Dakowicz & Gold (2003) use Delaunay triangulations and Voronoi diagrams to construct interpolated terrain models to obtain a realistic model of real surfaces.

3.2.3 Social Graphs

Social graphs, also known as social networks, are a type of graph used to represent relationships and interactions among individuals, organizations, or other entities (Figure 3.12). In a social graph, nodes represent the entities, and edges represent the relationships between them. These edges can be undirected, indicating mutual relationships like friendships, or directed, indicating one-way interactions such as followers on social media.

Figure 3.12 Natural ordering of a social graph



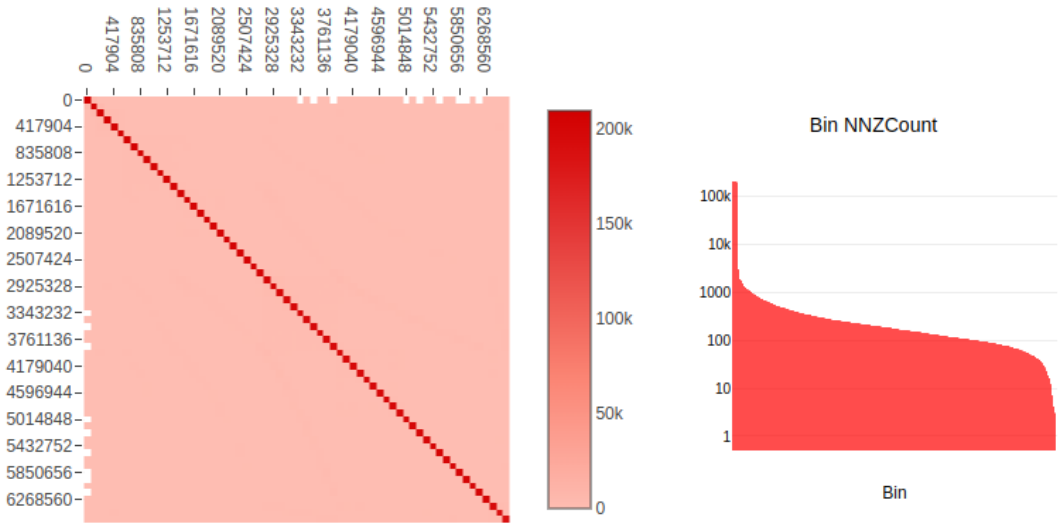
Formally, given a set of entities, an edge is formed between two entities if there exists a relationship or interaction between these two entities. The definition of a relationship or interaction varies depending on the type of entity group the graph is trying to represent. Social graphs can also include weighted edges, where the weight represents the strength or frequency of the interaction. They are used in works trying to understand the structure and dynamics of social interactions. Some real-life use cases include:

- Stattner & Vidot (2011) compile works showing the application of social network analysis and how to use it to understand the spread of infectious diseases.
- He & Chu (2010) introduce SNRS, a paradigm utilizing social network analysis to expand the usage of traditional recommender systems.
- (Ribeiro, Macambira & Neiva, 2017) is a work examining the use of social network analysis in organizational contexts. It focuses on the methodology of analysis and important choices when conducting such analyses.

3.2.4 Road Graphs

Road graphs, road networks or transportation networks are a type of graph representing the physical layout of roads and pathways within a geographic area. In a road graph, nodes represent intersections, endpoints, or key locations, while edges represent the road segments connecting these nodes (Figure 3.13). Edges, or roads, can be directed or undirected depending on whether an edge represents a one-way or a two-way road. Weights are often assigned to edges to represent distances, travel times or other relevant costs.

Figure 3.13 Natural ordering of a road graph



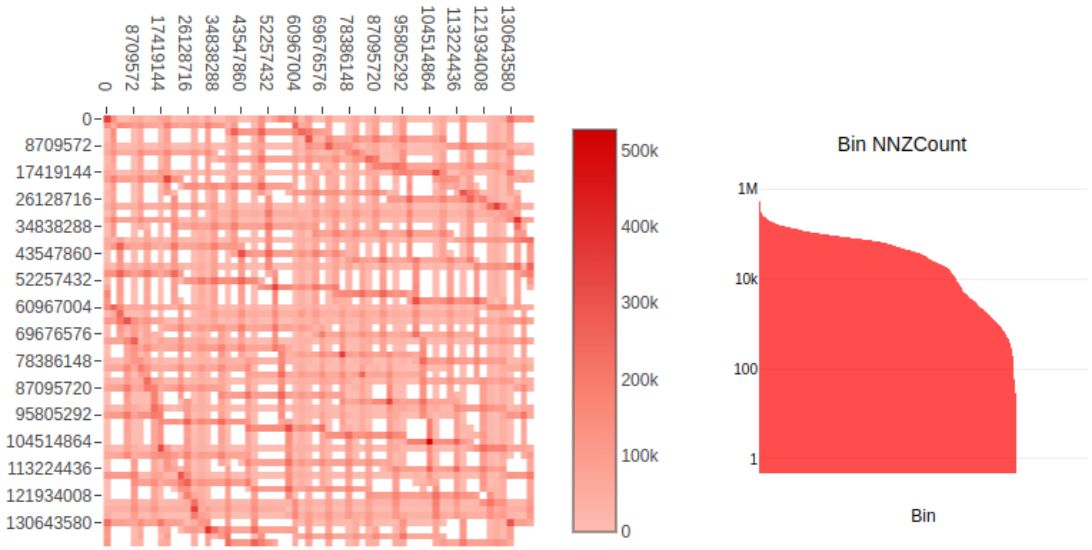
Road graphs are typically planar, reflecting the physical constraints of road layouts. Formally, given a set of intersections in a geographic space, an edge between two intersections is formed if there exists a road segment connecting these two intersections. Some real-life use cases include:

- Gharaee, Kowshik, Stromann & Felsberg (2021) presents a learning-based approach to classify road types by training a graph convolutional neural network.
- Barthélemy & Carletti (2017) use road graphs to tackle the problem of predicting traffic conditions and road demands, treating travelers as agents whose strategy is modeled by a neural network.
- Yuan, Wang & Fang (2023) propose a method based on geographic information systems (GIS) to evaluate and analyze the importance of various road segments using road graphs.

3.2.5 Protein K-mer Graphs

K-mers are substrings of length k contained within a sequence of biological sequences. Protein k-mer graphs are used in bioinformatics to analyze and represent protein sequences.

Figure 3.14 Natural state of a Protein K-mer Graph



In a protein k-mer graph, nodes represent k-mers, substrings of length k derived from a protein sequence. Edges are formed between nodes if the suffix of length $k-1$ of one k-mer matches the prefix of length $k-1$ of another kmer (Figure 3.14). Some real-life use cases include:

- REINDEER, introduced by Marchet, Iqbal, Gautheret, Salson & Chikhi (2020), is a computational method used to index k-mer sequences and their abundances using k-mer graphs.
- Rcorrector is an error correction method proposed by Song & Florea (2015) where k-mer graphs are used to correct random sequencing errors.

3.3 Reordering for Embedding Speed

One of the aims of this project is to come up with a scenario where reordering matrices results in a faster learning process for embeddings. One of the bottlenecks of the

node embedding process, especially for large-scale graphs, is the positive sampling process. Positive sampling involves selecting pairs of nodes that are close to each other in the graph (e.g., direct neighbors or nodes within a certain distance). These sampled pairs are then used to train the embedding model to learn representations that preserve the graph's local and global structure. For this part of the project, we use GOSH by Akyildiz et al. (2020) as the embedding tool.

3.3.1 Sampling in GOSH

For a normal scale graph, GOSH performs sampling and training Without going into specifics, for a graph $G = (V, E)$ GOSH needs to store the embedding matrix $\mathbf{M}$ which has $d \times |V|$ elements and the Compressed Sparse Row (CSR) representation of the graph which has $(|V| + 1) + |E|$ elements inside a single GPU. If the matrix represents a large-scale graph with millions of vertices this scenario becomes impossible because at one point a single GPU memory stops being sufficient, however large it is. GOSH overcomes this issue by choosing not to store $\mathbf{M}$ and G as a whole inside the GPU. How G is handled in this case is of particular importance to us for this part of the project. Before sampling starts, GOSH partitions the graph into k parts as seen in Figure 3.15. Then, the sampling procedure is carried out individually for each of these parts (Figure 3.16). When sampling for a part is completed, the positive samples are carried to their respective "sample pools", which are then carried over to the GPU memory. The overall part switching process is described in Figure 3.17.

Figure 3.15 GOSH Graph Partitioning Schema Before Sampling

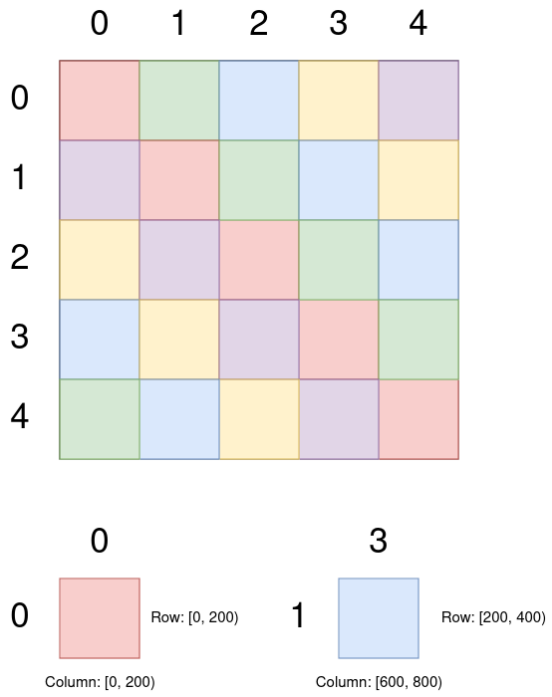


Figure 3.15 shows an example of how GOSH partitions a graph with $|V| = 1000$ and $k = 5$. Each partition has the same amount of rows and columns assigned to them, except the last partition for rows and columns since they just include whatever is left after the rest of the graph is partitioned.

Figure 3.16 GOSH sampling process. Each sampling thread fills their respective sample pool and when the pool fills the samples are carried into the GPU memory.

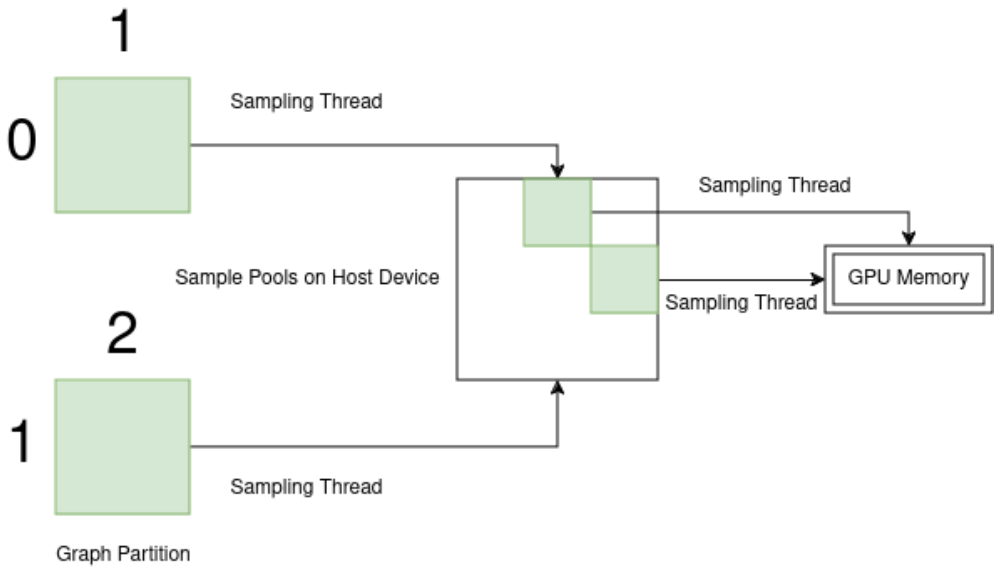
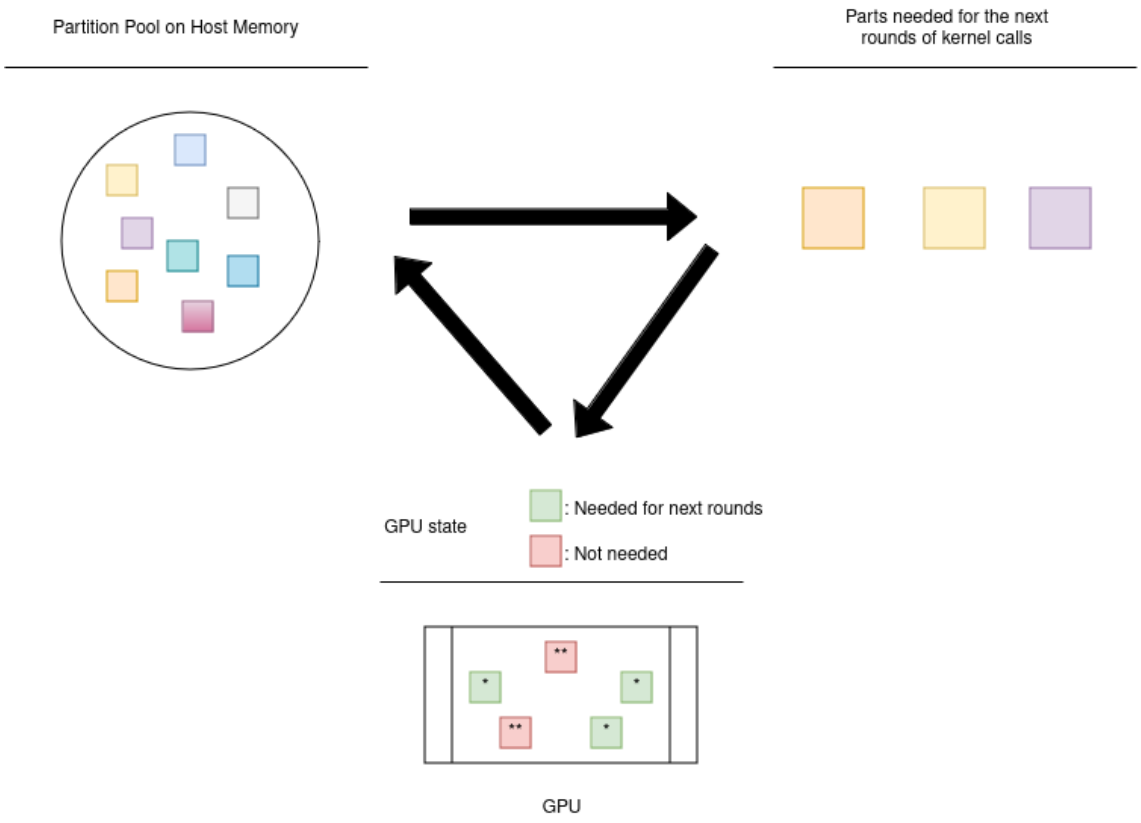


