

Implicit Graph Theory: Scaling Graph Mining with Vector Arithmetic

by

Tengkai Yu

Hon. B.Sc., University of Toronto, 2016

M.Sc., University of Victoria, 2020

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

© Tengkai Yu, 2026

University of Victoria

All rights reserved. This dissertation may not be reproduced in whole or in part, by
photocopying or other means, without the permission of the author.

We acknowledge and respect the Lək'wəŋən (Songhees and X^wsepsəm/Esquimalt)
Peoples on whose territory the university stands, and the Lək'wəŋən and W̱SÁNEĆ
Peoples whose historical relationships with the land continue to this day.

Implicit Graph Theory: Scaling Graph Mining with Vector Arithmetic

by

Tengkai Yu

Hon. B.Sc., University of Toronto, 2016

M.Sc., University of Victoria, 2020

Supervisory Committee

Dr. Alex Thomo, Co-Supervisor
(Department of Computer Science)

Dr. Venkatesh Srinivasan, Co-Supervisor
(Department of Computer Science)

Dr. Mu-Hsing Kuo, Outside Member
(School of Health Information Science)

ABSTRACT

The rapid ascent of representation learning has transformed how we model complex systems, replacing discrete identifiers with continuous, high-dimensional vectors. In modern pipelines, relationships between entities, whether users in a recommender system or text chunks in a retrieval corpus, are defined not by explicit links but by geometric proximity in a dense vector space. Because every pair of embeddings yields a non-zero similarity, the induced graph is fully connected by construction, containing $O(N^2)$ weighted edges that no standard adjacency representation can store at scale. Traditional graph mining algorithms, designed for sparse networks, remain tied to the “Explicit Paradigm” of materializing this adjacency matrix before performing computation. For the dense semantic graphs produced by modern embedding pipelines, this imposes a quadratic $O(N^2)$ memory bottleneck that renders standard methods such as Label Propagation and Louvain intractable beyond a few thousand nodes. Common workarounds, such as k -nearest-neighbour pruning, discard precisely the long-range structure that semantic embeddings are meant to capture.

This dissertation introduces **Implicit Graph Theory**, a unified framework that redefines the graph not as a static container of edges, but as a linear algebraic operator defined by the inner product of node embeddings ($\mathbf{A} = \mathbf{V}\mathbf{V}^\top$). By exploiting the low-rank structure of semantic data, we derive exact matrix-vector formulations for fundamental graph primitives that operate in linear memory in N , without approximation error relative to the inner-product adjacency model.

We validate this framework through three contributions. First, we introduce **Vector-Based Label Propagation (VLP)**, which addresses the memory wall for semi-supervised learning on dense graphs. We prove that VLP is mathematically equivalent to explicit label propagation and show that it scales to 1.5 million nodes on a single GPU, in a setting where standard dense-graph libraries exhaust memory beyond roughly 15,000 nodes. Second, we present **VLouvain**, a structure-discovery algorithm that detects hierarchical communities directly from embedding vectors, without ever generating an edge list. The key result, the Vector Supernode Theorem, shows that modularity-driven graph coarsening reduces to vector addition, reproducing the modularity scores of standard Louvain exactly. VLouvain clusters 1.57 million product embeddings in minutes on a single GPU, a scale at which cuGraph and NetworKit exhaust memory entirely. Finally, we apply these principles to retrieval-augmented generation with **GraphRAG-V**, which uses VLouvain to or-

ganize text-chunk embeddings into semantic communities, replacing the expensive LLM-based entity extraction pipeline used by prior graph-RAG systems. On a multi-hop reasoning benchmark, GraphRAG-V reduces indexing time from 3.0 hours to 5.3 minutes ($34\times$) while outperforming both a vector-store baseline and Microsoft’s GraphRAG in answer recall.

Together, these results argue for a different way of scaling graph mining to the era of dense representations: rather than storing the graph, we compute it on demand from the embeddings that already exist.

Contents

Supervisory Committee	ii
Abstract	iii
Table of Contents	v
List of Tables	x
List of Figures	xii
List of Mathematical Symbols	xiii
Acknowledgements	xiv
Dedication	xv
1 Introduction	1
1.1 The Graph Paradox	1
1.2 The Key Insight: Low-Rank Dense Graphs	2
1.3 Contributions	3
1.4 Outline	4
2 Background and Related Work	6
2.1 Graph Theory Primitives	6
2.1.1 The Graph Laplacian	7
2.2 Community Detection	8
2.2.1 Modularity Maximization	8
2.2.2 The Resolution Limit	9
2.2.3 The Louvain Algorithm	10
2.3 Label Propagation	12

2.3.1	Discrete “Voting” Label Propagation	13
2.3.2	Continuous “Linear” Label Propagation	13
2.3.3	The Density Penalty	14
2.4	Graph Sparsity vs. Density	15
2.5	Vector Embeddings	15
2.5.1	From Sparse to Dense Representations	15
2.5.2	Retrieval-Augmented Generation (RAG) and Vector Search . .	16
2.6	Implicit Graph Theory	17
2.6.1	The Bridge: Connecting Algebra to Geometry	17
2.6.2	The Gap: Exactness at Scale	18
3	Vector-Based Label Propagation	20
3.1	Introduction	20
3.2	Vector-Based Label Propagation (VLP)	21
3.2.1	Recap of the Classical Update	21
3.2.2	From Explicit Adjacency to Vector Factorization	22
3.2.3	Vector-Based Update Rule	23
3.2.4	Exactness	25
3.2.5	Complexity Analysis	25
3.3	Convergence Analysis	26
3.3.1	Practical Implications: Consensus is Trivial	28
3.4	Experiments	29
3.4.1	Runtime Results	31
3.4.2	Empirical Verification of Exactness	32
3.4.3	Sensitivity to the Embedding Dimension	33
3.5	Chapter Summary	33
4	VLouvain: Structure Discovery	35
4.1	Introduction	35
4.2	Vector-Based Louvain: Key Operations and Data Structures	37
4.2.1	Degree Calculation	38
4.2.2	Sum of Edge Weights	38
4.2.3	Affinity and Resolution Terms	38
4.2.4	Modularity Gain Calculation	39
4.2.5	Community Movement and Hardware-Aware State	40

4.3	The VLouvain Algorithm	41
4.3.1	Optimized One-Level Function	41
4.3.2	Optimized New Graph Generation	43
4.3.3	Optimized Louvain Partition Algorithm	44
4.3.4	Time and Memory Cost	45
4.4	Theoretical Analysis: Exactness	46
4.5	Experiments	49
4.5.1	Modularity Check on Subsampled Graphs	51
4.5.2	Runtime Results	51
4.5.3	The Fidelity Gap: VLouvain vs. Top- K Sparsification	52
4.6	Qualitative Case Study: Semantic Discovery in FastText Embeddings	53
4.7	Chapter Summary	55
5	GraphRAG-V: Fast Multi-Hop Retrieval via Text-Chunk Communities	57
5.1	Introduction	57
5.2	Related Work	59
5.2.1	Vector-Store RAG	59
5.2.2	Graph-Based RAG Systems	59
5.2.3	Community Detection Without Explicit Graphs	60
5.2.4	Embedding and Indexing Foundations	60
5.2.5	Evaluation Practice	61
5.3	System Architecture: GraphRAG-V	61
5.3.1	Indexing Pipeline	61
5.3.2	Community Summarization	62
5.3.3	System Behaviour on MultiHopRAG	62
5.4	Dual-Path Retrieval	63
5.4.1	Local and Global Search	63
5.4.2	Complexity Analysis	66
5.5	Experimental Setup	66
5.5.1	Dataset and Benchmark	66
5.5.2	Systems and Baselines	67
5.5.3	Evaluation Metrics	67
5.5.4	Hardware Configuration	68
5.6	Experimental Evaluation	68

5.6.1	Efficiency and Cost Analysis	68
5.6.2	Quality Results by Question Type	69
5.6.3	Comparison with Microsoft GraphRAG	71
5.6.4	Qualitative Case Study	72
5.7	Discussion	73
5.7.1	Community Structure Without Compromise	73
5.7.2	Efficiency in Practice	74
5.7.3	Limitations and Future Work	74
5.8	Chapter Summary	76
6	Future Directions	77
6.1	Introduction	77
6.2	Scalable Industrial Implementations	77
6.2.1	Recommender Systems and Two-Tower Architectures	77
6.2.2	Real-Time Fraud and Anomaly Detection	78
6.3	Extending V Louvain: Directed and Multi-Modal Graphs	79
6.3.1	Directed V Louvain	79
6.3.2	Cross-Modal Community Discovery	80
6.4	Differentiable V Louvain and Deep Representation Learning	81
6.4.1	Overcoming the Softmax Bottleneck via Prototype Initialization	81
6.4.2	Topologically-Guided Fine-Tuning and GNN Pooling	82
6.5	Future AI Architectures: LLM Sparse Attention	82
7	Conclusion	83
7.1	Summary of Contributions	83
7.1.1	Exact Diffusion on Implicit Graphs (VLP)	83
7.1.2	Hierarchical Structure Discovery (V Louvain)	84
7.1.3	Topological Retrieval-Augmented Generation (GraphRAG-V)	84
7.2	The Unified Theory: The Graph as an Operator	84
7.3	Limitations	85
7.4	Final Remarks	85
A	Detailed Proofs for V Louvain Exactness	87
A.1	Notation and Conventions	87
A.2	Degree Equivalence (Lemma 4.3)	88
A.3	Modularity Gain Equivalence (Lemma 4.4)	89

A.3.1	Insertion Side	89
A.3.2	Removal Side	90
A.3.3	Net Gain	92
A.4	Aggregation Equivalence (Lemma 4.5)	92
A.5	From the Three Lemmas to the Exactness Theorem	94
A.6	Summary	94
B	VLouvain Determinism Check	96
	Bibliography	98

List of Tables

Table 3.1	Summary of the four datasets. The “ $N \times N$ entries” column indicates the number of cells that an explicit dense adjacency matrix would have to materialize.	29
Table 3.2	Runtime (in seconds) for Amazon over 1000 iterations. Bold denotes our VLP, always the fastest. ‘X’ indicates library failure. .	31
Table 3.3	Runtime (in seconds) for Yelp over 1000 iterations. VLP is always the fastest, in bold. ‘X’ indicates library failure.	31
Table 3.4	Runtime (in seconds) for Taobao over 1000 iterations. VLP times are in bold; ‘X’ denotes failure.	32
Table 3.5	Empirical exactness verification on a 10,000-node subsample of Amazon. Downstream metrics for VLP and the explicit baseline agree to within numerical precision.	33
Table 4.1	Summary of datasets used in the VLouvain experiments.	50
Table 4.2	Modularity score (Q) comparison on 10,000-node subsamples where standard Louvain is still runnable. The identical scores confirm that VLouvain is an exact reformulation.	51
Table 4.3	Performance comparison on the Amazon Products dataset. ‘X’ indicates failure (out of memory or timeout). Bold entries mark the fastest method at each threshold.	51
Table 4.4	Performance comparison on the Taobao dataset. ‘X’ indicates failure.	52
Table 4.5	Exact and approximate Top- K performance on Amazon. Construction time, Louvain time, edge counts, community counts, and NMI relative to the VLouvain full-graph partition.	53
Table 4.6	Sample of semantic communities automatically detected by VLouvain from 100,000 unlabelled FastText embeddings. Community IDs correspond to algorithmic supernode assignments.	55

Table 5.1	Asymptotic Complexity of GraphRAG-V Pipeline Components .	66
Table 5.2	Indexing/setup time on the MultiHopRAG corpus. GraphRAG-V achieves 34× faster indexing than Microsoft GraphRAG. . . .	69
Table 5.3	Wall-clock time to answer all 2,556 questions. $\diamond$ = extrapolated from partial run. GraphRAG-V achieves over 140 hours of savings compared to Microsoft GraphRAG.	69
Table 5.4	Accuracy / Recall (%) for the full 2,556-question corpus across question types. GraphRAG-V consistently outperforms baselines, with largest gains on Comparison (+12.0 pp) and Temporal (+24.5 pp) queries.	70
Table 5.5	Recall (%) by Question Type (subset of data from Table 5.4 for emphasis). GraphRAG-V significantly outperforms the baseline on complex Comparison and Temporal queries.	70
Table 5.6	Performance on 125-question stratified subset. This enables direct comparison with MS GraphRAG, which could not complete the full corpus due to prohibitive runtime ($\approx$ 144 hours estimated).	71
Table 5.7	Performance on full 2,556-question MultiHopRAG corpus. MS GraphRAG excluded as it could not complete within reasonable time.	72
Table B.1	Per-run results for the determinism check (5 runs, 10,000 nodes, M4 Mac MPS, $\gamma = 1$).	96

List of Figures

Figure 1.1 The Memory Wall. Explicitly storing edges for dense semantic graphs requires terabytes of RAM, whereas the embedding matrix fits comfortably in GPU memory.	2
Figure 2.1 Basic Graph Primitives. Nodes represent entities, and weighted edges represent the strength of relationships.	7
Figure 2.2 Community Detection. The goal is to identify densely connected subgraphs (clusters) that have sparse connections to the rest of the network.	9
Figure 2.3 The Hierarchical Louvain Method. Nodes are recursively aggregated into supernodes to optimize modularity.	12
Figure 2.4 Adjacency Matrix Density. Sparse graphs (left) are efficient to store explicitly. Dense semantic graphs (right) are fully populated, causing memory explosion.	16
Figure 2.5 The Associativity Advantage. By reordering the matrix multiplication parenthesis, we bypass the construction of the adjacency matrix entirely.	18
Figure 3.1 The associativity reformulation. By decomposing the adjacency matrix and reordering operations, we bypass the $O(N^2)$ bottleneck entirely.	23
Figure 5.1 The Dual-Path Architecture. By combining local precision with global structure, GraphRAG-V answers multi-hop questions without explicit knowledge graphs.	64

List of Mathematical Symbols

Symbol	Definition
N	Number of nodes in the graph
d	Dimension of the embedding vector (e.g., 768)
$\mathbf{V}$	Embedding Matrix ($N \times d$)
$\mathbf{A}$	Adjacency Matrix ($N \times N$)
$\mathbf{D}$	Degree Matrix ($N \times N$)
Q	Modularity score
$\mathbf{Y}$	Label Matrix ($N \times c$)
ρ	Graph Density ($ E /N(N-1)$)

ACKNOWLEDGEMENTS

I would like to extend my deepest gratitude to:

Dr. Thomo, Thank you for your guidance and support throughout my Master’s and Doctoral studies. Your teaching on the fundamentals of machine learning and evaluation methods laid the foundation for my research, and your passion for graph theory has been a constant source of inspiration. I am deeply grateful for your patience in reviewing my papers and for giving me the opportunity to present at conferences, where I could learn from and connect with researchers around the world.

Dr. Srinivasan, I am sincerely grateful for your mentorship over the years. Your course on randomized algorithms transformed how I think about problem-solving, opening my mind beyond deterministic approaches. Thank you for your thoughtful feedback on my papers and for encouraging me to engage with the broader research community. Your guidance has profoundly shaped my growth as a researcher.

My family, Thank you for your unwavering support throughout this journey. Your patience during the long hours when I was absorbed in my work, and your understanding when research took me away, have meant everything. Even though my field is far from your own, you always listened with genuine interest as I shared my excitement about new findings, patiently trying to understand the terminology when I couldn’t help but talk about my latest papers. Your encouragement lifted me during the most challenging moments, and your pride in my achievements—from conference presentations to paper publications—has been a constant source of motivation. Thank you for celebrating every milestone with me and for always believing in me.

DEDICATION

To my family

Chapter 1

Introduction

1.1 The Graph Paradox

Graph theory has always been built on the concept of the *edge*: a stored connection between two nodes. For the sparse networks of the 20th century (road maps, power grids, citation networks), this was natural. The number of edges was small, adjacency lists were compact, and algorithms could traverse the structure in linear time.

Modern representation learning has upended this picture. Neural networks now routinely map entities (users, products, paragraphs of text) to dense numerical vectors called *embeddings*, and the similarity between any two entities reduces to an inner product of their vectors. The resulting graph is no longer sparse: every node is connected to every other node with some non-zero weight, and the number of edges grows as N^2 . This creates a paradox at the heart of modern data science: **the richer our representations become, the harder it is to apply the graph algorithms that should benefit from them.**

Consider the scale of the problem. For a corpus of $N = 1$ million embedding vectors, the fully materialized similarity matrix contains on the order of 10^{12} entries. Even with 32-bit weights, storing it requires several terabytes of RAM, far exceeding the capacity of any single GPU. In practice the failure arrives much earlier: as we demonstrate in Chapters 3 and 4, standard industrial libraries (cuGraph, iGraph, NetworkKit) exhaust memory on dense semantic graphs as small as 15,000 nodes on a 40 GB GPU, because they attempt to materialize hundreds of millions of non-zero edges. Figure 1.1 illustrates the contrast: the explicit adjacency requires terabytes, while the embedding matrix that encodes the same information fits in gigabytes.

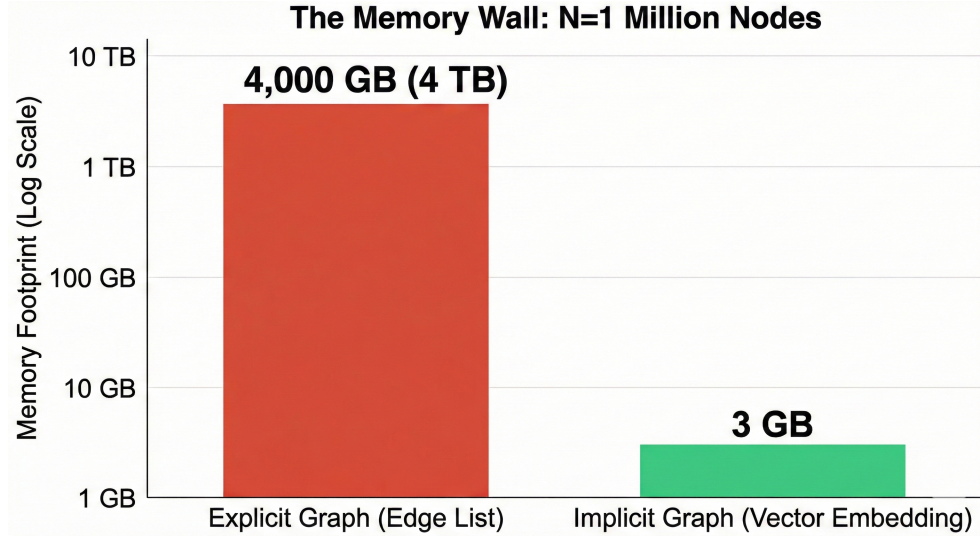


Figure 1.1: The Memory Wall. Explicitly storing edges for dense semantic graphs requires terabytes of RAM, whereas the embedding matrix fits comfortably in GPU memory.

Faced with this *memory wall*, practitioners are forced into a compromise: graph sparsification. Techniques such as k -nearest-neighbour pruning or thresholding discard all but the strongest connections to fit the data into memory [49]. But this cure is worse than the disease. Pruning severs the weak ties that carry global information flow and multi-hop reasoning paths, sacrificing precisely the long-range structure that dense embeddings were designed to capture.

This dissertation argues that the compromise is unnecessary. The bottleneck lies not in the data but in how we choose to represent it. The same $N = 1$ million entities that require terabytes as an explicit similarity matrix occupy roughly 3 GB when stored as $d = 768$ -dimensional embeddings, comfortably fitting on a single commodity GPU. The key insight is that the edges are already encoded in these embeddings: the similarity between any two nodes can be computed on demand from their vectors, without ever materializing the full N^2 matrix.

1.2 The Key Insight: Low-Rank Dense Graphs

The observation above can be made precise. Let $\mathbf{V} \in \mathbb{R}^{N \times d}$ collect the embedding vectors as rows. The induced similarity matrix is $\mathbf{A} = \mathbf{V}\mathbf{V}^\top \in \mathbb{R}^{N \times N}$, with $A_{ij} = \mathbf{v}_i \cdot \mathbf{v}_j$. Although $\mathbf{A}$ has N^2 entries, it factors through a d -dimensional space: $\text{rank}(\mathbf{A}) =$

$\text{rank}(\mathbf{V}) \leq d$, and for generic neural-network embeddings this bound is tight. The entire similarity structure is therefore fully determined by the Nd numbers in $\mathbf{V}$; its degrees of freedom scale linearly in N , not quadratically.

We call such graphs *low-rank dense graphs*: dense, because almost every pair of nodes has a non-zero similarity, and low-rank, because the similarity structure lies in a d -dimensional column space with $d \ll N$. For this class of graphs, materializing the adjacency matrix is an unnecessary artifact: any graph algorithm that operates on $\mathbf{A}$ through linear algebra can be reformulated to operate directly on $\mathbf{V}$, reducing memory from $O(N^2)$ to $O(Nd)$ without approximation error.