Figure 3.17 GPU-CPU Part switches happening during the embedding process. Needed parts for the next few stages are determined, and parts that are in GPU but not needed for the next stages are flagged for removal. For each part to be removed a new part arrives in the GPU and older parts are flagged for removal if they are not needed in the future.



The part switches depicted in Figure 3.17 are the performance bottleneck for GOSH's learning process. In the figure, there are two parts flagged for removal and three parts that need to be carried over to the GPU memory. This means there will be two part switches for the next step and at least one more for the next since the one part left out won't be carried into the GPU in the current step. Also, if the part coming into the GPU has a low amount of edges there is no guarantee the part will stay in there for a long time and this leads to more part switches.

GOSH uses multiple sampling strategies whose particulars are not relevant to the problem. What is relevant is that each strategy performs some task on the nonzero values existing in the part assigned to the thread executing the strategy. Depending on the structure and layout of the graph, this may present a problem. As shown by Wang, Yao, Tong, Xu & Lu (2019), all edges are not of the same importance when it comes to learning representational embeddings and changing the structure of the network may even be beneficial, depending on certain factors. Using this as motivation, we implement a scheduling strategy that aims to ignore edges that may

not be as impactful for the embedding stage while speeding up the overall process by eliminating some of the part switches.

3.3.2 Problem Definition

Figure 3.18 Nonzero ratios per part for a randomly ordered graph

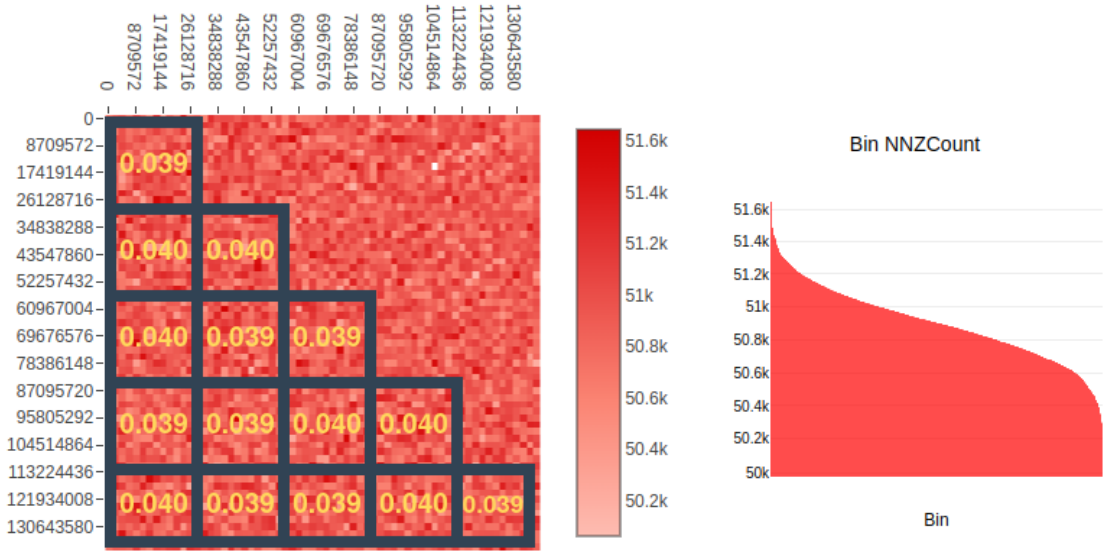
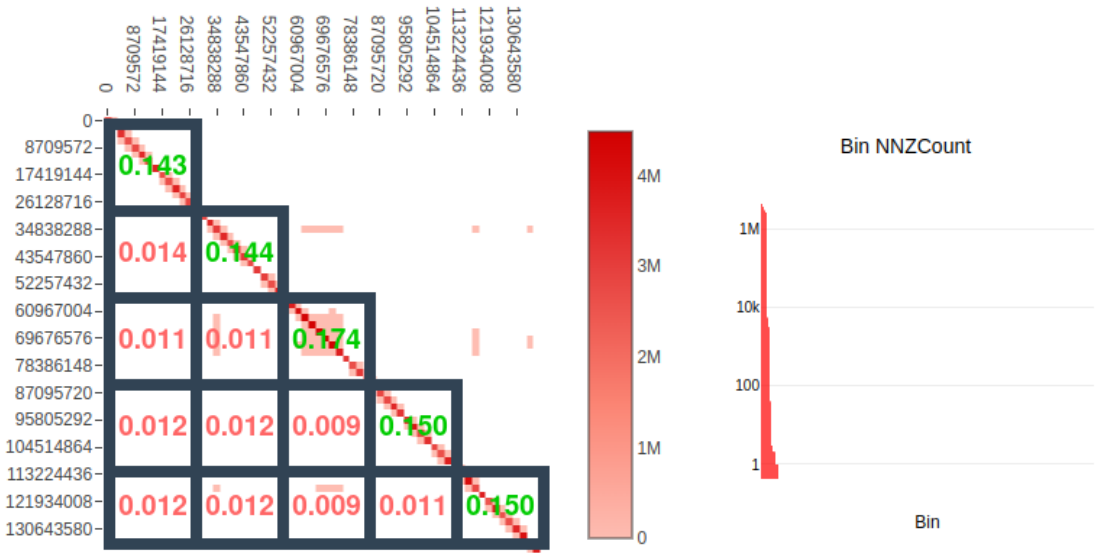


Figure 3.18 shows the distribution of nonzero values per part for a randomly ordered graph after GOSH calculates the part limits. As seen in the figure, a randomly ordered matrix representing a graph has its nonzero values (edges of the graph) about evenly distributed between each part. This means the number of positive samples coming from a part should be about as equal to the amount coming from other parts with no way of making use of a meaningful distinction between edges.

Figure 3.19 Nonzero ratios per part for a Rabbit ordered graph



The matrix shown in Figure 3.19 is the Rabbit ordered version of the same graph seen in Figure 3.18. The difference between the two figures is clear, as the Rabbit ordered matrix has partitions that are denser in terms of nonzero ratios. The problem then becomes what happens when we schedule the sampling such that the parts with nonzero ratios lower than a threshold value t are skipped in sampling rounds with probability p. In theory, this would speed up the embedding process since not only the sampling process itself is performed less but also the results that would have been obtained from these samplings won't be carried into the GPU memory. To test this theory, we introduce simple changes to GOSH's sampling algorithm to accommodate the skipping strategy.

3.4 Reordering for Downstream Task Accuracy

Node embeddings are almost always learned to use them for some downstream task (e.g. link prediction, node feature prediction). Therefore it is logical to expect that any change to any embedding strategy, whether it is changing the algorithmic details of the learning process or changing the topology and structure of the input data, should result in better or at the very least similar to the state of the art in terms of task accuracy. Reordering graphs is no different, and to show the effects of reordering graphs on embedding quality for link prediction we test two different

node embedding algorithms, LINE and DeepWalk, on reordered graphs. We use the implementations of LINE and DeepWalk provided by the work of Zhu et al. (2019), GraphVite.

3.4.1 LINE Algorithm

LINE (Tang et al., 2015) is a network embedding algorithm that aims to preserve first-order proximity and second-order proximity of nodes while being able to scale for large graphs with millions of nodes and being able to process arbitrary types of edges (e.g. directed, undirected, weighted). LINE first learns the representations for first-order proximity and second-order proximity separately. After the initial learning process is done, LINE concatenates the representations to preserve first-order and second-order proximity together in a single representation.

Algorithm 5 LINE-FirstOrderProximityLearning($epochs, nodes, edges$)

Input: *epochs*: training epoch count
Input: *nodes*: node set of the graph $G = (V, E)$
Input: *edges*: edge set of the graph $G = (V, E)$

```
for each epoch in epochs do
    Randomly initialize weights
    for each edge between nodes in edges do
        Compute the similarity between the two connected nodes
        Calculate the error between predicted and actual similarity
        Backpropagate the error to update weights
    end for
end for
```

To learn first-order proximity representations, LINE first defines a joint probability p_1 for an undirected edge (i, j) between any vertex pair i and j. Then, minimizing the KL divergence of the distribution of p_1 and the distribution of its empirical probability $\hat{p}_1$ becomes the learning objective for this part of LINE (Algorithm 5).

For second order proximity, LINE takes a similar approach in terms of logic. This time a vertex has two different uses, one as itself and one as the context for other vertices. This leads to the definition of another probability for the context representations. With this, the objective becomes to minimize the distance between the conditional distribution of the context representations given a vertex i and their empirical probability distributions. A value is introduced that represents the "prestige" of a vertex, and KL divergence is chosen as the distance function again.

Algorithm 6 LINE-SecondOrderProximityLearning($epochs, nodes, neighbors$)

Input: *epochs*: number of training epochs
Input: *nodes*: node set of the graph $G = (V, E)$
Input: *neighbors*: neighbor set of each node in *nodes*

```
for each epoch in epochs do
  Randomly initialize weights
  for each node in nodes do
    for each neighbor of the node in neighbors do
      Compute the similarity between the node and its neighbor's neighbourhoods
      Calculate the error between the predicted and actual similarity of neighbourhoods
      Backpropagate the error to update weights
    end for
  end for
end for
```

Algorithm 6 provides a simplified version of this process.

3.4.2 DeepWalk Algorithm

DeepWalk (Perozzi et al., 2014) is a framework that treats random walk sequences that are obtained from sampling as sentences. DeepWalk starts by performing random walks on the graph. For each node in the graph, DeepWalk generates multiple random walks of fixed length. These walks serve as sentences in the analogy to natural language processing (NLP). Similar to how words have context in sentences, nodes in random walks have context within a window. A sliding window of a fixed size moves over the nodes in the random walk, treating each node as a word and its neighboring nodes within the window as its context.

DeepWalk uses the skip-gram model, a neural network architecture from NLP, to learn node embeddings. The skip-gram model aims to predict the context nodes given a center node. The input to the skip-gram model is the center node, and the output is the context node. The objective is to maximize the probability of predicting the correct context nodes for each center node (Algorithm 7).

3.5 Problem Definition

Algorithm 7 DeepWalkNodeEmbeddingAlgorithm($G, wpn, wl, ws, ed, epochs$)

Input: $G = (V, E)$: input graph
Input: wpn: number of random walks per node (*walks per node*)
Input: wl: length of each random walk (*walk length*)
Input: ws: context window size for skip-gram model (*window size*)
Input: ed: dimension of node embedding vectors (*embedding dimension*)
Input: *epochs*: number of training epochs

Step 1 : Generate random walks
for each node v in G **do**
  for each i from 1 to *walks_per_node* **do**
    Perform a random walk of length *walk_length* starting from v
    Add the random walk to the corpus
  end for
end for
Step 2 : Train skip-gram model
Initialize weights randomly for the skip-gram model
for each epoch from 1 to *epochs* **do**
  for each random walk in the corpus **do**
    for each node in the random walk **do**
      Use the node as the center word
      Extract the context nodes within the window size
      Maximize the probability of the context nodes given the center node using the skip-gram model
      Calculate the error between predicted and actual context nodes
      Backpropagate the error to update weights
    end for
  end for
end for

The challenge arises when considering the impact of reordering graphs on the performance of embedding algorithms like LINE and DeepWalk. Specifically, we aim to understand how reordering affects the accuracy and convergence of these algorithms. By reordering the graph's nodes and edges, we hypothesize that the structural properties captured by the embeddings may vary, influencing their effectiveness in downstream tasks. To evaluate this, we propose reordering the graph and then applying LINE and DeepWalk to generate embeddings. Subsequently, we will test these embeddings using a link prediction model to measure their accuracy and convergence.