We term this approach **Implicit Graph Theory**. By keeping the graph in its factored form $\mathbf{V}$ rather than expanding it into $\mathbf{A}$, we can perform exact, global graph operations (diffusion, community detection, retrieval) on datasets of millions of nodes where explicit methods fail entirely.

1.3 Contributions

This dissertation demonstrates the implicit approach through three contributions:

1. Vector-Based Label Propagation (VLP), Chapter 3.

T. Yu, V. Srinivasan, A. Thomo. *Efficient Vector-Based Label Propagation for Massive Low-Rank Graphs*. WSDM 2026, pp. 1278–1282 [75].

We reformulate label propagation to operate directly on the embedding matrix through associative matrix-vector products, proving mathematical equivalence to the explicit algorithm. VLP scales to 1.5 million nodes on a single GPU in a setting where standard dense-graph implementations exhaust memory beyond roughly 15,000 nodes. We also prove convergence of label propagation in the continuous low-rank setting, addressing a gap in the stability analysis of these methods.

2. VLouvain, Chapter 4.

T. Yu, V. Srinivasan, A. Thomo. *Efficient Vector-Based Louvain Algorithm for Massive Low-Rank Graphs*. EDBT 2026, pp. 537–543 [76].

We present a community detection algorithm that optimizes modularity directly in the embedding space. By deriving a Vector Supernode formulation, we show that hierarchical community aggregation reduces to vector addition, eliminating

the need to materialize edges. V Louvain produces the same partition sequence as standard Louvain (under identical tie-breaking), and we prove this formally through three lemmas covering degree, modularity-gain, and aggregation equivalence. It scales to million-node dense graphs in minutes on a single GPU, a regime where cuGraph and NetworKit crash due to memory exhaustion.

3. GraphRAG-V, Chapter 5.

T. Yu, V. Srinivasan, A. Thomo. *GraphRAG-V: Fast Multi-Hop Retrieval via Text-Chunk Communities*. ASONAM 2025, pp. 3–14 [77].

We apply implicit graph theory to Retrieval-Augmented Generation. GraphRAG-V uses V Louvain to organize text-chunk embeddings into semantic communities, replacing the expensive LLM-based entity extraction pipeline of prior graph-RAG systems [16]. On the MultiHopRAG benchmark [69], it reduces indexing time from 3.0 hours to 5.3 minutes (34×) while outperforming both a vector-store baseline and Microsoft’s GraphRAG in answer recall.

All three algorithms are released as open-source Python libraries compatible with the PyTorch ecosystem:

- VLP: <https://github.com/yutengkai/vector-based-label-propagation>
- V Louvain: <https://github.com/yutengkai/VLouvain>
- GraphRAG-V: <https://github.com/yutengkai/GraphRAG-V>

1.4 Outline

Chapter 2 establishes the fundamental concepts required to understand implicit graph theory. It reviews classical graph mining, community detection, label propagation, and the evolution of vector embeddings, and defines the mathematical equivalence between explicit edge lists and implicit vector operations, setting the notation used throughout the thesis.

Chapter 3 introduces Vector-Based Label Propagation (VLP). We demonstrate how label propagation can be reformulated as a sequence of dense matrix operations, allowing for exact global diffusion on massive graphs without creating edges. We also provide a theoretical convergence proof for label propagation in the continuous low-rank setting.

Chapter 4 addresses the challenge of structure discovery. It presents VLouvain, an algorithm that detects hierarchical communities directly in the embedding space. We derive the Vector Supernode formulation, proving that modularity maximization can be performed on vector aggregates with $O(Nd)$ memory.

Chapter 5 applies implicit graph theory to Retrieval-Augmented Generation (RAG). We introduce GraphRAG-V, a system that uses VLouvain to organize text-chunk embeddings into semantic communities, replacing the LLM-based entity extraction pipeline of prior graph-RAG systems.

Chapter 6 outlines future directions, including extensions to directed graphs, cross-modal embeddings, differentiable community detection, and sparse attention mechanisms.

Chapter 7 concludes the dissertation by summarizing our contributions, discussing limitations, and reflecting on the shift from memory-bound explicit structures to compute-bound implicit vector arithmetic.

Chapter 2

Background and Related Work

To establish a rigorous foundation for the implicit algorithms proposed in this dissertation, we first define the standard primitives of explicit graph theory. We adhere to standard notation: integer counts are written in uppercase italic (e.g., N for the number of nodes), other scalars in lowercase italic (e.g., γ , m), vectors in bold lowercase (e.g., $\mathbf{v}$), and matrices in bold uppercase (e.g., $\mathbf{A}$). We use V (italic) to denote the vertex set of a graph and $\mathbf{V}$ (bold) to denote the embedding matrix introduced later in the chapter; context will make clear which is meant.

2.1 Graph Theory Primitives

A graph is formally defined as a tuple $G = (V, E, \mathbf{W})$, where $V = \{v_1, v_2, \dots, v_N\}$ is a set of N vertices (or nodes), and $E \subseteq V \times V$ is a set of edges connecting them. In the general case of weighted graphs, which are the primary focus of this work, $\mathbf{W}$ represents a weight function $w : E \rightarrow \mathbb{R}^+$ assigning a non-negative strength to each connection. While signed graphs with negative weights exist in specialized literature, this dissertation works exclusively under the assumption of non-negativity. Non-negative weights are required both for the modularity formulation used by V Louvain in Chapter 4 and for the convergence proof of label propagation in Chapter 3.

The structural information of G is encoded in the **Adjacency Matrix** $\mathbf{A} \in \mathbb{R}^{N \times N}$. For an unweighted graph, the entries A_{ij} are binary, indicating the presence or absence of an edge. For weighted graphs, $A_{ij} = w_{ij}$ represents the strength of the relationship between node i and node j . We assume the graph is undirected, implying $\mathbf{A}$ is symmetric ($A_{ij} = A_{ji}$), and typically contains no self-loops ($A_{ii} = 0$). The Gram-

matrix construction introduced in Section 2.6 naturally produces non-zero diagonal entries; our implicit algorithms enforce $A_{ii} = 0$ by subtracting the diagonal explicitly when needed.

A fundamental derived quantity is the **Degree Matrix $\mathbf{D}$** , a diagonal matrix where each entry $D_{ii} = k_i$ represents the weighted degree of node i :

$$k_i = \sum_{j=1}^N A_{ij} \quad (2.1)$$

The degree k_i captures the total connectivity of a node within the system. Figure 2.1 shows a small example of these primitives.

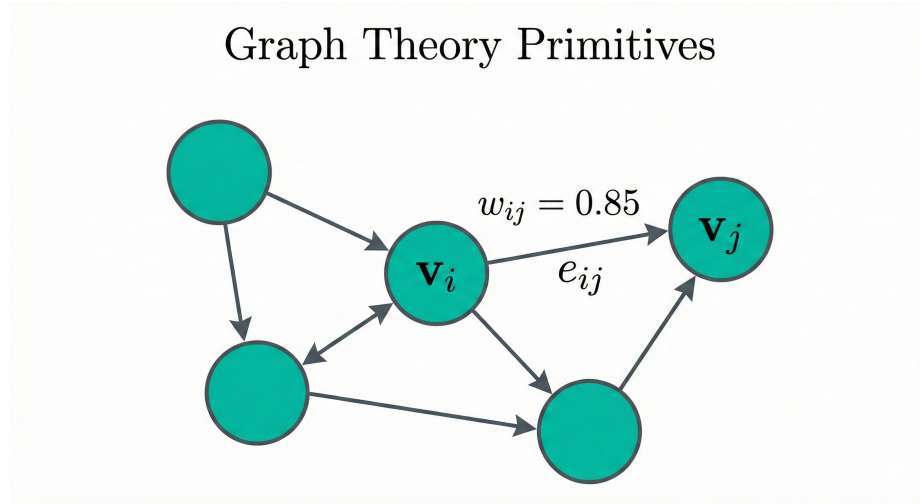


Figure 2.1: Basic Graph Primitives. Nodes represent entities, and weighted edges represent the strength of relationships.

2.1.1 The Graph Laplacian

Beyond storing topology, the adjacency matrix gives rise to the **Graph Laplacian**, which governs how quantities such as labels or signals flow across edges. The unnormalized Laplacian is $\mathbf{L} = \mathbf{D} - \mathbf{A}$. In practice, high-degree nodes naturally accumulate more edge weight, so it is useful to divide each entry of $\mathbf{A}$ by the degrees of its endpoints. This gives the symmetric normalized Laplacian,

$$\mathcal{L} = \mathbf{I} - \mathbf{D}^{-1/2} \mathbf{A} \mathbf{D}^{-1/2}, \quad (2.2)$$

which has eigenvalues $0 = \lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_N$ and orthogonal eigenvectors that encode the graph’s global connectivity [5]. The Laplacian is the foundation of spectral clustering, where Cheeger’s inequality links the second eigenvalue λ_2 to the sparsest cut of the graph [9].

A closely related object is the **row-normalized transition matrix** $\mathbf{D}^{-1}\mathbf{A}$, which drives continuous label propagation (Section 2.3.2). Each entry $(\mathbf{D}^{-1}\mathbf{A})_{ij} = A_{ij}/k_i$ is the probability that a random walker at node i steps to node j ; multiplying this matrix by a label vector averages each node’s label over its neighbours, which is exactly the diffusion step. The two matrices are mathematically linked. Let μ_i denote the eigenvalues of $\mathbf{D}^{-1}\mathbf{A}$. Then $\mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$ has the same eigenvalues μ_i (the two matrices are similar via the transform $\mathbf{D}^{1/2}$), and the Laplacian eigenvalues are simply $\lambda_i = 1 - \mu_i$.

In other words, spectral clustering and label propagation are two sides of the same coin: they look at the same graph structure through different lenses (eigenvectors vs. iterative diffusion). This dissertation uses the diffusion lens, not the spectral one, but they are mathematical cousins, not strangers.

2.2 Community Detection

With the graph primitives in hand, we now review the two problems this dissertation addresses, starting with community detection. Community detection is the process of identifying dense subgraphs where nodes are more connected to each other than to the rest of the network, and it is a cornerstone of graph mining [22]. Among the many approaches, ranging from random walks [60] to statistical inference, modularity maximization remains the most widely adopted framework due to its interpretability and effectiveness on real-world networks. Figure 2.2 illustrates the idea: communities are groups of nodes with dense internal connections and sparse connections to the rest of the graph.

2.2.1 Modularity Maximization

The concept of modularity, introduced by Newman and Girvan [53], quantifies the quality of a partition by comparing the density of edges within communities to a random null model. Formally, for a weighted graph with adjacency matrix $\mathbf{A}$ and

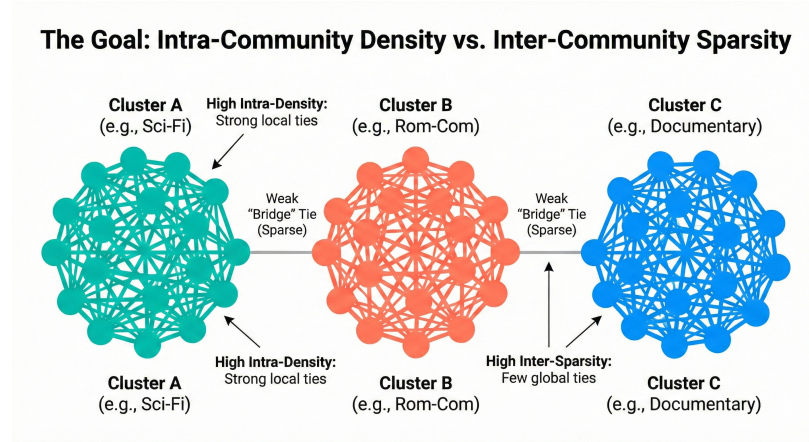


Figure 2.2: Community Detection. The goal is to identify densely connected sub-graphs (clusters) that have sparse connections to the rest of the network.

total edge weight $m = \frac{1}{2} \sum_{ij} A_{ij}$, the modularity Q is defined as:

$$Q = \frac{1}{2m} \sum_{ij} \left(A_{ij} - \gamma \frac{k_i k_j}{2m} \right) \delta(c_i, c_j) \quad (2.3)$$

Here, k_i is the degree of node i , c_i is the community assignment of node i , and $\delta(c_i, c_j)$ is the Kronecker delta function, which is 1 if i and j are in the same community and 0 otherwise. The term $\gamma \frac{k_i k_j}{2m}$ represents the expected weight of the edge between i and j in a random graph with the same degree distribution (the configuration model), scaled by the resolution parameter γ . Maximizing Q is NP-hard; thus, heuristic methods are required for large-scale graphs [8].

2.2.2 The Resolution Limit

Despite its popularity, modularity is not without flaws. Fortunato and Barthélemy demonstrated that modularity suffers from a **Resolution Limit** [23]. In very large graphs, the null model expectation of edges between two groups becomes very small. Consequently, the modularity objective may merge small, well-defined communities into larger, heterogeneous ones simply because any connection between them exceeds the near-zero expectation.

To counteract this, the resolution parameter γ was introduced (as shown in the equation above). Tuning $\gamma > 1$ biases the algorithm towards discovering smaller communities, while $\gamma < 1$ favors larger macro-structures. In Chapter 4, our implicit

formulation must seamlessly incorporate this γ parameter without requiring explicit degree lookups.

2.2.3 The Louvain Algorithm

To contextualize the contributions of this work, it is necessary to formally define the existing paradigms for modularity optimization. The Louvain method [3] is the de facto standard heuristic for this task. It operates via a two-phase iterative process of local moving and community aggregation.

Phase 1: Local Modularity Optimization. Initially, each node is assigned to its own community. The algorithm iterates through every node i and considers moving it into the community of each neighbour j . The move is executed if it yields a positive gain in modularity ($\Delta Q > 0$). The modularity gain ΔQ for moving node i from its current community C to a neighbouring community D is calculated as:

$$\Delta Q = \left[\frac{\sum_{j \in D} A_{ij}}{2m} - \gamma \frac{k_i \sum_{j \in D} k_j}{(2m)^2} \right] - \left[\frac{\sum_{j \in C} A_{ij}}{2m} - \gamma \frac{k_i \sum_{j \in C} k_j}{(2m)^2} \right] \quad (2.4)$$

This formula compares the benefit of placing node i in community D against keeping it in community C , accounting for both observed edges and the null model expectation. We give the formula in its general form, which applies whether the source community C is a singleton or already contains multiple nodes. In Chapter 4, VLouvain computes an equivalent quantity using community-level vector aggregates rather than explicit edge sums; the algebraic equivalence is the subject of the exactness proof in that chapter. This greedy process repeats until no further local moves can improve Q .

Phase 2: Community Aggregation. Once local maxima are reached, the algorithm constructs a new, coarser graph. Each community found in Phase 1 becomes a single “supernode” in the new graph. The weights of the edges between these supernodes are calculated as the sum of the weights of the edges between their constituent nodes in the original graph:

$$W_{C_u, C_v} = \sum_{i \in C_u, j \in C_v} A_{ij} \quad (2.5)$$

Self-loops on supernodes (W_{C_u, C_u}) represent the internal density of the community.

Algorithm 1 The Louvain Method (Explicit Adjacency)

Require: Graph $G = (V, E, \mathbf{W})$
Ensure: Hierarchical community partition

```

1: Initialize each node in its own community:  $c_i \leftarrow i, \forall i \in V$ 
2: repeat
3:   // Phase 1: Local Modularity Optimization
4:   repeat
5:     improvement  $\leftarrow$  false
6:     for each node  $i \in V$  do
7:        $\Delta Q_{\max} \leftarrow 0, c_{\text{best}} \leftarrow c_i$ 
8:       for each neighbour  $j$  of  $i$  do
9:         Calculate  $\Delta Q$  of moving  $i$  to  $c_j$ 
10:        if  $\Delta Q > \Delta Q_{\max}$  then
11:           $\Delta Q_{\max} \leftarrow \Delta Q, c_{\text{best}} \leftarrow c_j$ 
12:        end if
13:      end for
14:      if  $\Delta Q_{\max} > 0$  then
15:         $c_i \leftarrow c_{\text{best}}$ 
16:        improvement  $\leftarrow$  true
17:      end if
18:    end for
19:  until not improvement
20:  // Phase 2: Community Aggregation
21:  Construct new graph  $G'$  where nodes are communities from Phase 1
22:   $W_{C_u, C_v} \leftarrow \sum_{i \in C_u, j \in C_v} A_{ij}$  ▷ Bottleneck: Requires  $O(|E|)$  traversal
23:   $G \leftarrow G'$ 
24: until no further aggregation is possible

```

The algorithm repeats Phase 1 and Phase 2 on this coarsened graph, as shown in Figure 2.3, until no further changes occur. However, the aggregation step explicitly sums edge weights, requiring iteration over an explicit edge list E . While effective for sparse networks, in the dense semantic graphs targeted by this dissertation, where $|E|$ is on the order of N^2 , this foundational step becomes the primary obstacle to scalability.

While Louvain is powerful, it can produce disconnected communities: a commu-

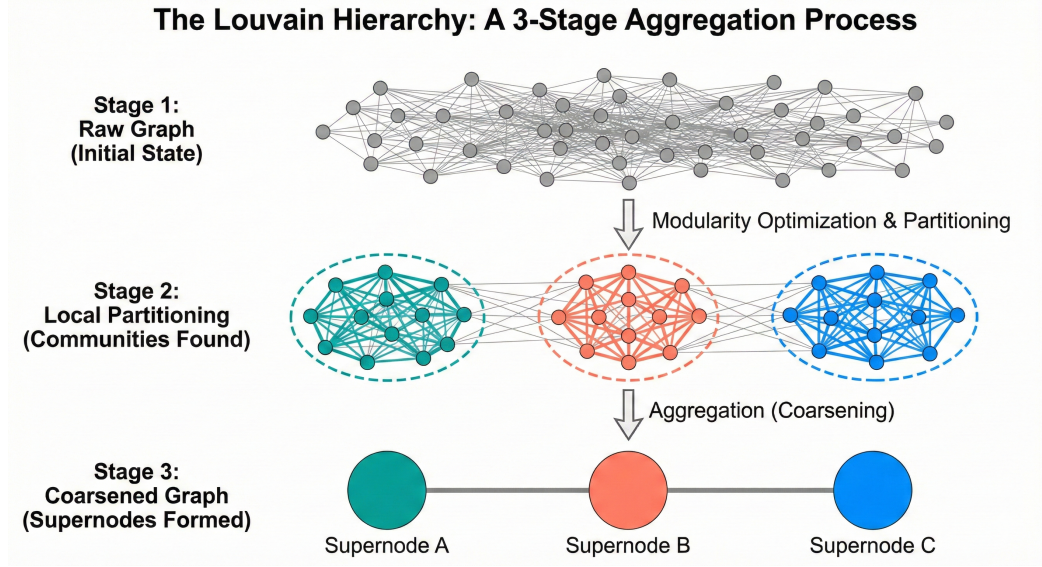


Figure 2.3: The Hierarchical Louvain Method. Nodes are recursively aggregated into supernodes to optimize modularity.

nity may contain nodes that have no path to each other through other members of the same community. Traag et al. introduced the Leiden algorithm [71] to address this. Leiden adds a refinement step: after the local moves converge, each community is checked for internal connectivity, and disconnected components within a community are split into separate sub-communities before aggregation. This guarantees well-connected partitions but does not change the fundamental edge-access pattern: Leiden, like Louvain, requires an explicit edge list, making it equally susceptible to the memory wall on dense graphs. This dissertation focuses on Louvain as the base algorithm; extending the implicit formulation to include Leiden’s refinement step is straightforward, since the refinement operates within already-formed communities.

2.3 Label Propagation

The term “Label Propagation” is used in the literature to describe two fundamentally different algorithms: a discrete voting heuristic for community detection, and a continuous diffusion process for semi-supervised learning. Since this dissertation builds on the latter, we disambiguate the two here.

2.3.1 Discrete “Voting” Label Propagation

Introduced primarily for community detection by Raghavan et al. [57], discrete LP is a heuristic where each node iteratively updates its label to match the most frequent label (the mode) among its immediate neighbours. Ties are broken randomly.

While highly efficient ($O(|E|)$ per iteration), this voting mechanism (Algorithm 2) is notoriously unstable. It lacks a formal objective function and is prone to infinite oscillations, particularly in bipartite or highly symmetric graph structures, where a set of nodes may continuously swap labels back and forth without ever converging.