4. RESULTS AND DISCUSSION

In this chapter, we present the results of our tests on reordering graphs to enhance node embedding, as detailed in the previous chapter. First, the dataset used is shared and details on the graphs are given. Then, a detailed explanation of the link prediction performed on embeddings obtained by GraphVite's implementation of LINE and DeepWalk is given, followed by the results. Finally, the details of testing on reordering graphs to improve embedding speed are given together with results measuring both embedding speed and accuracy on link prediction.

All matrix visualisations seen in this work have been produced by SparseViz [1]. SparseViz is a matrix reordering and visualisation library written in C/C++. It includes implementations for all the reordering algorithms mentioned and also produces visualisations for each of the desired algorithms given an input graph. The visualisations also include critical stats that are helpful for matrix-based applications such as bandwidth, average and median nonzero values, empty bin counts, and average, maximum, and minimum row and column spans.

4.1 Evaluation Metrics

To evaluate the various methods and approaches we apply to see the effects of reordering on both embedding speed and embedding quality, we decided on several metrics. Since we want to show if there are any improvements in terms of execution speed, we present GOSH's execution time in two segments:

- Time it took to carry samples into GPU memory
- Time it took to remove samples from GPU memory and carry over new samples

[1] https://github.com/sparcityeu/SparseViz

To measure the quality of the embeddings after being subjected to our scheme, we present accuracy values on the validation set obtained from a logistic regression model trained for link prediction. The preprocessing and evaluation pipeline is mostly adopted from the work of Akyildiz et al. (2020). First the input graph G is separated into training and test graphs with an 80-20 split. If there are any isolated edges in the training set we remove them, and if there are any edges (u, v) in the test graph where one of u or v is not in the training graph the edge is removed from the test graph. Features for the training samples are obtained by element-wise multiplication of the embedding vectors of the nodes belonging to a given sample (negative or positive).

To show how reordering affects embedding quality, we present AUC scores and TopHits@K calculations for K values of 20, 50, and 100 obtained from the link prediction network included in GraphVite (Zhu et al., 2019).

4.2 Dataset

Table 4.1 Graphs used for tests

Graph Name	Node Count	Edge Count
soc-douban	154908	327162
soc-slashdot	70068	358647
soc-gowalla	196591	950327
soc-youtube	495957	1936748
soc-buzznet	101163	2763066
soc-lastfm	1191805	4519330
soc-digg	770799	5907132
soc-livejournal	4033137	27933062
soc-sinaweibo	58655849	261321071
delaunay_n21	2097152	6291408
delaunay_n22	4194304	12582869
delaunay_n24	16777216	100663202
rgg_n_2_19_s0	524288	3269766
rgg_n_2_20_s0	1048576	6891620
rgg_n_2_24_s0	16777216	265114400
road-belgium-osm	1441295	1549970
road-italy-osm	6686493	7013978
kmer_P1a	139353211	148914992

We chose to use a varied selection of graphs obtained from SuiteSparse (Davis & Hu, 2011), and The Network Repository (Rossi & Ahmed, 2015,1). For the tests involving the convergence of models, we use social graphs `soc-livejournal` and `soc-buzznet`. To show the effects of our partition skipping strategy for GOSH's sampling we use `kmer_P1a`, `soc-sinaweibo`, `rgg_n_2_24_s0` and `delaunay_n24` which are all large-scale graphs. 7 social graphs, 2 Delaunay graphs, 2 random geometric graphs and 2 road graphs are used to obtain and analyse link prediction AUC scores. In total, we use 8 social graphs, 3 Delaunay graphs, 3 random geometric graphs, 2 road graphs, and 1 k-mer protein graph. Readers can refer to Table 4.1 for details.

To prepare training and validation data for link prediction models, we removed edges from the original graph randomly, reordered the training set, and used the permutation obtained on the removed edges as well. The validation set did not have any effect on the final matrices produced by reordering.

4.3 Embedding Quality

4.3.1 Convergence - Social Graphs

Figure 4.1 Convergence on `soc-buzznet`

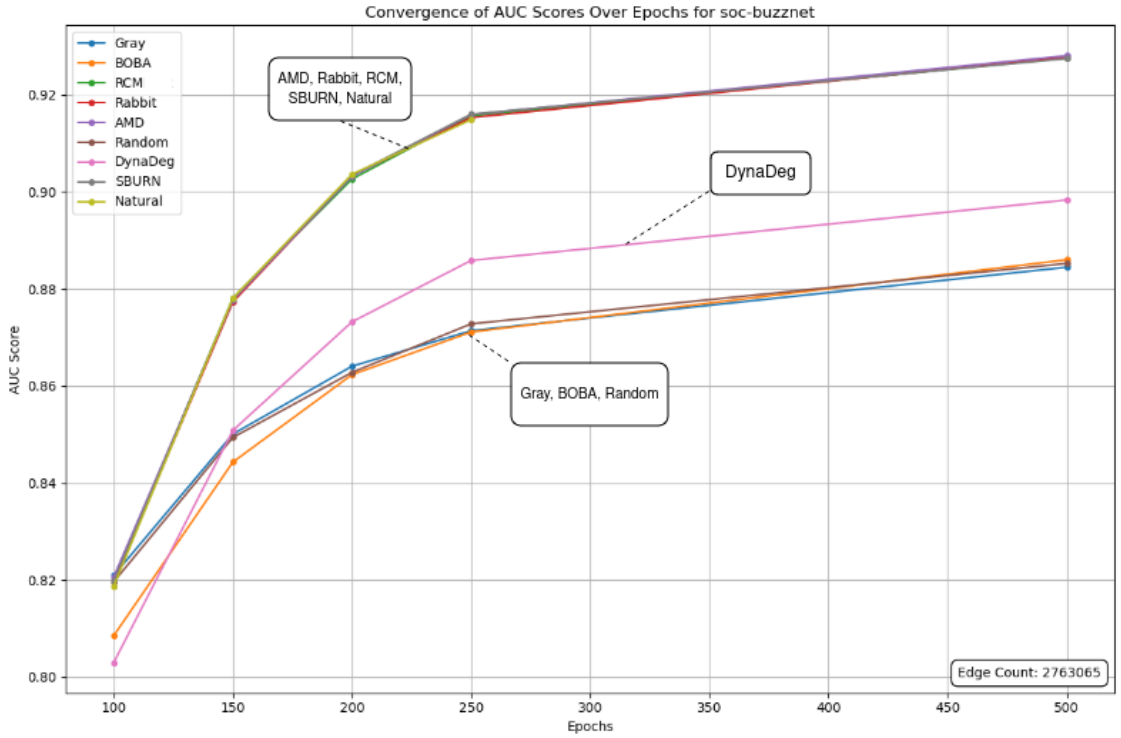


Figure 4.1 shows the AUC scores of training embeddings after 100, 150, 200, 250 and 500 epochs. As the figure shows, there are clear distinctions between the performance of different reordering algorithms. Among the best-performing algorithms are AMD, Rabbit, RCM and SBURN. DynaDeg, BOBA and Gray still outperform the random ordering but they can not separate themselves as fast or as drastically from the better-performing reorderings.

Figure 4.2 Convergence on `soc-livejournal`

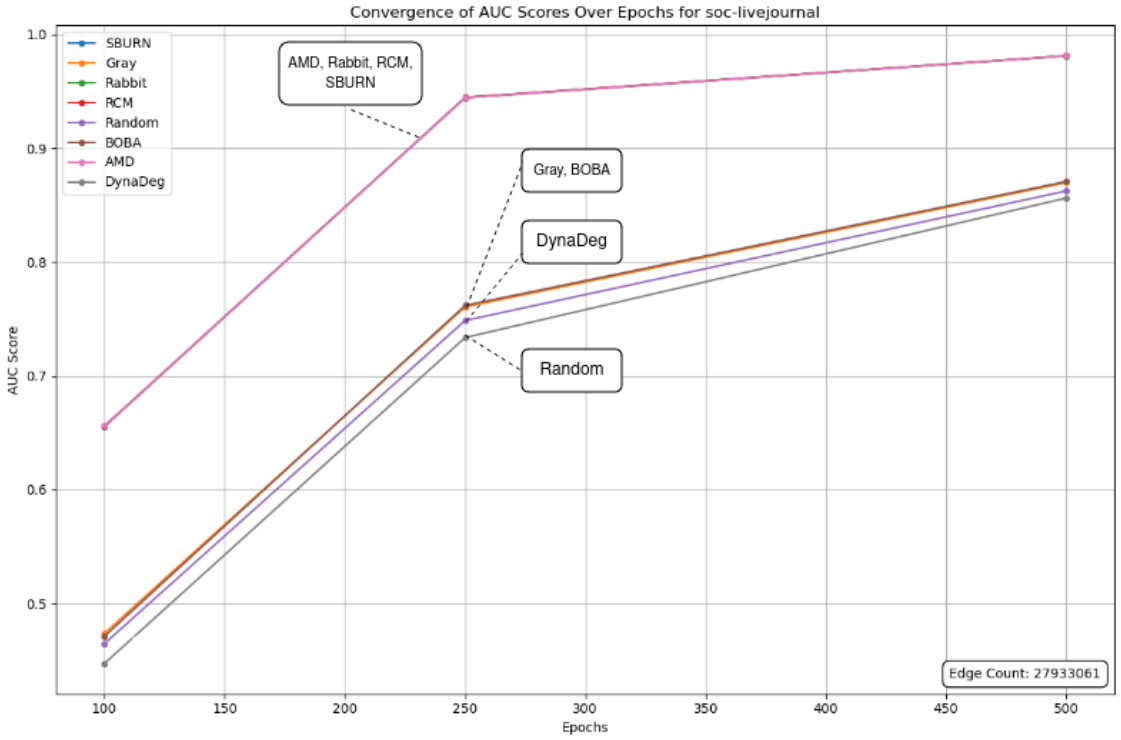


Figure 4.2 shows the AUC scores after 100, 250 and 500 epochs of embedding training. This time the scores start at lower values compared to `soc-buzznet`. This is expected since `soc-livejournal` has a lot more relationships, therefore representing it with embeddings is harder for models. When it comes to convergence, the better-performing algorithms provide the fastest convergence to higher scores again. This time, the separation between the best-performing algorithms and the rest is even more clear. DynaDeg performs poorly, however, as it has been beaten by random ordering.

4.3.2 AUC and TopHits@K - Social Graphs

For social graphs, Table 4.2 shows models are usually successful to some degree in learning the relationships within the graph. This is due to the fact that social graphs usually have well-defined community structures and homogenous connectivity patterns. Learning embeddings on random orderings usually result in the worst AUC scores but even then the values are around 0.8. The better-performing algorithms like AMD, RCM, Rabbit and SBURN usually exceed 0.9 and the other algorithms

Table 4.2 AUC scores for social graph reorderings

Graph Name	AMD	BOBA	DynaDeg	Gray	RCM	Rabbit	Random	SBURN
soc-douban	**0.882**	0.852	0.827	0.831	**0.882**	**0.882**	0.821	0.881
soc-slashdot	**0.925**	0.883	0.891	0.883	**0.925**	0.924	0.884	**0.925**
soc-gowalla	**0.977**	0.914	0.911	0.911	**0.977**	**0.977**	0.904	**0.977**
soc-youtube	**0.960**	0.905	0.909	0.903	**0.960**	**0.960**	0.900	**0.960**
soc-buzznet	**0.925**	0.897	0.903	0.895	**0.925**	0.924	0.896	0.924
soc-lastfm	**0.975**	0.970	0.971	0.968	**0.975**	**0.975**	0.969	**0.975**
soc-digg	**0.975**	0.968	0.969	0.967	**0.975**	**0.975**	0.968	**0.975**

never perform worse than the random ordering.