Algorithm 2 Discrete “Voting” Label Propagation

Require: Graph $G = (V, E)$, Initial labels y_i for $i \in V$

Ensure: Final discrete labels for all nodes

```

1: repeat
2:   changed  $\leftarrow$  false
3:   for each node  $i \in V$  in a random order do
4:      $L \leftarrow$  multiset of labels from neighbours of  $i$ 
5:      $y_{\text{new}} \leftarrow$  most frequent label in  $L$  (break ties randomly)
6:     if  $y_i \neq y_{\text{new}}$  then
7:        $y_i \leftarrow y_{\text{new}}$ 
8:       changed  $\leftarrow$  true
9:     end if
10:  end for
11: until not changed or maximum iterations reached

```

2.3.2 Continuous “Linear” Label Propagation

In contrast, the continuous formulation of LP, introduced by Zhu and Ghahramani [82] for semi-supervised learning, treats labels as probability distributions that diffuse across the graph’s edges, analogous to heat diffusion on a manifold. This continuous approach is mathematically rigorous and serves as the direct foundation for the Vector-Based Label Propagation (VLP) algorithm developed in Chapter 3.

Let $\mathbf{Y} \in \mathbb{R}^{N \times c}$ be a label matrix where Y_{ij} represents the probability that node i belongs to class j . The initial matrix $\mathbf{Y}^{(0)}$ contains ground-truth labels (typically as one-hot vectors) for the labelled subset and zeros (or uniform distributions) for the unlabelled nodes. The core intuition is the *smoothness assumption*: nodes connected

by strong edges should possess similar label distributions.

This is operationalized by an iterative linear diffusion process. At each iteration, the diffusion step

$$\mathbf{Y}^{(t+1)} \leftarrow \mathbf{D}^{-1} \mathbf{A} \mathbf{Y}^{(t)} \quad (2.6)$$

is applied, where $\mathbf{D}^{-1} \mathbf{A}$ is the row-normalized transition matrix of a random walk on the graph. The process iterates until convergence.

Algorithm 3 Continuous “Linear” Label Propagation [82]

Require: Adjacency $\mathbf{A}$, Degree matrix $\mathbf{D}$, Initial label matrix $\mathbf{Y}^{(0)}$

Ensure: Converged label probability matrix $\mathbf{Y}^{(\infty)}$

- 1: $t \leftarrow 0$
 - 2: $\mathbf{P} \leftarrow \mathbf{D}^{-1} \mathbf{A}$ ▷ Row-normalized transition matrix
 - 3: **repeat**
 - 4: $\mathbf{Y}^{(t+1)} \leftarrow \mathbf{P} \mathbf{Y}^{(t)}$ ▷ Diffusion step. Bottleneck: requires $O(|E|)$ traversal
 - 5: $t \leftarrow t + 1$
 - 6: **until** $\|\mathbf{Y}^{(t)} - \mathbf{Y}^{(t-1)}\| < \epsilon$
-

2.3.3 The Density Penalty

Unlike the discrete voting method, continuous LP is a linear iteration governed by the row-normalized transition matrix $\mathbf{D}^{-1} \mathbf{A}$, which makes its convergence behaviour amenable to formal analysis. We give a precise convergence statement for the low-rank setting in Chapter 3. However, standard implementations of continuous LP suffer from a critical limitation known as the **Density Penalty**.

As shown in line 5 of Algorithm 3, the matrix-vector multiplication requires traversing every non-zero entry in $\mathbf{A}$. For a dense similarity graph where $|E|$ is on the order of N^2 , a single iteration costs $O(N^2)$, which is prohibitively expensive for large-scale datasets. The standard practical workaround is to sparsify the graph via k -nearest-neighbour pruning, retaining only the strongest connections per node; this reduces cost but discards the long-range similarities that the dense graph encodes. In Chapter 3, we show how the explicit traversal can be completely bypassed using vector operations, preserving the full graph and the rigorous convergence properties of continuous LP. The density penalty affects community detection equally: the Louvain aggregation step (Algorithm 1, line 15) also requires $O(|E|)$ traversal. We now quantify what “dense” means precisely.

2.4 Graph Sparsity vs. Density

The computational cost of every algorithm described so far depends on how many edges the graph has. We measure this by the graph’s density, ρ , defined as the ratio of existing edges to possible edges. For an undirected graph with no self-loops, the maximum number of edges is $\binom{N}{2} = N(N-1)/2$, so:

$$\rho = \frac{|E|}{N(N-1)/2} \quad (2.7)$$

Sparse Graphs ($\rho \ll 1$): Historically, most graphs studied in network science, such as the World Wide Web, citation networks, or road maps, are sparse. In these structures, the number of edges scales linearly with the number of nodes ($|E| \approx \bar{k} \cdot N$, where $\bar{k}$ is the average degree). For sparse graphs, the adjacency matrix $\mathbf{A}$ is predominantly filled with zeros. Storing $\mathbf{A}$ using Compressed Sparse Row (CSR) formats requires only $O(|E|)$ memory, making explicit edge traversal highly efficient.

Dense Semantic Graphs ($\rho \approx 1$): In contrast, graphs derived from vector similarity (the focus of this dissertation) are inherently dense. If a graph is constructed by computing the similarity between every pair of node embeddings, then $|E| = \binom{N}{2}$, on the order of $N^2/2$. In this setting, the adjacency matrix is fully populated with non-zero values. As noted in the Introduction, explicitly instantiating $\mathbf{A}$ for a dense graph incurs a quadratic memory cost $O(N^2)$, rendering standard matrix operations intractable for large N . The contrast is visible in Figure 2.4: the sparse adjacency matrix is mostly empty, while the dense one is fully populated.

But where do these dense graphs come from? The answer is vector embeddings.

2.5 Vector Embeddings

The algorithms in this dissertation operate on vector embeddings rather than explicit edge lists. This section reviews how they are produced and how they are currently used for retrieval.

2.5.1 From Sparse to Dense Representations

Early vectorization methods such as TF-IDF [66] produced **sparse, high-dimensional** representations: most dimensions were zero, documents matched only on shared key-

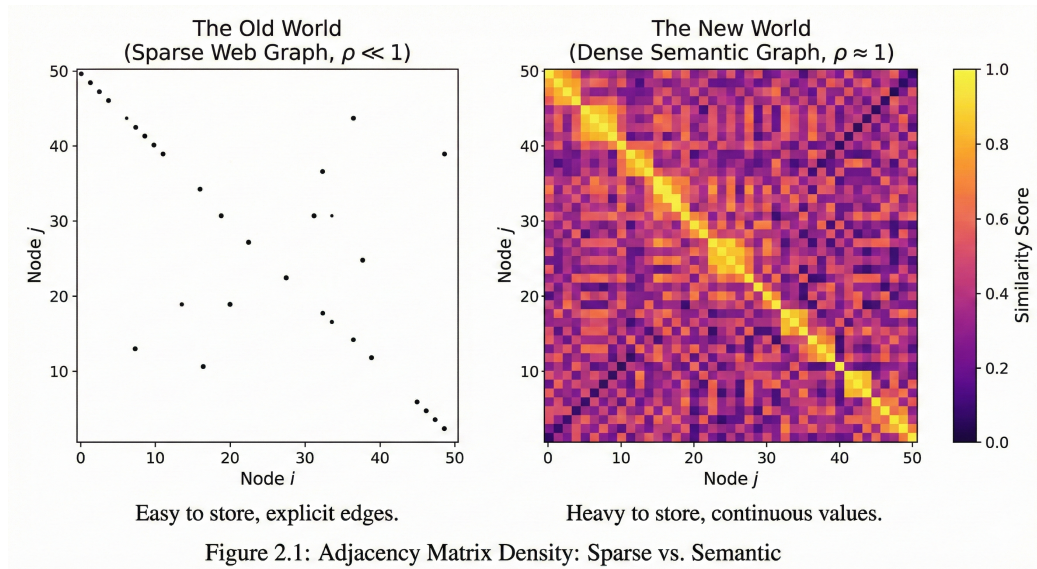


Figure 2.1: Adjacency Matrix Density: Sparse vs. Semantic

Figure 2.4: Adjacency Matrix Density. Sparse graphs (left) are efficient to store explicitly. Dense semantic graphs (right) are fully populated, causing memory explosion.

words, and the resulting similarity graph was naturally sparse. The shift toward **dense, low-rank embeddings** changed this picture fundamentally. Beginning with Word2Vec [52] and FastText [51] (the latter used in our VLouvain experiments in Chapter 4), and continuing with Transformer-based encoders such as Sentence-BERT [59] and BGE-M3 [7] (used in our GraphRAG-V experiments in Chapter 5), modern models map semantic meaning into dense vectors where every dimension is non-zero. The consequence is the full connectivity that breaks explicit graph algorithms: every pair of embeddings has a non-trivial similarity, and the resulting graph has $O(N^2)$ edges.

2.5.2 Retrieval-Augmented Generation (RAG) and Vector Search

Dense embeddings are not only the source of the scalability problem described above; they are also the foundation of modern information retrieval, which is the application domain of Chapter 5. Retrieval-Augmented Generation (RAG) [44] is the dominant paradigm. In RAG, a corpus is chunked and embedded. When a user queries a Large Language Model (LLM), the query is embedded, and a Vector Database retrieves the

top- k most similar chunks using Cosine Similarity:

$$\text{sim}(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \|\mathbf{v}\|} \quad (2.8)$$

To achieve millisecond latency over millions of vectors, these databases rely on Approximate Nearest Neighbor (ANN) indices, most notably **HNSW** (Hierarchical Navigable Small Worlds) [49]. While efficient, HNSW explicitly *prunes* connections, returning only the strongest local matches and discarding the global topology of the dataset.

2.6 Implicit Graph Theory

We conclude this chapter by formally defining the central paradigm of this dissertation. Explicitly storing the adjacency matrix $\mathbf{A}$ for a dense semantic graph is intractable. However, the semantic relationships are derived from low-rank embeddings. This duality allows us to redefine the graph as a linear algebraic operator.

2.6.1 The Bridge: Connecting Algebra to Geometry

Let $\mathbf{V} \in \mathbb{R}^{N \times d}$ be a matrix of node embeddings ($d \ll N$). We define the **Implicit Graph** associated with $\mathbf{V}$ as the weighted graph whose adjacency matrix is the Gram matrix:

$$\mathbf{A} = \mathbf{V}\mathbf{V}^\top. \quad (2.9)$$

Strictly speaking, $\mathbf{V}\mathbf{V}^\top$ has non-zero diagonal entries $\|\mathbf{v}_i\|^2$, whereas the standard convention adopted in Section 2.1 sets $A_{ii} = 0$. The algorithms in subsequent chapters reconcile this by explicitly subtracting the diagonal whenever the convention is required, at a cost of $O(Nd)$ rather than $O(N^2)$.

This formulation connects directly to the kernel methods literature in machine learning [63]. The Gram matrix $\mathbf{A}$ corresponds to a linear kernel $k(\mathbf{v}_i, \mathbf{v}_j) = \mathbf{v}_i^\top \mathbf{v}_j$ evaluated over all pairs of node embeddings. In kernel theory, the kernel trick allows algorithms to operate implicitly in high-dimensional feature spaces without ever constructing those spaces explicitly. Our implicit graph approach applies a related principle: rather than lifting data into higher dimensions, we exploit the existing low-rank factorization $\mathbf{A} = \mathbf{V}\mathbf{V}^\top$ to avoid materializing the dense $N \times N$ kernel matrix. This connection to kernel methods provides additional grounding for the algebraic

manipulations employed throughout this dissertation.

The core insight of Implicit Graph Theory is that any graph algorithm which can be expressed as a sequence of matrix-vector products $\mathbf{x}' \leftarrow \mathbf{A}\mathbf{x}$ can be reformulated using the associative property of matrix multiplication:

$$\mathbf{A}\mathbf{x} = (\mathbf{V}\mathbf{V}^\top)\mathbf{x} = \mathbf{V}(\mathbf{V}^\top\mathbf{x}) \quad (2.10)$$

By computing the right-hand side, we decompose the $O(N^2)$ graph operation into two $O(Nd)$ vector operations, as illustrated in Figure 2.5.

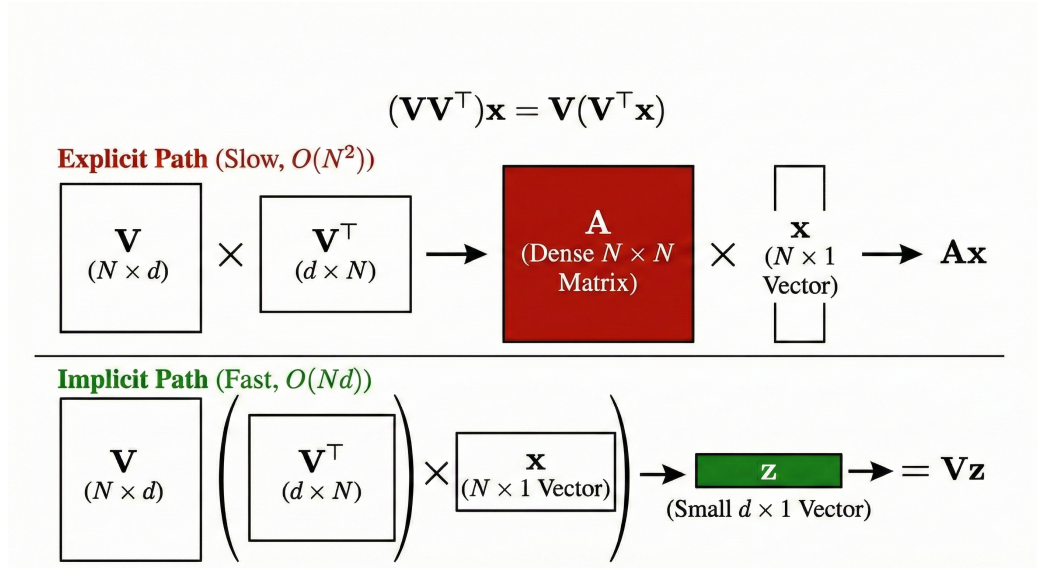


Figure 2.5: The Associativity Advantage. By reordering the matrix multiplication parenthesis, we bypass the construction of the adjacency matrix entirely.

2.6.2 The Gap: Exactness at Scale

The algorithms proposed in this dissertation, VLP (Chapter 3) and VLouvain (Chapter 4), achieve **exact implicit mining**: they yield results mathematically identical to running the algorithm on the full unpruned graph, while operating within the linear memory constraints of the vector space. This exactness distinguishes them from approximate methods such as graph pruning or neighbourhood sampling, which sacrifice completeness for computational tractability.

With these tools in hand, the next three chapters put implicit graph theory to work: Chapter 3 reformulates label propagation, Chapter 4 reformulates community

detection, and Chapter 5 applies the result to retrieval-augmented generation.

Chapter 3

Vector-Based Label Propagation

3.1 Introduction

Chapter 2 introduced continuous Label Propagation (LP) as a foundational method for semi-supervised learning on graphs, governed by the iterative diffusion rule $\mathbf{Y}^{(t+1)} \leftarrow \mathbf{D}^{-1}\mathbf{A}\mathbf{Y}^{(t)}$. There we observed that the per-iteration cost is dominated by the matrix-vector product against $\mathbf{A}$, which becomes the central obstacle when the underlying graph is dense. Sparse-network approaches such as k -nearest-neighbour pruning [21] reduce cost but discard precisely the long-range similarities that semantic embeddings are meant to capture.

This chapter takes the perspective developed in Chapter 2 (Section 2.6) and applies it to label propagation. The graphs we care about are not arbitrary: their adjacency is the Gram matrix $\mathbf{A} = \mathbf{V}\mathbf{V}^\top$ of an underlying $N \times d$ embedding $\mathbf{V}$, with $d \ll N$. This low-rank form arises in similarity graphs from text or image embeddings, latent-factor recommender systems, kernelized machine-learning graphs, and dense affinity graphs in biology and vision [47, 24, 13, 80]. The natural question, then, is: *can we exploit this low-rank structure to make label propagation scalable without sacrificing accuracy or global connectivity?*

We answer in the affirmative by introducing **VLP** (*Vector-Based Label Propagation*). Instead of enumerating edges, VLP performs each update directly in the embedding space using two dense matrix multiplications. This reduces the memory footprint from $\mathcal{O}(N^2)$ to $\mathcal{O}(Nd)$ and lowers the per-iteration cost from $\mathcal{O}(N^2c)$ to $\mathcal{O}(Ndc)$, where c is the number of label classes. Crucially, VLP is *mathematically equivalent* to classical LP: it produces the same propagation outcomes, exactly, with

no approximation and no loss of connectivity. We further establish a formal convergence guarantee that is unavailable in the general setting.

Two prior methods also exploit low-rank structure for label propagation, but in different ways. **BigLP** [55] scales via gradient-based approximations using Nyström methods, and **LowrankTLP** [45] addresses multi-relational graphs using tensor-product and low-rank constructions. Both rely on approximations. In contrast, VLP works directly with dense graphs of the form $\mathbf{V}\mathbf{V}^\top$ and performs *exact* propagation without constructing the quadratic adjacency.

Contributions of this chapter.

- We identify the low-rank setting as a practically important class of graphs and introduce **VLP**, a vector-based reformulation of classical LP that exploits this structure.
- VLP avoids the $\Theta(N^2)$ adjacency, reducing memory from $\mathcal{O}(N^2)$ to $\mathcal{O}(Nd)$ and per-iteration cost from $\mathcal{O}(N^2c)$ to $\mathcal{O}(Ndc)$. Each update consists of two dense matrix multiplications and is well suited to GPU execution.
- We prove that VLP is mathematically exact (Theorem 3.1) and establish a formal convergence guarantee under natural assumptions on the embeddings (Theorem 3.7).
- Experiments on large-scale graphs show that VLP scales to millions of nodes while exactly matching classical LP, reaching regimes that existing dense-graph implementations cannot.

3.2 Vector-Based Label Propagation (VLP)

3.2.1 Recap of the Classical Update

We recall the continuous LP update from Chapter 2 (Section 2.3.2, Algorithm 3). Given a weighted adjacency $\mathbf{A} \in \mathbb{R}^{N \times N}$ and a label matrix $\mathbf{Y}^{(t)} \in \mathbb{R}^{N \times c}$ holding one row per node, classical LP iterates the degree-normalized linear diffusion

$$\mathbf{Y}^{(t+1)} = \mathbf{D}^{-1} \mathbf{A} \mathbf{Y}^{(t)}, \quad (3.1)$$

where $\mathbf{D}$ is the diagonal degree matrix with $D_{ii} = \sum_j A_{ij}$. The bottleneck is the matrix-vector product $\mathbf{A}\mathbf{Y}^{(t)}$, which costs $\mathcal{O}(N^2c)$ per iteration and requires storing every entry of $\mathbf{A}$.

3.2.2 From Explicit Adjacency to Vector Factorization

In our setting, the adjacency is not arbitrary but is the Gram matrix $\mathbf{A} = \mathbf{V}\mathbf{V}^\top$ of the embeddings. Such factorizations cover a wide range of similarity measures expressible as inner products in a feature space, including normalized cosine similarity of embeddings and polynomial kernels with explicit feature maps. To be suitable for label propagation, these inner products must be non-negative (for example, polynomial kernels of even degree).

Cosine similarity as a special case. In our experiments, we adopt cosine-based similarity, a common choice in representation learning. Given unit-norm embeddings $\mathbf{v}_i$, the weight between nodes i and j is

$$A_{ij} = \frac{1}{2} \left(1 + \frac{\mathbf{v}_i^\top \mathbf{v}_j}{\|\mathbf{v}_i\| \|\mathbf{v}_j\|} \right), \quad (3.2)$$

which linearly rescales cosine similarity from $[-1, 1]$ to $[0, 1]$, ensuring non-negativity while retaining the full similarity signal. This can be written compactly by augmenting each vector as

$$\mathbf{v}'_i = \frac{[\mathbf{v}_i; 1]}{\sqrt{2}}, \quad (3.3)$$

so that $A_{ij} = \mathbf{v}'_i{}^\top \mathbf{v}'_j$. Indeed, for unit-norm $\mathbf{v}_i$ and $\mathbf{v}_j$,

$$\begin{aligned} \mathbf{v}'_i{}^\top \mathbf{v}'_j &= \frac{1}{2} ([\mathbf{v}_i; 1]^\top [\mathbf{v}_j; 1]) \\ &= \frac{1}{2} (\mathbf{v}_i^\top \mathbf{v}_j + 1) \\ &= \frac{1 + \cos(\theta_{ij})}{2}, \end{aligned} \quad (3.4)$$

where θ_{ij} is the angle between $\mathbf{v}_i$ and $\mathbf{v}_j$. The mapping sends perfectly aligned vectors ($\cos \theta = 1$) to weight 1, orthogonal vectors ($\cos \theta = 0$) to weight 0.5, and perfectly opposed vectors ($\cos \theta = -1$) to weight 0. Collecting the augmented rows gives $\mathbf{V} = [\mathbf{v}'_1; \dots; \mathbf{v}'_N] \in \mathbb{R}^{N \times (d+1)}$, and hence $\mathbf{A} = \mathbf{V}\mathbf{V}^\top$ with the diagonal entries set to zero. For ease of notation, in the remainder we drop the prime and simply write $\mathbf{v}_i$ for the augmented vectors.