Table 4.3 TopHits@20 for social graph reorderings

Graph Name	AMD	BOBA	DynaDeg	Gray	RCM	Rabbit	Random	SBURN
soc-douban	0.75	0.80	0.75	**0.85**	0.80	**0.85**	0.70	0.75
soc-slashdot	0.85	0.70	0.80	0.80	**1.00**	0.85	0.90	0.95
soc-gowalla	**0.95**	0.90	0.80	0.85	**0.95**	0.90	0.80	**0.95**
soc-youtube	**1.00**	0.85	0.95	0.85	0.90	0.90	0.85	0.90
soc-buzznet	**0.95**	0.85	0.75	0.70	**0.95**	0.90	0.75	0.85
soc-lastfm	0.95	0.90	0.95	0.95	0.90	**1.00**	0.90	0.90
soc-digg	**0.95**	0.90	**0.95**	0.90	**0.95**	0.90	0.90	0.90

Table 4.4 TopHits@50 for social graph reorderings

Graph Name	AMD	BOBA	DynaDeg	Gray	RCM	Rabbit	Random	SBURN
soc-douban	**0.84**	0.74	0.70	**0.84**	0.78	0.80	0.70	**0.84**
soc-slashdot	0.80	0.76	0.76	0.82	**0.94**	0.82	0.86	**0.94**
soc-gowalla	0.94	0.88	0.84	0.88	0.92	0.92	0.82	**0.96**
soc-youtube	**0.94**	0.88	0.76	0.84	0.82	0.90	0.84	0.90
soc-buzznet	0.84	0.80	0.76	0.78	**0.90**	0.86	0.78	0.88
soc-lastfm	**0.98**	0.94	0.96	0.96	0.94	0.94	0.92	0.92
soc-digg	0.94	0.92	0.92	0.90	**0.98**	0.92	0.94	0.90

Table 4.5 TopHits@100 for social graph reorderings

Graph Name	AMD	BOBA	DynaDeg	Gray	RCM	Rabbit	Random	SBURN
soc-douban	0.79	0.78	0.68	**0.81**	0.79	**0.81**	0.70	0.80
soc-slashdot	0.85	0.76	0.78	0.82	0.90	0.88	0.82	**0.95**
soc-gowalla	0.91	0.83	0.83	0.89	**0.95**	**0.95**	0.81	0.94
soc-youtube	**0.93**	0.81	0.76	0.84	0.82	0.87	0.86	0.89
soc-buzznet	0.84	0.78	0.81	0.78	**0.86**	0.85	0.75	0.85
soc-lastfm	**0.98**	0.93	0.97	0.96	0.95	0.90	0.93	0.94
soc-digg	0.94	0.92	0.91	0.89	**0.96**	0.92	0.94	0.93

When it comes to predicting positive edges, the TopHits@K metrics show the better-performing algorithms still produce the best results for the large part, but the difference in performance is not as drastic as the previous category of AUC scores. The results are shown in Tables 4.3, 4.4 and 4.5 for k values of 20, 50 and 100 respectively.

4.3.3 AUC and TopHits@K - Road Graphs

Table 4.6 AUC scores for road graph reorderings

Graph Name	AMD	BOBA	DynaDeg	Gray	RCM	Rabbit	Random	SBURN
road-belgium-osm	**0.632**	0.585	0.590	0.584	0.622	**0.632**	0.584	**0.632**
road-italy-osm	**0.624**	0.585	0.590	0.585	**0.624**	**0.624**	0.584	**0.624**

In Table 4.6 the results show that the models have a harder time learning the embeddings of a road graph than a social graph. This is because road graphs show complex connectivity patterns. The lengths of roads and complex intersections make it harder for models to understand the underlying structure. Road graphs also have spatial characteristics. A road may be a highway, a local street or a roundabout and being connected to different types of nodes would have different implications for intersections of these roads, making the learning process even harder.

Reordering still improves link prediction compared to a randomly ordered road graph. The better-performing class of algorithms again shows the best results with no algorithm performing worse than random ordering.

Table 4.7 TopHits@20 for road graph reorderings

Graph Name	AMD	BOBA	DynaDeg	Gray	RCM	Rabbit	Random	SBURN
road-belgium-osm	0.50	0.45	0.35	0.25	0.50	**0.60**	0.50	0.55
road-italy-osm	**0.55**	0.35	0.45	0.30	0.25	0.50	0.45	**0.55**

Table 4.8 TopHits@50 for road graph reorderings

Graph Name	AMD	BOBA	DynaDeg	Gray	RCM	Rabbit	Random	SBURN
road-belgium-osm	0.40	0.46	0.42	0.30	0.52	**0.68**	0.50	0.46
road-italy-osm	0.44	0.32	0.50	0.38	0.52	**0.56**	0.40	0.54

Table 4.9 TopHits@100 for road graph reorderings

Graph Name	AMD	BOBA	DynaDeg	Gray	RCM	Rabbit	Random	SBURN
road-belgium-osm	0.42	0.49	0.49	0.49	0.51	0.51	**0.54**	0.47
road-italy-osm	0.47	0.41	0.44	0.43	**0.53**	0.52	0.39	**0.53**

TopHits@K measurements for road graphs, seen in Tables 4.7, 4.8 and 4.9 show that models are helped by reordering in predicting positive edges since the best results come from reordered graphs again with the exception of TopHits@100 for road-belgium-osm.

4.3.4 AUC and TopHits@K - Delaunay Graphs

Table 4.10 AUC scores for Delaunay graph reorderings

Graph Name	AMD	BOBA	DynaDeg	Gray	RCM	Rabbit	Random	SBURN
delaunay_n21	**0.999**	0.996	0.761	0.893	**0.999**	**0.999**	0.756	**0.999**
delaunay_n22	**0.999**	0.997	0.764	0.885	**0.999**	**0.999**	0.756	**0.999**

Like road graphs, Delaunay graphs also have spatial properties. The difference is that while road graphs show complex connectivity structures, Delaunay graphs have well-defined and straightforward properties on what an edge between two entities means. Even without reordering the connected nodes are very close to each other in space. With reordering, this locality is emphasized even more which helps learners capture local relationships and neighborhoods. Most of the reorderings show excellent results in Table 4.10 and algorithms that may show poor performance for other types of graphs perform significantly better than the randomly ordered graph.

Table 4.11 TopHits@20 for Delaunay graph reorderings

Graph Name	AMD	BOBA	DynaDeg	Gray	RCM	Rabbit	Random	SBURN
delaunay_n21	**1.00**	**1.00**	0.60	0.65	**1.00**	**1.00**	0.65	**1.00**
delaunay_n22	**1.00**	0.95	0.80	0.90	**1.00**	**1.00**	0.35	**1.00**

TopHits@K measurements seen in Tables 4.11i 4.12 and 4.13 reinforce the findings of Table 4.10, as the models produce good results for almost all of the reordered graphs.

Table 4.12 TopHits@50 for Delaunay graph reorderings

Graph Name	AMD	BOBA	DynaDeg	Gray	RCM	Rabbit	Random	SBURN
delaunay_n21	**1.00**	**1.00**	0.62	0.78	**1.00**	**1.00**	0.68	**1.00**
delaunay_n22	**1.00**	0.96	0.74	0.92	**1.00**	**1.00**	0.54	**1.00**

Table 4.13 TopHits@100 for Delaunay graph reorderings

Graph Name	AMD	BOBA	DynaDeg	Gray	RCM	Rabbit	Random	SBURN
delaunay_n21	**1.00**	**1.00**	0.65	0.79	**1.00**	**1.00**	0.70	**1.00**
delaunay_n22	**1.00**	0.96	0.74	0.91	**1.00**	**1.00**	0.64	**1.00**

4.3.5 AUC and TopHits@K - Random Geometric Graphs

Table 4.14 AUC scores for random geometric graph reorderings

Graph Name	AMD	BOBA	DynaDeg	Gray	RCM	Rabbit	Random	SBURN
rgg_n_2_19_s0	**0.999**	**0.997**	0.923	0.943	**0.999**	**0.999**	0.923	**0.999**
rgg_n_2_20_s0	**0.999**	**0.998**	0.929	0.952	**0.999**	**0.999**	0.929	**0.999**

Similarly, random geometric graphs benefit greatly from reordering when learning embeddings, though they are inherently easier for models to understand. Even with a random node order, the spatial coherence of these graphs ensures that connected nodes are generally close to each other, resulting in good embedding performance. When reorderings are applied, the spatial locality is further emphasized, enhancing the model's ability to capture local relationships and neighborhoods effectively. Consequently, even algorithms that struggle with other graph types show marked improvements with random geometric graphs, achieving impressive results regardless of the specific reordering used. Even with a graph type that is easily learned, the improvement that is obtained by using reordered graphs is significant and helps show the consistent performance gains achieved by reordering for link prediction tasks, as can be seen in Table 4.14. These findings are further emphasized by Tables 4.15, 4.16 and 4.17 as the TopHits@K results for each graph is consistent with the performance of reorderings shown in Table 4.14.

Table 4.15 TopHits@20 for random geometric graph reorderings

Graph Name	AMD	BOBA	DynaDeg	Gray	RCM	Rabbit	Random	SBURN
rgg_n_2_19_s0	**1.00**	**1.00**	0.85	**0.95**	**1.00**	**1.00**	0.75	**1.00**
rgg_n_2_20_s0	**1.00**	**1.00**	**0.90**	0.85	**1.00**	**1.00**	**0.95**	**1.00**

Table 4.16 TopHits@50 for random geometric graph reorderings

Graph Name	AMD	BOBA	DynaDeg	Gray	RCM	Rabbit	Random	SBURN
rgg_n_2_19_s0	**1.00**	**1.00**	0.80	**0.95**	**1.00**	**1.00**	0.76	**1.00**
rgg_n_2_20_s0	**1.00**	**1.00**	**0.88**	0.88	**1.00**	**1.00**	**0.88**	**1.00**

Table 4.17 TopHits@100 for random geometric graph reorderings

Graph Name	AMD	BOBA	DynaDeg	Gray	RCM	Rabbit	Random	SBURN
rgg_n_2_19_s0	**1.00**	**1.00**	0.80	0.87	**1.00**	**1.00**	0.82	**1.00**
rgg_n_2_20_s0	**1.00**	**1.00**	0.86	0.88	**1.00**	**1.00**	**0.89**	**1.00**

4.4 Embedding Speed

To measure the effects of our part-skipping strategy on embedding speed, we run the augmented version of GOSH with four different experiment setups. Three of them use a fixed probability of skipping a region for a given round of embedding. For all three, the nonzero (edge count) ratio threshold for skipping the part is 10% of the total edges. If a part has at least 10% of the total amount of edges inside it it won't be skipped. Our most forgiving setup has a probability of 0.2 of skipping a part if it has less than 10% of the edges. The other strategies get more aggressive as one of them sets 0.5 as the probability and the most aggressive one has 0.8 probability of skipping an eligible part. One of the setups use a dynamic probability, using the ratio of non-zeros contained by a given region as the probability value. We present the results of these experiments performed on `soc-sinaweibo`, `kmer_P1a`, `rgg_n_2_24_s0` and `delaunay_n24`.

Figure 4.3 Embedding speed measurements on three different part skipping strategies. As the skipping probability increases, the orderings that can skip parts get faster while the random ordering embedding speed stays about the same. Graph: `soc-sinaweibo`, $d = 256$