Why factorization matters. Working directly with $\mathbf{V}$ avoids the explicit construction of the $N \times N$ adjacency, reducing space from $\mathcal{O}(N^2)$ to $\mathcal{O}(Nd)$, which is crucial since in practice $d \ll N$. This observation underpins our vector-based reformulation of LP, summarized visually in Figure 3.1.

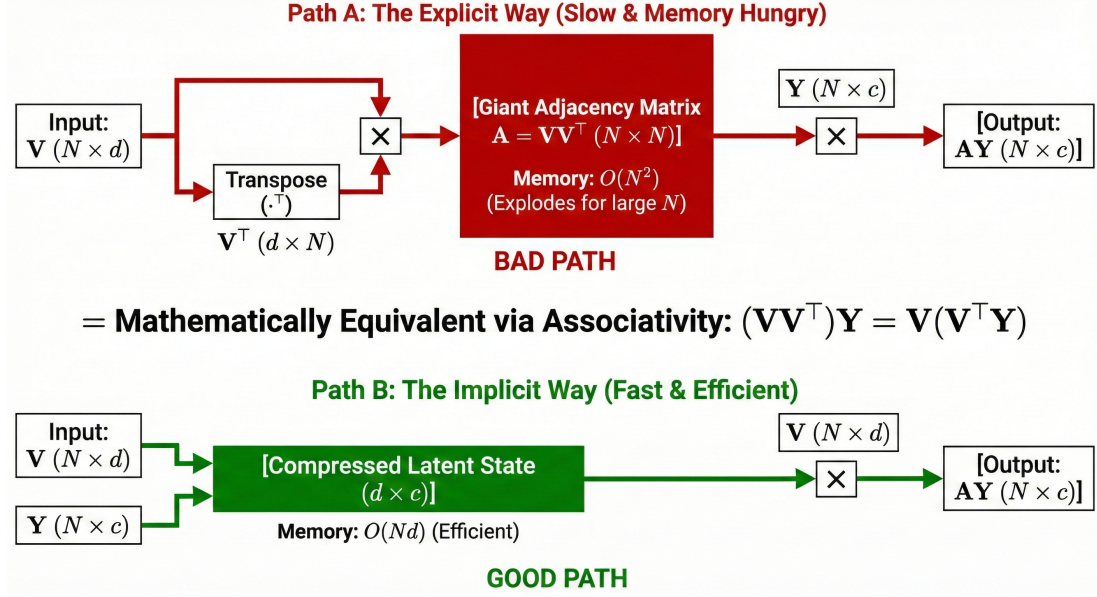


Figure 3.1: The associativity reformulation. By decomposing the adjacency matrix and reordering operations, we bypass the $\mathcal{O}(N^2)$ bottleneck entirely.

3.2.3 Vector-Based Update Rule

With $\mathbf{A}$ in low-rank form, label propagation can be carried out *without ever constructing* the $N \times N$ adjacency:

$$\mathbf{Y}^{(t+1)} = \text{inv_deg} \odot \left(\mathbf{V}(\mathbf{V}^T \mathbf{Y}^{(t)}) - \text{self_loop} \odot \mathbf{Y}^{(t)} \right), \quad (3.5)$$

where $\text{self_loop}_i = \mathbf{v}_i^T \mathbf{v}_i$ and $\odot$ is the Hadamard (elementwise) product, with broadcasting applied along the column dimension when needed. Intuitively, each iteration first *gathers* label information into a compact embedding summary via $\mathbf{V}^T \mathbf{Y}^{(t)}$, then *spreads* it back across all nodes through $\mathbf{V}$, removes self-contributions, and normalizes by degree.

Efficient self-loop and degree computation. Both self-loops and degrees can be obtained once in vectorized form. The self-loop weights are the diagonal entries of

$\mathbf{V}\mathbf{V}^\top$, which can be written as

$$\mathbf{s} = (\mathbf{V} \odot \mathbf{V}) \mathbf{1}_d, \quad (3.6)$$

where each $s_i = \mathbf{v}_i^\top \mathbf{v}_i$. The degree of node i is the sum of all similarities $\mathbf{v}_i^\top \mathbf{v}_j$ to other nodes, which equals $\mathbf{v}_i^\top \mathbf{V}_{\text{sum}}$ with $\mathbf{V}_{\text{sum}} = \sum_j \mathbf{v}_j$, minus its own self-loop s_i . In vector form, this gives

$$\mathbf{k} = \mathbf{V}(\mathbf{V}^\top \mathbf{1}_N) - \mathbf{s}. \quad (3.7)$$

Thus $\mathbf{k}$ collects the row sums of $\mathbf{V}\mathbf{V}^\top$ with the diagonal removed. Finally, the normalization factors are $\mathbf{inv_deg} = \mathbf{k}^{-1}$, applied elementwise.

Algorithm 4 Vector-Based Label Propagation (VLP)

Require: Node embedding matrix $\mathbf{V} \in \mathbb{R}^{N \times d}$ (after augmentation), initial label matrix $\mathbf{Y}^{(0)} \in \mathbb{R}^{N \times c}$, number of iterations T

Ensure: Final label distributions $\mathbf{Y}^{(T)}$

```

1:  $\mathbf{V}_{\text{sum}} \leftarrow \mathbf{V}^\top \mathbf{1}_N$  (size  $d$ )
2:  $\mathbf{s} \leftarrow (\mathbf{V} \odot \mathbf{V}) \mathbf{1}_d$  (self-loops, size  $N$ )
3:  $\mathbf{k} \leftarrow \mathbf{V} \mathbf{V}_{\text{sum}} - \mathbf{s}$  (degrees, size  $N$ )
4:  $\mathbf{inv\_deg} \leftarrow \mathbf{k}^{-1}$  (elementwise reciprocal)
5: for  $t = 0$  to  $T - 1$  do
6:    $\mathbf{Z} \leftarrow \mathbf{V}^\top \mathbf{Y}^{(t)}$  ( $d \times c$  GEMM)
7:    $\mathbf{W} \leftarrow \mathbf{V} \mathbf{Z}$  ( $N \times c$  GEMM)
8:    $\mathbf{W} \leftarrow \mathbf{W} - (\mathbf{s} \mathbf{1}_c^\top) \odot \mathbf{Y}^{(t)}$  (rowwise broadcast)
9:    $\mathbf{Y}^{(t+1)} \leftarrow (\mathbf{inv\_deg} \mathbf{1}_c^\top) \odot \mathbf{W}$  (rowwise normalize)
10: end for
11: return  $\mathbf{Y}^{(T)}$ 

```

Algorithm 4 summarizes the procedure. Each iteration involves two dense matrix-matrix multiplications ($\mathbf{V}^\top \mathbf{Y}^{(t)}$ and $\mathbf{V} \mathbf{Z}$), followed by vectorized subtraction of self-contributions and normalization. In this way, labels flow efficiently through the graph without constructing the full adjacency. The per-iteration complexity is $O(Ndc)$ rather than $O(N^2c)$ as in classical LP, a crucial saving when $N \gg d$. Because the heavy operations reduce to dense matrix multiplications (GEMMs), VLP is well suited to modern GPU architectures.

3.2.4 Exactness

A common misconception in scalable graph mining is that efficiency requires approximation. We emphasize that VLP is neither an approximation nor a heuristic: it is mathematically equivalent to classical LP. The following theorem makes this precise.

Theorem 3.1 (Exactness). *Let $\mathbf{Y}_{cls}^{(t)}$ denote the label matrix produced by the classical LP update $\mathbf{Y} \leftarrow \mathbf{D}^{-1}\mathbf{A}\mathbf{Y}$, with $\mathbf{A} = \mathbf{V}\mathbf{V}^\top - \text{diag}(\mathbf{V}\mathbf{V}^\top)$, and let $\mathbf{Y}_{vlp}^{(t)}$ denote the label matrix produced by Algorithm 4. If $\mathbf{Y}_{cls}^{(0)} = \mathbf{Y}_{vlp}^{(0)}$, then $\mathbf{Y}_{cls}^{(t)} = \mathbf{Y}_{vlp}^{(t)}$ for all $t \geq 0$, up to floating-point rounding.*

Proof. We show that one iteration of Algorithm 4 performs the same linear-algebra operation as one iteration of classical LP.

The classical update is

$$\mathbf{Y}^{(t+1)} = \mathbf{D}^{-1}\mathbf{A}\mathbf{Y}^{(t)} = \mathbf{D}^{-1}(\mathbf{V}\mathbf{V}^\top - \text{diag}(\mathbf{V}\mathbf{V}^\top))\mathbf{Y}^{(t)},$$

where the diagonal is removed to eliminate self-loops.

In Algorithm 4, the inner-loop computation is

$$\begin{aligned}\mathbf{W} &= \mathbf{V}(\mathbf{V}^\top \mathbf{Y}^{(t)}) - (\mathbf{s} \mathbf{1}_c^\top) \odot \mathbf{Y}^{(t)}, \\ \mathbf{Y}^{(t+1)} &= (\text{inv_deg} \mathbf{1}_c^\top) \odot \mathbf{W}.\end{aligned}$$

By the associativity of matrix multiplication, $\mathbf{V}(\mathbf{V}^\top \mathbf{Y}^{(t)}) = (\mathbf{V}\mathbf{V}^\top)\mathbf{Y}^{(t)}$. The term $(\mathbf{s} \mathbf{1}_c^\top) \odot \mathbf{Y}^{(t)}$ subtracts $s_i \mathbf{Y}_i^{(t)}$ from row i of $\mathbf{V}\mathbf{V}^\top \mathbf{Y}^{(t)}$, which is exactly the action of $\text{diag}(\mathbf{V}\mathbf{V}^\top)\mathbf{Y}^{(t)}$ since $s_i = \mathbf{v}_i^\top \mathbf{v}_i$ is the i -th diagonal entry of $\mathbf{V}\mathbf{V}^\top$. Finally, the elementwise multiplication $(\text{inv_deg} \mathbf{1}_c^\top) \odot \mathbf{W}$ scales row i by $1/k_i$, which is exactly left-multiplication by $\mathbf{D}^{-1} = \text{diag}(\text{inv_deg})$. Combining, $\mathbf{Y}_{vlp}^{(t+1)}$ equals $\mathbf{D}^{-1}(\mathbf{V}\mathbf{V}^\top - \text{diag}(\mathbf{V}\mathbf{V}^\top))\mathbf{Y}_{vlp}^{(t)} = \mathbf{Y}_{cls}^{(t+1)}$.