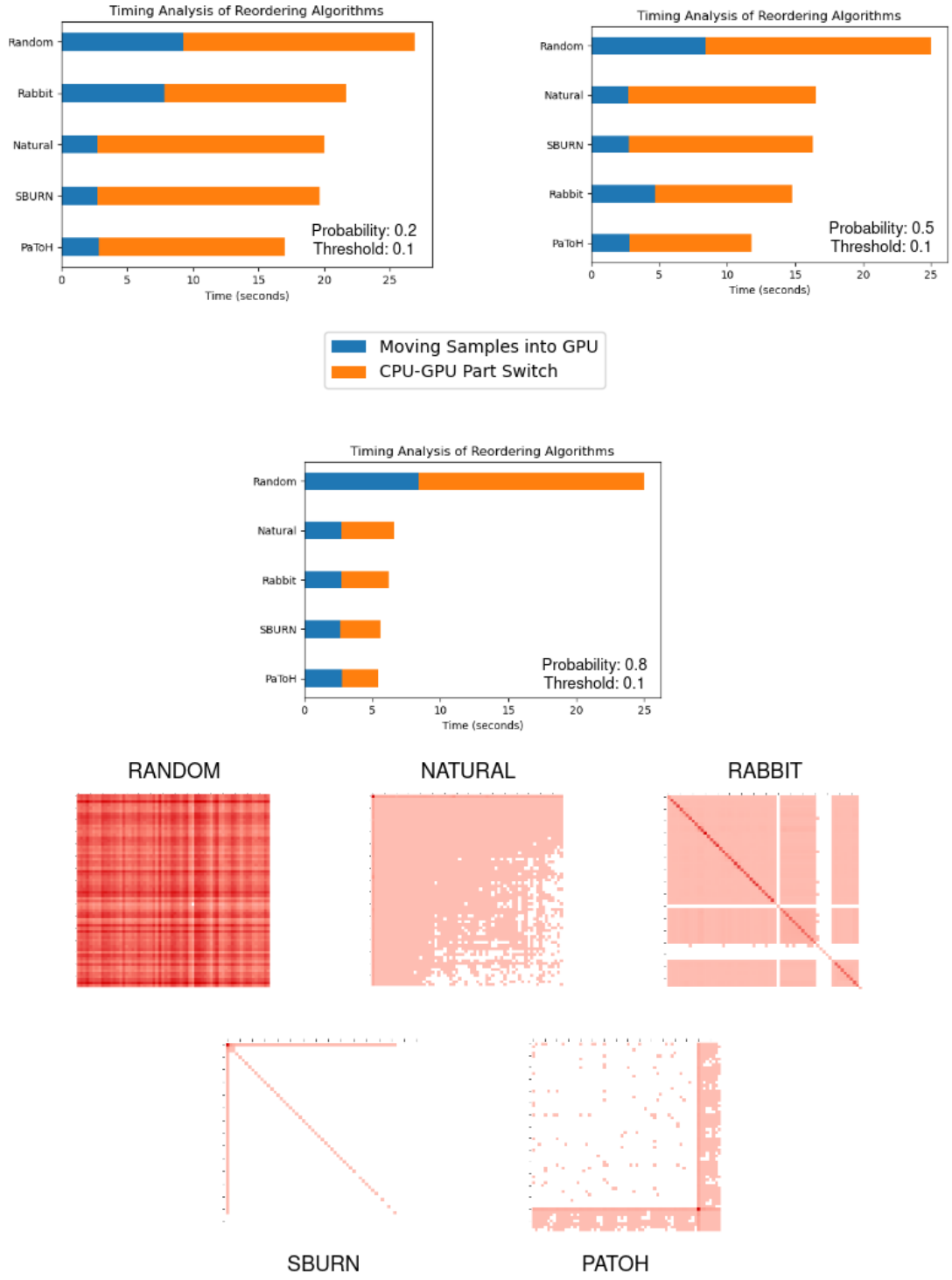
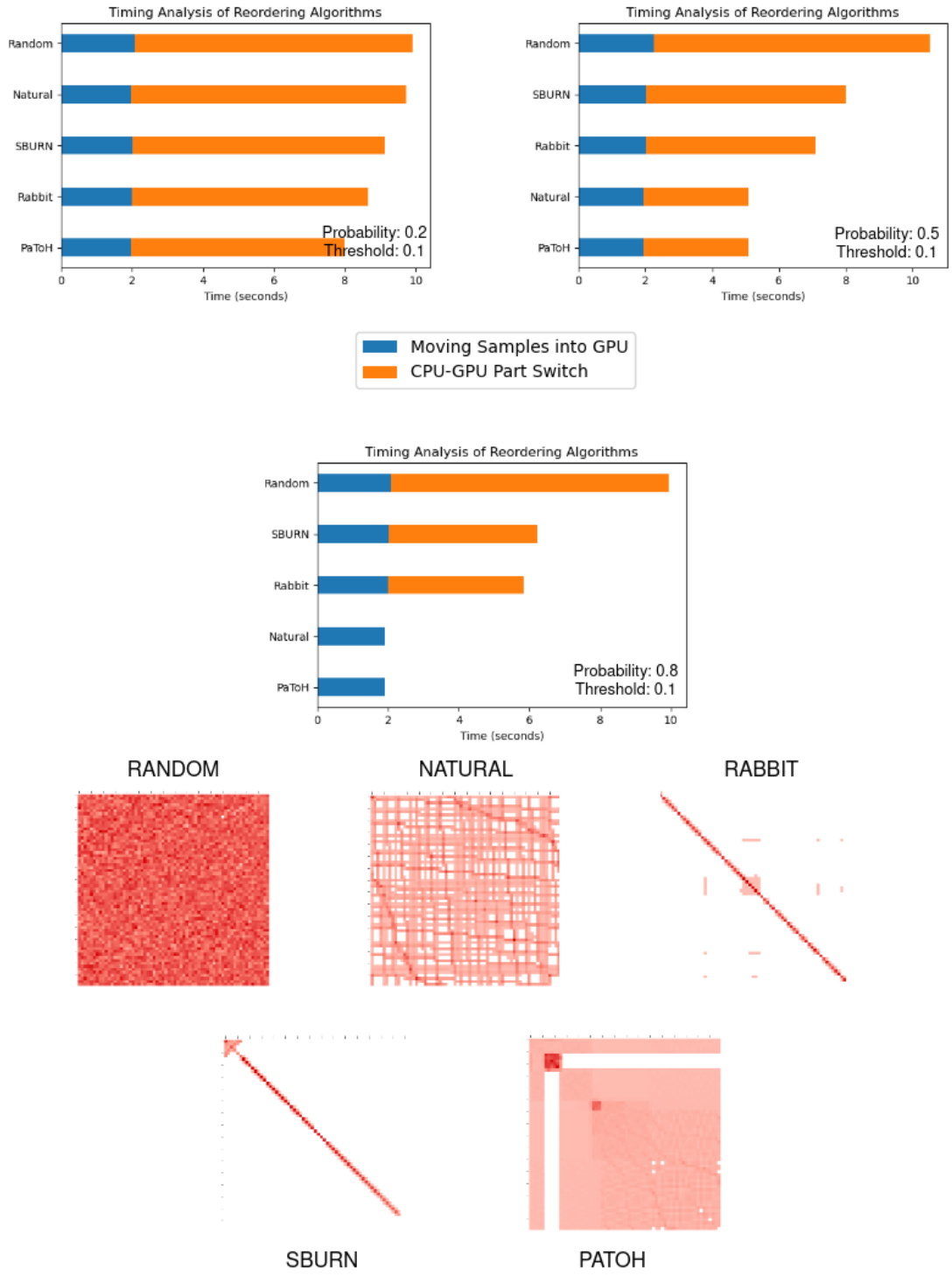


Figure 4.3 show that the part-skipping strategy accelerates the embedding process of GOSH for `soc-sinaweibo`, with the speedup becoming more apparent as the skipping probability increases. For `soc-sinaweibo`, we set d embedding vector

dimension, as 256. When the total node count of `soc-sinaweibo` is considered, the embedding matrix would have a size of *60.06 gigabytes*. This puts into perspective why one may need to carry data in and out of the GPU as most GPUs don't have enough memory to contain such data at once.

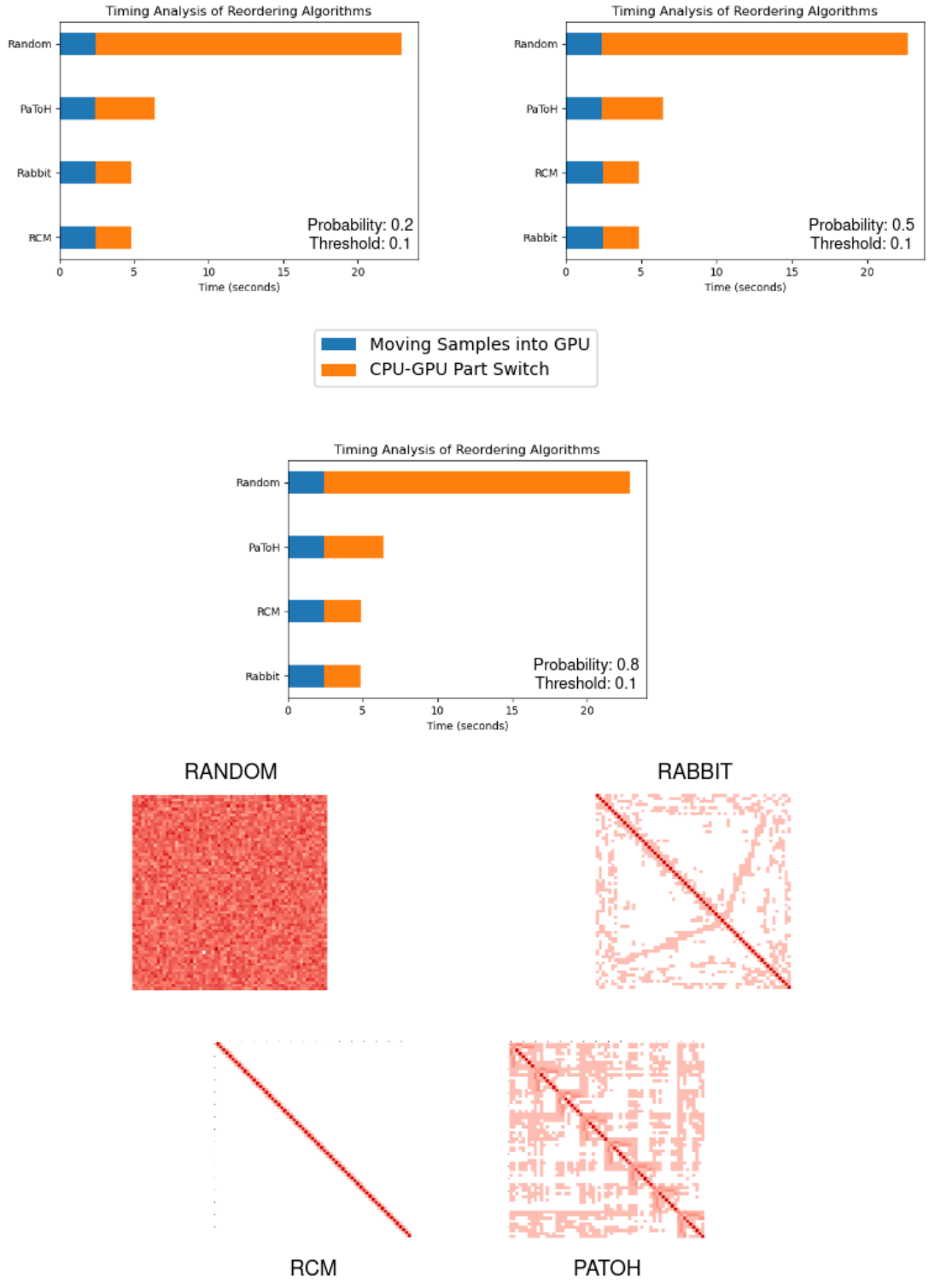
Figure 4.4 Embedding speed measurements on three different part skipping strategies. As the skipping probability increases, the orderings that can skip parts get faster while the random ordering embedding speed stays about the same. Graph: `kmer_P1a`, $d = 64$



With a graph that has fewer edges and more nodes in `kmer_P1a`, we can create more regions that are sparse, hence a lot more "skippable". We can see why this is relevant for the skipping strategy looking at Figure 4.4. The random ordering still

produces the worst results as there are no part skips there, but the most interesting result comes from the setup with the highest skip probability. Since there aren't many dense regions inside the natural ordered and the PaToH ordered graphs, there are no part switches performed for those two orderings. The parts needed by the embedding process are only carried into the GPU once and then the embeddings are learned only on those parts.

Figure 4.5 Embedding speed measurements on three different part skipping strategies. As the skipping probability increases, the orderings that can skip parts get faster while the random ordering embedding speed stays about the same. Graph: `delaunay_n24`, $d = 1024$

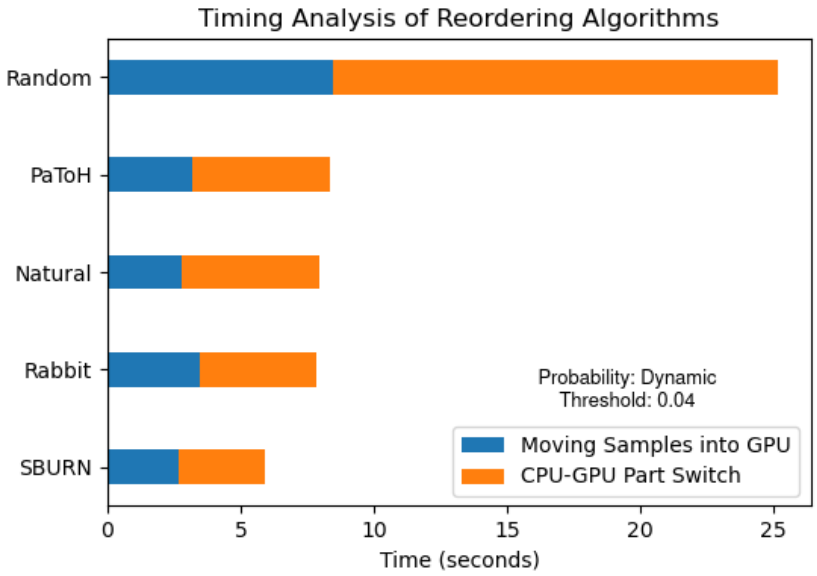


The contrast between Figure 4.4 and Figure 4.5 shows that even if the embedding vector dimensions are very small or very large, as long as part switches are needed

due to the memory constraints of a device the part skipping strategy accelerates the process by a significant amount.

Another way to approach this problem is to make the probability of skipping a part dynamic. We can use the non-zero ratio of the part in question directly as the probability of skipping that part if it does not have the required amount of non-zeros to be directly included in the process. This would allow us to be even more aggressive while also giving ourselves a chance at including the parts that might have just fallen short of the threshold given.

Figure 4.6 Embedding speed measurements with dynamic probability scheme. Graph: `soc-sinaweibo`



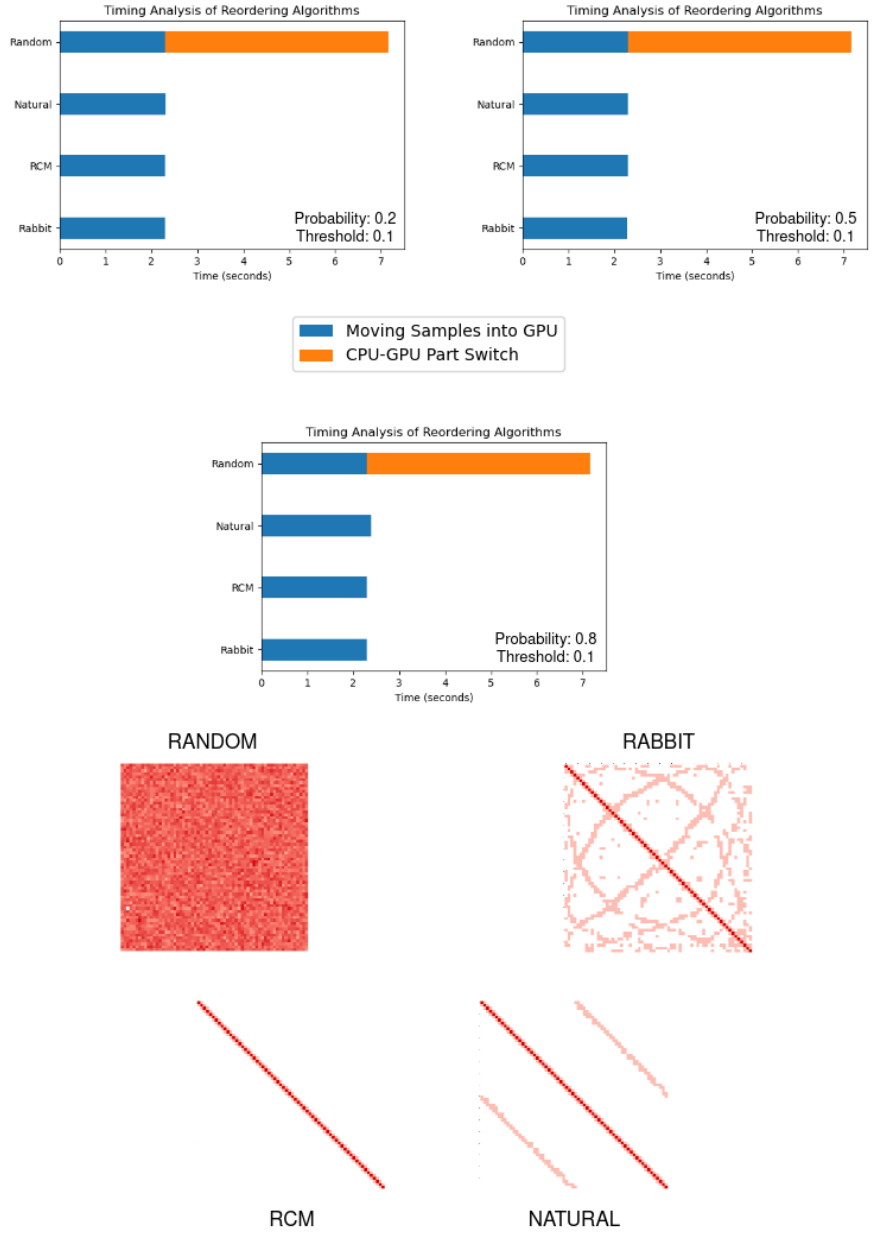
We can see in Figure 4.6 that if we lower the threshold with dynamic probability scheduling we can achieve similar execution times to the fixed probability setting while also including more information in the process by way of allowing smaller partitions to be included automatically as well. With this, we can treat the probability and threshold selection as a trade-off between execution time and the amount of information allowed inside the process.

To show that this process does not affect the quality of the resulting embedding matrix, we present link prediction results on `rgg_n_2_24_s0` in Table 4.18 together with the speedup results in Figure 4.7.

Table 4.18 ROC-AUC values of a logistic regression model trained on various orderings of `rgg_n_2_24_s0` for link prediction.

Skip Probability	0.2	0.5	0.8	No Skip
Random	0.740	0.739	0.738	**0.739**
Natural	**0.975**	**0.975**	**0.975**	**0.975**
Rabbit	**0.974**	**0.974**	0.973	**0.976**
RCM	**0.976**	0.973	**0.976**	**0.976**

Figure 4.7 Embedding speed measurements on three different part skipping strategies. As the skipping probability increases, the orderings that can skip parts get faster while the random ordering embedding speed stays about the same. Graph: `rgg_n_2_24_s0`, $d = 512$



5. CONCLUSION

This work shows the effects of reordering matrices representing graphs on node embedding speed and quality. We propose several scheduling strategies for positive sampling on GOSH, a node embedding framework that can learn embeddings from arbitrarily large graphs on a single GPU. Taking inspiration from previous work, we introduced different scheduling schemes with the aim of eliminating certain parts of the graph by skipping them with a certain probability during the sampling process. To create regions in the graph that are less meaningful to the embedding process and therefore more sensible to skip, we reordered the graphs using various reordering algorithms. The sampling schemes accelerated the overall embedding process and as the used scheme got more aggressive in terms of probability of skipping a part the acceleration became more pronounced. We also showed that skipping these regions with a certain probability does not affect the overall quality of the embeddings as the ROC-AUC values obtained by performing link prediction on `rgg_n_2_24_s0` are not distinguishable to a significant degree whether the used embedding matrix is produced by our part skipping scheme or the default setting.

We also showed the overall effects of reordering on embedding quality by testing LINE and DeepWalk embedding algorithms on reordered graphs and using the learned embeddings on link prediction. Reordering algorithms that emphasize the community structures within graphs or help make the planar properties more apparent to the embedding learners performed better than algorithms which focus on how parallelizable the algorithm is or on preserving or enhancing sub-structures within the graphs. The learners showed poor performance on road graphs as they are the ones with the most complex connectivity structures. Reordering still had a positive effect on road graphs as most reordering algorithms lead to improvements on AUC scores. Delaunay graphs and especially random geometric graphs were very easy to learn for learners even when they were randomly ordered but this made the improvements obtained by reordering the graphs more apparent. Overall, most reordering algorithms helped the embedding process by improving the quality of the embeddings for all of the different types of graphs used in this thesis.

Improvements to this thesis are possible. The quality of the embeddings could be tested on many other downstream tasks where node embeddings are used such as node label prediction. The work could be expanded to show the effects of reordering on edge embedding or sub-graph embedding. Another possible improvement is introducing variety to the part skipping strategy for GOSH. The probability choice could be dependent on many other functions of the node or edge count or the non-zero ratio of a given region. Reordering algorithms could be used in the initialization stage for embeddings by choosing initial values for vectors representing a certain node by looking at the order the node appears in the new ordering and determining a range for the possible values to put into the vector. This could emphasize the closeness of connected nodes even further and help improve the quality of the embeddings.

BIBLIOGRAPHY

Abu-El-Haija, S., Perozzi, B., Al-Rfou, R., & Alemi, A. (2018). Watch your step: Learning node embeddings via graph attention.

Ajazi, Fioralba (2018). *Random geometric graphs and their applications in neuronal modelling*. PhD thesis, Lund University.

Akyildiz, T. A., Alabsi Aljundi, A., & Kaya, K. (2020). Gosh: Embedding big graphs on small hardware. In *49th International Conference on Parallel Processing - ICPP*, ICPP '20. ACM.

Albert, R. & Barabási, A.-L. (2002). Statistical mechanics of complex networks. *Rev. Mod. Phys.*, *74*, 47–97.

Amestoy, P., Davis, T., & Duff, I. (2004). Algorithm 837: Amd, an approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw.*, *30*, 381–388.

Arai, J., Shiokawa, H., Yamamuro, T., Onizuka, M., & Iwamura, S. (2016). Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, (pp. 22–31).

Balaji, V. & Lucia, B. (2018). When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs. (pp. 203–214).

Barthélemy, J. & Carletti, T. (2017). A dynamic behavioural traffic assignment model with strategic agents. *Transportation Research Part C: Emerging Technologies*, *85*, 23–46.

Cai, H., Zheng, V. W., & Chang, K. C.-C. (2018). A comprehensive survey of graph embedding: Problems, techniques and applications.

Çatalyürek, Ü. & Aykanat, C. (2011). *PaToH (Partitioning Tool for Hypergraphs)*, (pp. 1479–1487). Boston, MA: Springer US.

Che, W., Liu, Z., Wang, Y., & Liu, J. (2023). Merit: multi-level graph embedding refinement framework for large-scale graph. *Complex & Intelligent Systems*, *10*(1), 1303–1318.

Coleman, B., Segarra, S., Shrivastava, A., & Smola, A. (2021). Graph reordering for cache-efficient near neighbor search.

Cuthill, E. & McKee, J. (1969). Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*, ACM '69, (pp. 157–172)., New York, NY, USA. Association for Computing Machinery.

Dakowicz, M. & Gold, C. (2003). Visualizing terrain models from contours–plausible ridge, valley and slope estimation.

Dash, N. (2008). Context and contextual word meaning. *Journal of Theoretical Linguistics*, *5*.

Davis, T. A. & Hu, Y. (2011). The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, *38*(1).

Dehghan, A., Kamiński, B., Kraiński, L., Pralat, P., & Theberge, F. (2021a). Evaluating node embeddings of complex networks.

Dehghan, A., Kamiński, B., Kraiński, L., Pralat, P., & Theberge, F. (2021b). Evaluating node embeddings of complex networks.

Drescher, M., Awad, M. A., Porumbescu, S. D., & Owens, J. D. (2023). Boba: A

parallel lightweight graph reordering algorithm with heavyweight implications.
Estrada, E., Meloni, S., Sheerin, M., & Moreno, Y. (2016). Epidemic spreading in random rectangular networks. *Physical Review E*, *94*(5).
George, A. & Liu, J. (1981). *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall series in computational mathematics. Prentice-Hall.
George, A. & Liu, J. W. (1979). An implementation of a pseudoperipheral node finder. *ACM Transactions on Mathematical Software (TOMS)*, *5*(3), 284–295.
George, A. & Liu, J. W. H. (1989). The evolution of the minimum degree ordering algorithm. *SIAM Rev.*, *31*, 1–19.
Gharaee, Z., Kowshik, S., Stromann, O., & Felsberg, M. (2021). Graph representation learning for road type classification. *Pattern Recognition*, *120*, 108174.
Gilbert, E. N. (1961). Random plane networks. *Journal of the Society for Industrial and Applied Mathematics*, *9*(4), 533–543.
Grover, A. & Leskovec, J. (2016). node2vec: Scalable feature learning for networks.
He, J. & Chu, W. (2010). *A social network-based recommender system (SNRS)*, volume 12, (pp. 47–74).
Jin, E. M., Girvan, M., & Newman, M. E. J. (2001). Structure of growing social networks. *Phys. Rev. E*, *64*, 046132.
Kang, D.-S., Kim, Y.-J., & Shin, B.-S. (2006). Efficient large-scale terrain rendering method for real-world game simulation. In Pan, Z., Aylett, R., Diener, H., Jin, X., Göbel, S., & Li, L. (Eds.), *Technologies for E-Learning and Digital Entertainment*, (pp. 597–605)., Berlin, Heidelberg. Springer Berlin Heidelberg.
Karaman, S. & Frazzoli, E. (2011). Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research*, *30*(7), 846–894.
Liben-nowell, D. & Kleinberg, J. (2003). The link prediction problem for social networks. *Journal of the American Society for Information Science and Technology*, *58*.
Lim, Y., Kang, U., & Faloutsos, C. (2014). Slashburn: Graph compression and mining beyond caveman communities. *Knowledge and Data Engineering, IEEE Transactions on*, *26*, 3077–3089.
Marchet, C., Iqbal, Z., Gautheret, D., Salson, M., & Chikhi, R. (2020). REINDEER: efficient indexing of k-mer presence and abundance in sequencing datasets. *Bioinformatics*, *36*(Supplement_1), i177–i185.
Martinez Santa, F., Martínez, F., & Gómez, E. (2014). Using the delaunay triangulation and voronoi diagrams for navigation in observable environments. *Revista Tecnura*, *18*.
Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space.
Perozzi, B., Al-Rfou, R., & Skiena, S. (2014). Deepwalk: online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '14. ACM.
Ribeiro, E., Macambira, M., & Neiva, E. R. (2017). *Social Network Analysis in Organizations as a Management Support Tool*, (pp. 243–265). Springer International Publishing.
Rossi, R. A. & Ahmed, N. K. (2015). The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*.
Rossi, R. A. & Ahmed, N. K. (2016). An interactive data repository with visual

analytics. *SIGKDD Explor.*, *17*(2), 37–41.

Song, L. & Florea, L. (2015). Rcorrector: Efficient and accurate error correction for illumina rna-seq reads. *GigaScience*, *4*(1). Cited by: 311; All Open Access, Gold Open Access, Green Open Access.

Stattner, E. & Vidot, N. (2011). Social network analysis in epidemiology: Current trends and perspectives. (pp. 1 – 11).

Tang, J., Qu, M., Wang, M., Zhang, M., Yan, J., & Mei, Q. (2015). Line: Large-scale information network embedding. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15. International World Wide Web Conferences Steering Committee.

Tiago Pimentel, Adriano Veloso, N. Z. (2018). Fast node embeddings: Learning ego-centric representations.

Wang, D., Cui, P., & Zhu, W. (2016). Structural deep network embedding. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, (pp. 1225–1234)., New York, NY, USA. Association for Computing Machinery.

Wang, Y., Yao, Y., Tong, H., Xu, F., & Lu, J. (2019). Discerning edge influence for network embedding. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, CIKM '19, (pp. 429–438)., New York, NY, USA. Association for Computing Machinery.

Yuan, J., Wang, H., & Fang, Y. (2023). Identification of critical links in urban road network based on gis. *Sustainability*, *15*(20).

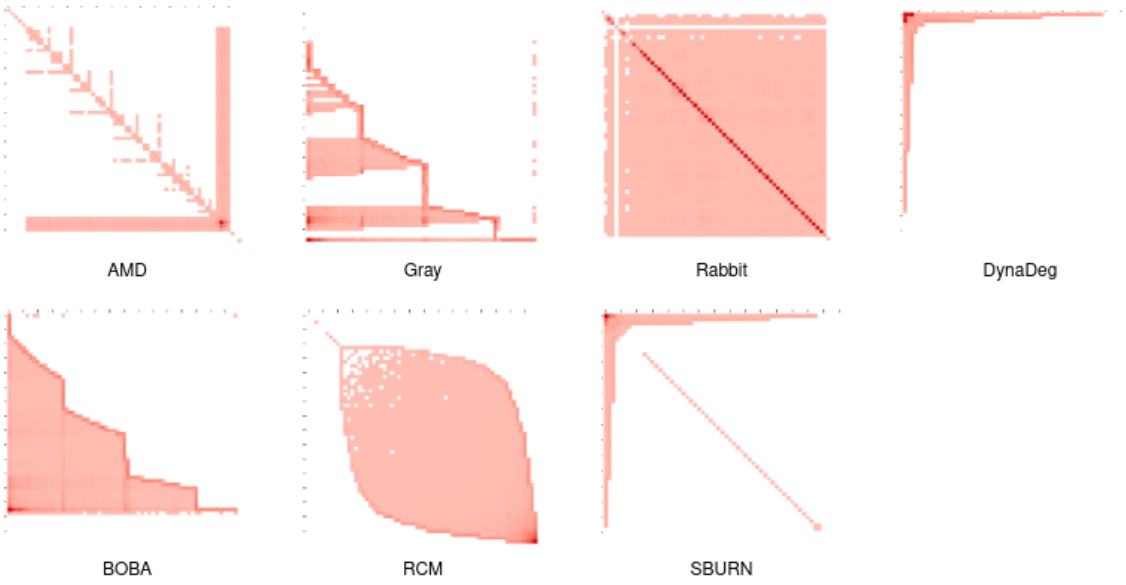
Zhao, H., Xia, T., Li, C., Zhao, W., Zheng, N., & Ren, P. (2020). Exploring better speculation and data locality in sparse matrix-vector multiplication on intel xeon. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, (pp. 601–609).

Zhao, K., Rong, Y., Yu, J. X., Huang, J., & Zhang, H. (2020). Graph ordering: Towards the optimal by learning.

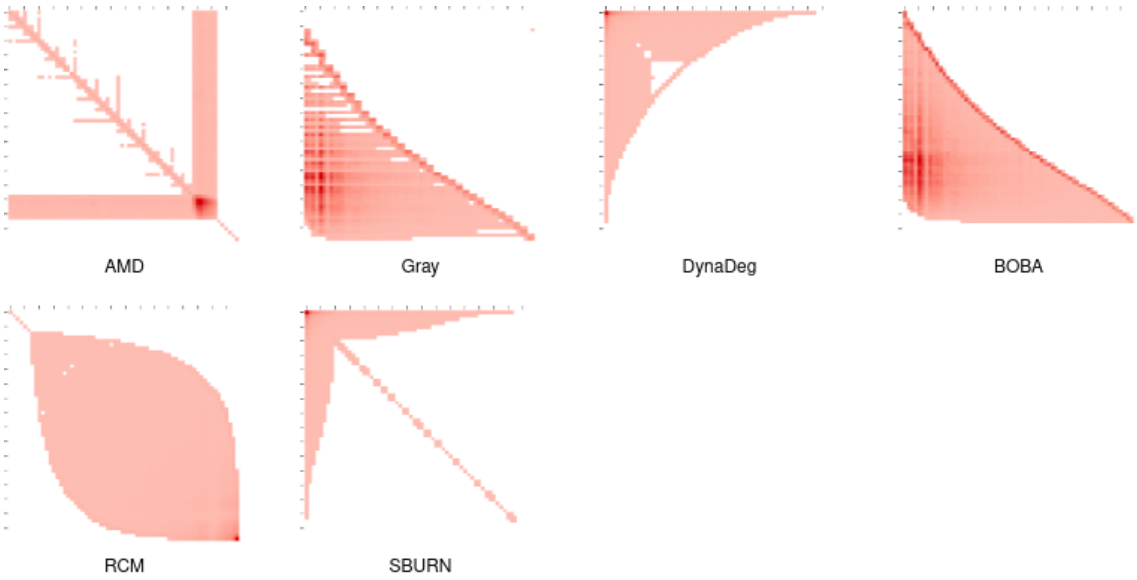
Zhu, Z., Xu, S., Tang, J., & Qu, M. (2019). Graphvite: A high-performance cpu-gpu hybrid system for node embedding. In *The World Wide Web Conference*, WWW '19. ACM.

APPENDIX A

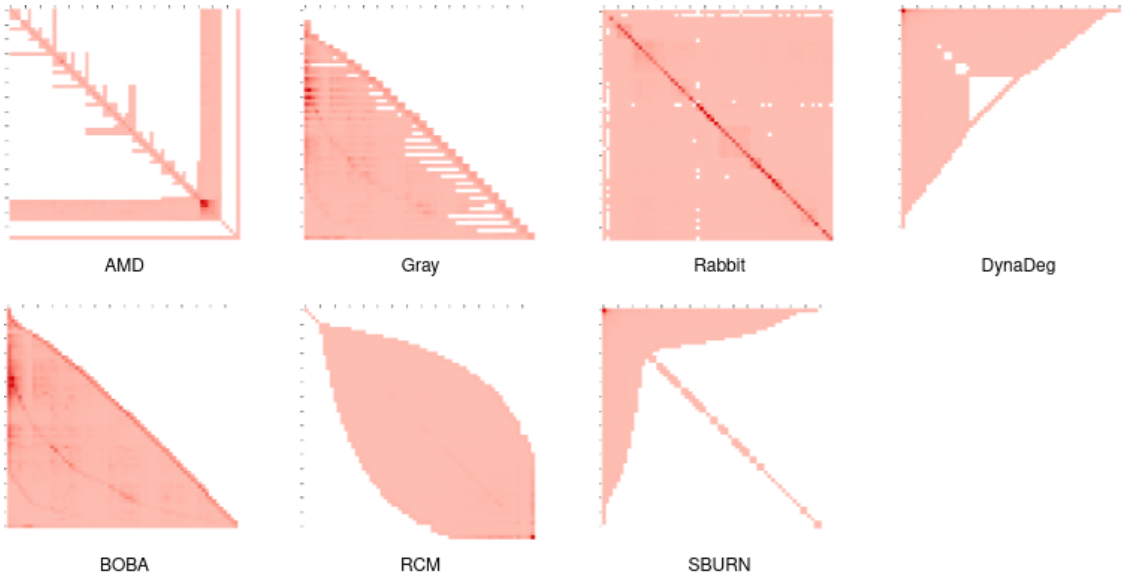
Reordered states of soc-douban



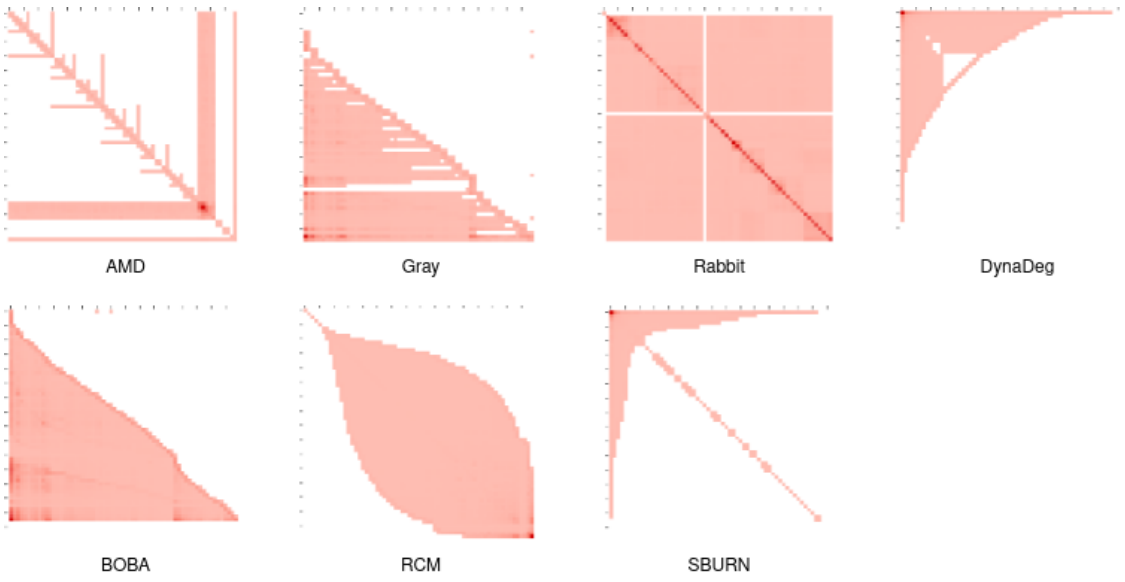
Reordered states of soc-slashdot



Reordered states of soc-gowalla

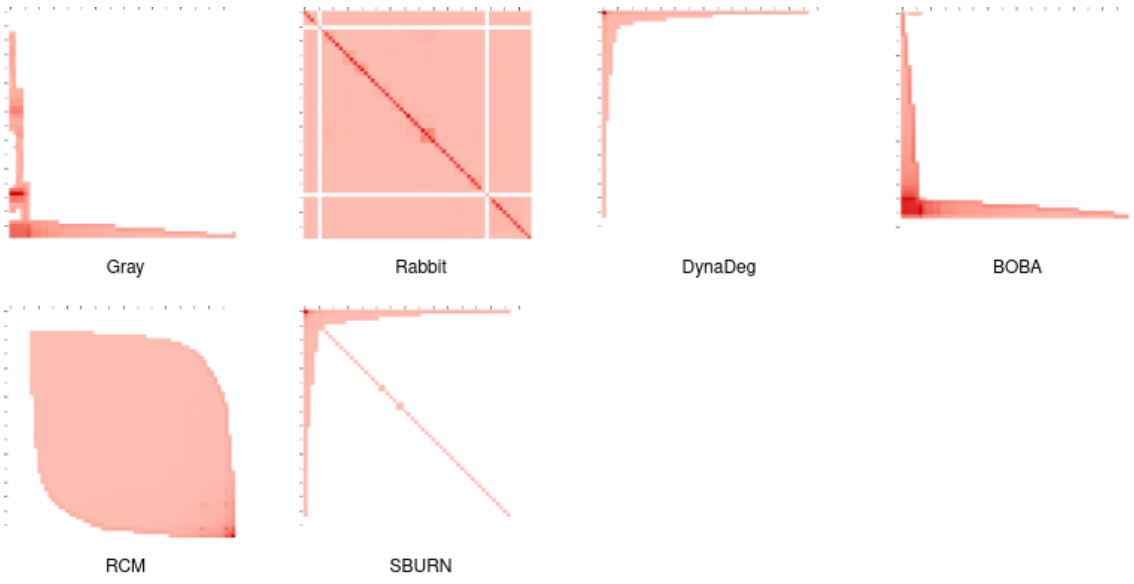


Reordered states of soc-youtube

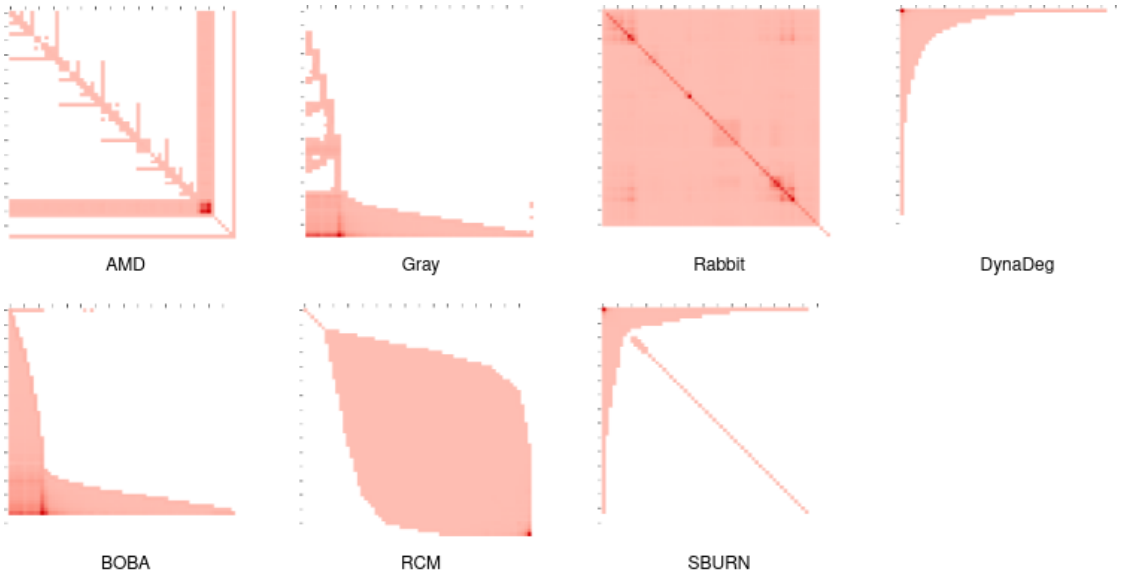


Reordered states of soc-buzznet

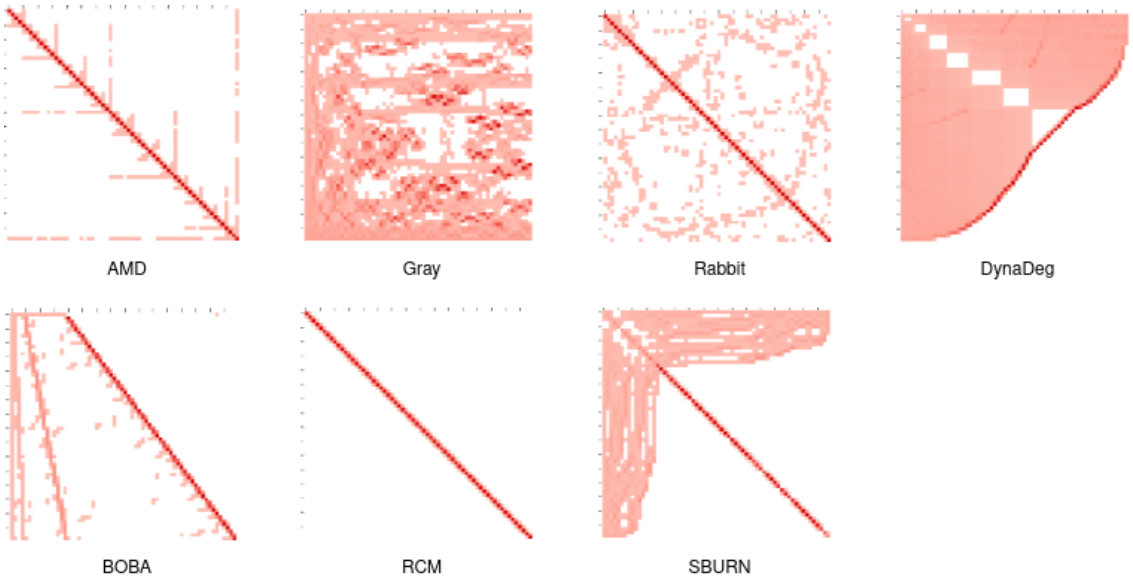
Reordered states of soc-lastfm



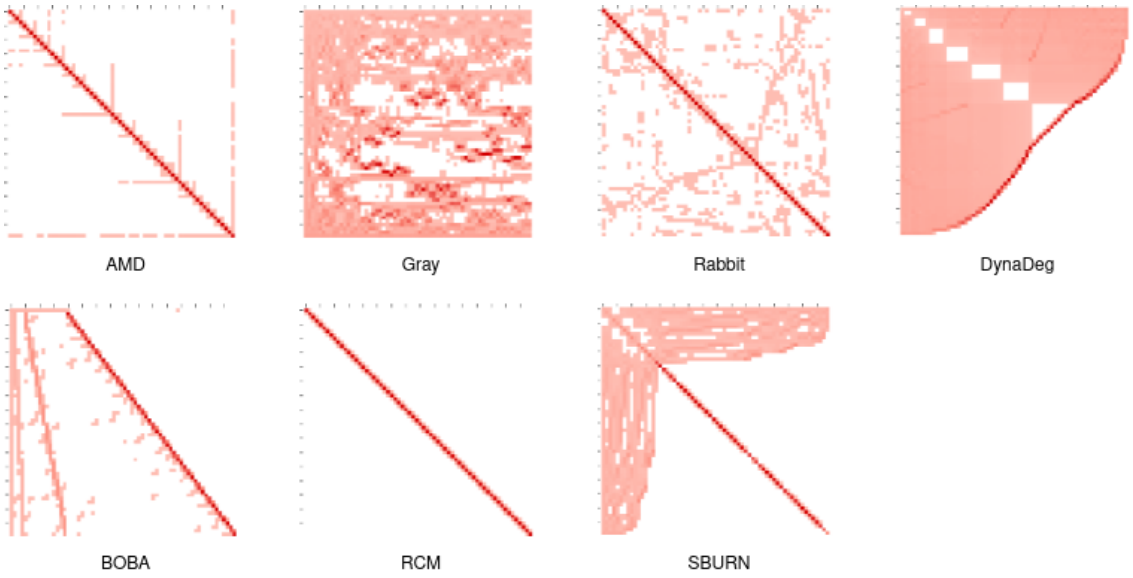
Reordered states of soc-digg



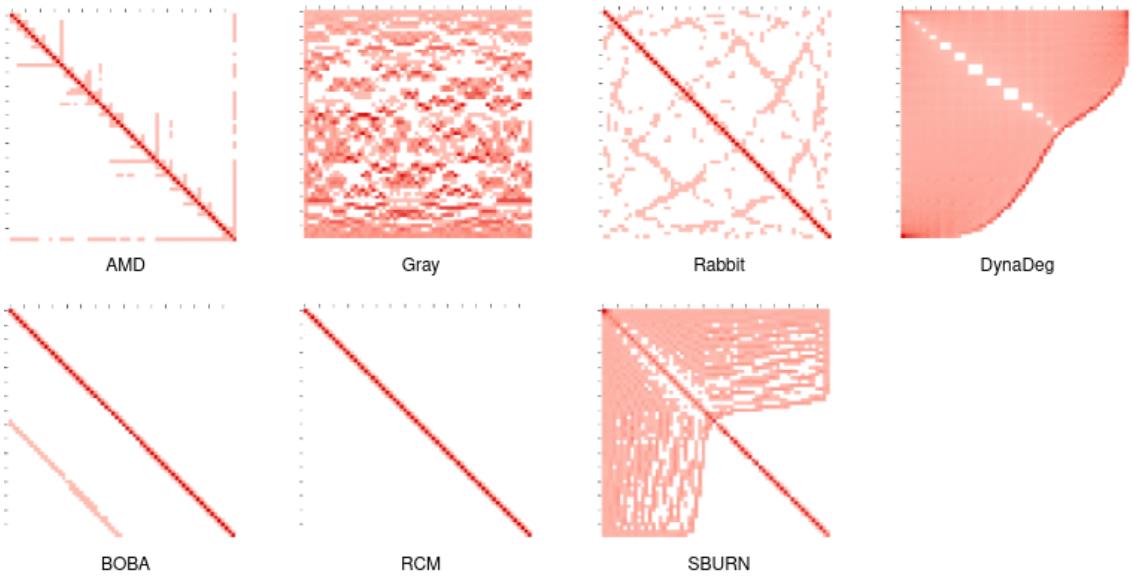
Reordered states of delaunay_n21



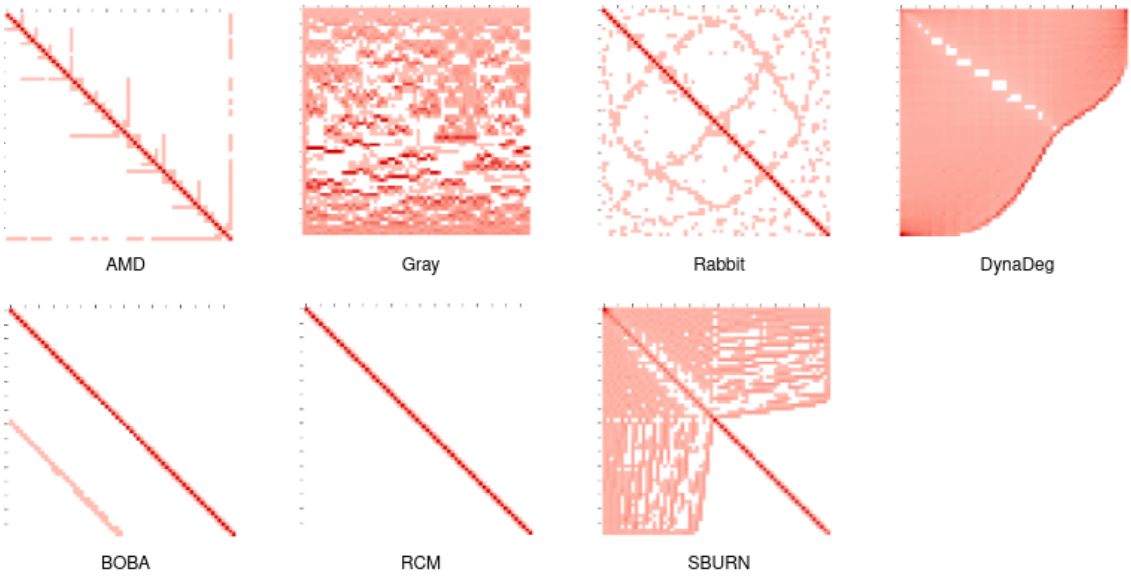
Reordered states of delaunay_n22



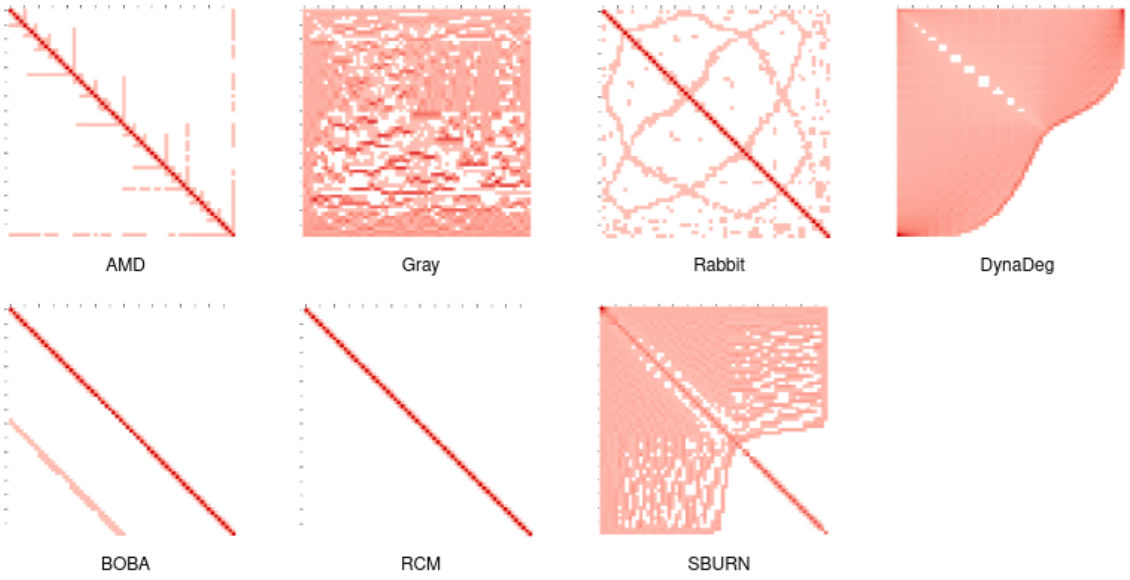
Reordered states of rgg_n_2_19_s0



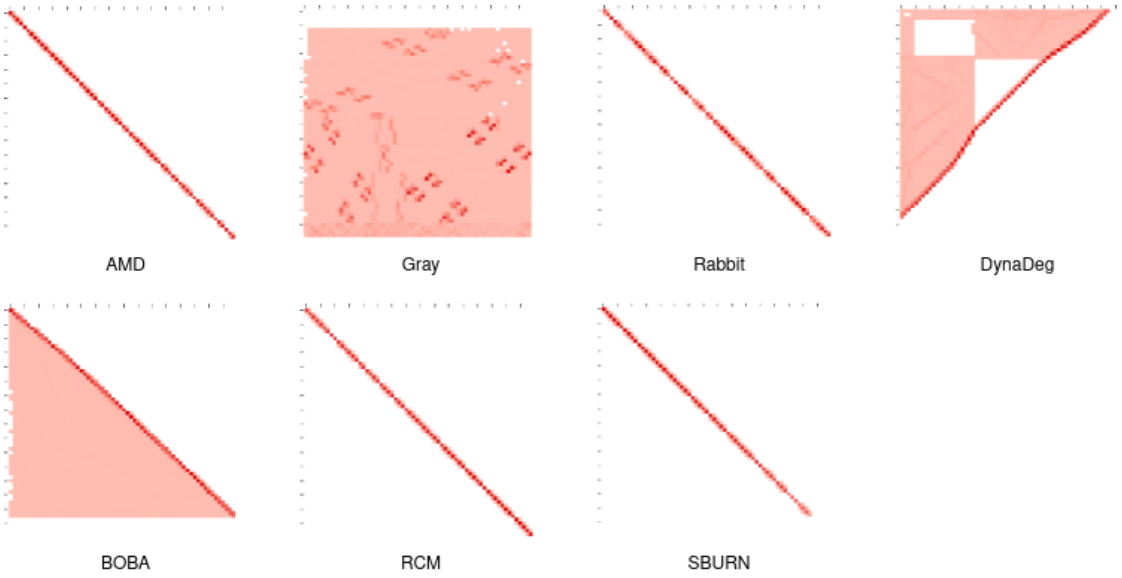
Reordered states of rgg_n_2_20_s0



Reordered states of rgg_n_2_23_s0



Reordered states of road-belgium-osm



Reordered states of road-italy-osm