By induction on t , the two sequences agree for all iterations. Numerical differences arise only from the different order in which floating-point operations are accumulated.

□

□

3.2.5 Complexity Analysis

We compare the computational complexity of VLP against classical (dense) LP and against the common k NN-pruned alternative. Let N be the number of nodes, d the embedding dimension, c the number of classes, and T the number of iterations.

- **Classical LP (dense adjacency).** Storing $\mathbf{A}$ requires $\mathcal{O}(N^2)$ memory and each iteration costs $\mathcal{O}(N^2c)$, since the matrix-vector product $\mathbf{A}\mathbf{Y}^{(t)}$ touches every entry of $\mathbf{A}$.
- **k NN-pruned LP.** Restricting each node to its k nearest neighbours reduces memory to $\mathcal{O}(Nk)$ and per-iteration cost to $\mathcal{O}(Nkc)$, but introduces an indexing cost of $\mathcal{O}(Nd \log N)$ and, more importantly, discards the long-range similarities that the original graph encodes.
- **VLP (this chapter).** Storing $\mathbf{V}$ requires $\mathcal{O}(Nd)$ memory. Each iteration consists of two matrix products ($\mathbf{V}^\top \mathbf{Y}^{(t)}$ at $\mathcal{O}(Ndc)$ and $\mathbf{V}\mathbf{Z}$ at $\mathcal{O}(Ndc)$), plus $\mathcal{O}(Nc)$ work for the self-loop subtraction and normalization. The per-iteration cost is therefore $\mathcal{O}(Ndc)$.

Since $d \ll N$ in semantic graphs (e.g., $d = 768$ versus $N = 10^6$), VLP is orders of magnitude cheaper than dense classical LP, while remaining *exact* unlike k NN pruning.

3.3 Convergence Analysis

Label propagation methods have long faced convergence issues. Majority-vote schemes can oscillate or fail to stabilize in dense graphs [43, 25]. Heuristic refinements such as LabelRank [74] and semi-synchronous updates [10] attempt to enforce convergence through tie-breaking or scheduling, while theoretical studies [41] provide guarantees only in restricted settings. In summary, most existing formulations lack general convergence results and are limited to the case of discrete labels. Here we show that classical LP admits a formal convergence guarantee in the low-rank setting, even for continuous labels.

Recall that, in vector-based label propagation, the update is carried out via Equation (3.5). This yields an effective propagation matrix

$$\mathbf{P} = \text{inv_deg} \odot (\mathbf{V}\mathbf{V}^\top - \text{diag}(\mathbf{V}\mathbf{V}^\top)). \quad (3.8)$$

This is precisely the row-normalized transition matrix $\mathbf{D}^{-1}\mathbf{A}$ introduced in Chapter 2 (Section 2.3.2), written here in implicit factored form. Normalizing by degree ensures each row sums to one, making $\mathbf{P}$ row-stochastic.

Lemma 3.2 (Row-Stochasticity and Non-negativity). *The matrix $\mathbf{P}$ is non-negative and row-stochastic; that is, for every row i ,*

$$\sum_j P_{ij} = 1, \quad \text{and} \quad P_{ij} \geq 0 \text{ for all } i, j.$$

Proof. By construction, the entries of $\mathbf{V}\mathbf{V}^\top$ are the augmented cosine similarities $(1 + \cos \theta_{ij})/2$ derived in Section 3.2.2, which lie in $[0, 1]$. After removing the diagonal (zeros on the diagonal) and dividing each row by its row sum k_i , every entry of $\mathbf{P}$ is non-negative and each row sums to $\sum_j P_{ij} = (1/k_i) \sum_j (\mathbf{V}\mathbf{V}^\top - \text{diag}(\mathbf{V}\mathbf{V}^\top))_{ij} = k_i/k_i = 1$. $\square$ $\square$

Given a row-stochastic $N \times N$ matrix $\mathbf{P}$, construct a directed graph $G(\mathbf{P})$ with N nodes and an edge from node i to j whenever $P_{ij} > 0$.

Definition 3.3. *A matrix $\mathbf{P}$ is said to be **irreducible** if $G(\mathbf{P})$ is strongly connected.*

In our analysis, we assume that the embedding vectors are drawn from a continuous distribution (with a density) on the sphere. This is natural in practice: modern embedding methods (e.g., neural encoders, PCA, random projections) produce continuous-valued vectors, and even in discrete settings the probability of exact antipodality is negligible in high dimensions. Since the event of two vectors being exactly opposite has measure zero under a continuous distribution, with probability one all off-diagonal augmented similarities are strictly positive, making the digraph complete and strongly connected.

Lemma 3.4 (Irreducibility). *Under the continuous-distribution assumption, the matrix $\mathbf{P}$ is irreducible.*

Proof. With probability one, no pair of distinct embedding vectors is exactly antipodal, so $\cos \theta_{ij} > -1$ and the augmented similarity $(1 + \cos \theta_{ij})/2 > 0$ for all $i \neq j$. Therefore $P_{ij} > 0$ for all $i \neq j$, so $G(\mathbf{P})$ is a complete directed graph and is trivially strongly connected. Hence $\mathbf{P}$ is irreducible. $\square$ $\square$

Definition 3.5. *For each node i in $G(\mathbf{P})$, consider all possible cycle lengths (closed walks from i back to itself). Define the **period** of i , $p(i)$, as the greatest common divisor of those cycle lengths. If $p(i) = 1$ for all i , then $\mathbf{P}$ is called **aperiodic**.*

Lemma 3.6 (Aperiodicity). *The matrix $\mathbf{P}$ is aperiodic.*

Proof. Since $G(\mathbf{P})$ is a complete directed graph on N nodes (Lemma 3.4), it contains cycles of every length ≥ 2 . In particular, for any node i there exist a 2-cycle $i \rightarrow j \rightarrow i$ and a 3-cycle $i \rightarrow j \rightarrow k \rightarrow i$. Since $\gcd(2, 3) = 1$, the period $p(i) = 1$ for every i , and $\mathbf{P}$ is aperiodic. $\square$ $\square$

As $\mathbf{P}$ is finite, irreducible, and aperiodic, the Fundamental Theorem of Markov chains (Perron–Frobenius) guarantees that $\mathbf{P}$ has a unique stationary distribution π , and that $\mathbf{P}^t \rightarrow \mathbf{1}\pi^\top$ where $\mathbf{1}$ is the N -vector of ones. Hence, for any initial label column $\mathbf{y}$, $\mathbf{P}^t \mathbf{y} \rightarrow \mathbf{1}(\pi^\top \mathbf{y})$: all nodes reach consensus. Thus, the low-rank setting admits a formal convergence guarantee for LP, unlike the general setting where convergence can fail.

Theorem 3.7 (Global Convergence). *Under the continuous-distribution assumption on the embeddings, the VLP iteration $\mathbf{Y}^{(t+1)} = \mathbf{P} \mathbf{Y}^{(t)}$ converges to $\mathbf{1}\pi^\top \mathbf{Y}^{(0)}$ as $t \rightarrow \infty$, where π is the unique stationary distribution of $\mathbf{P}$.*

Proof. By Lemmas 3.2, 3.4, and 3.6, $\mathbf{P}$ is non-negative, row-stochastic, irreducible, and aperiodic. The Perron–Frobenius theorem then yields a unique stationary distribution π with $\pi^\top \mathbf{P} = \pi^\top$ and $\mathbf{P}^t \rightarrow \mathbf{1}\pi^\top$ entrywise. Applying this to any column of $\mathbf{Y}^{(0)}$ gives $\mathbf{P}^t \mathbf{Y}^{(0)} \rightarrow \mathbf{1}\pi^\top \mathbf{Y}^{(0)}$. $\square$ $\square$

3.3.1 Practical Implications: Consensus is Trivial

While Theorem 3.7 provides a formal convergence guarantee, the limiting distribution is, in fact, *trivial* for downstream classification: it assigns the same label distribution $\pi^\top \mathbf{Y}^{(0)}$ to every node, irrespective of local connectivity. Run to convergence, VLP would homogenize the entire graph.

This is not a flaw of the algorithm but an inherent property of iterative diffusion on a fully connected graph. The practical value of label propagation lies not in the asymptotic limit but in the *intermediate* states. After a moderate number of iterations, label distributions reflect meaningful local structure: nodes in densely connected neighbourhoods share similar labels, while distinctions between well-separated clusters are preserved. Stopping early preserves this structure before global homogenization erases it.

Early stopping criteria. In practice, we use one of two strategies to determine when to halt propagation:

- **Fixed iteration count.** Run a predetermined number of iterations (e.g., $T = 50$ or $T = 100$) tuned for the dataset. This is simple and reproducible, and is the strategy used in the experiments of Section 3.4.
- **Convergence threshold.** Monitor $\|\mathbf{Y}^{(t+1)} - \mathbf{Y}^{(t)}\|_F$ and halt once it falls below a small ϵ (e.g., 10^{-4}). This adapts the iteration count to the data while still stopping well before the trivial consensus state.

Both strategies balance the need for sufficient diffusion (to propagate labels across the graph) against the need to preserve local distinctiveness. We use a fixed iteration count in our experiments to ensure a fair comparison with baselines that also run for a fixed number of iterations.

3.4 Experiments

We evaluate VLP on four real-world datasets: Amazon Products, Yelp, Taobao, and Flickr, each accessed through PyTorch Geometric [20]. A brief overview of their node counts and feature dimensionalities is given in Table 3.1. For Amazon Products, Flickr, and Yelp, node feature vectors come from the GraphSAINT repository [78]; Taobao is processed according to user–category usage data [31].

All experiments were conducted on Google Colab Pro with A100 GPU instances (83.5 GB of system RAM, 40 GB of GPU memory, 235.7 GB of disk). Source code is available at the URL listed in Chapter 1.

Table 3.1: Summary of the four datasets. The “ $N \times N$ entries” column indicates the number of cells that an explicit dense adjacency matrix would have to materialize.

Dataset	Nodes (N)	$N \times N$ entries	Feature dim. (d)
Flickr	89,249	7.9×10^9	500
Yelp	716,777	5.1×10^{11}	300
Taobao	936,946	8.7×10^{11}	66
Amazon Products	1,569,960	2.5×10^{12}	200

Choice of metrics. VLP is mathematically equivalent to adjacency-based LP (Theorem 3.1). Our experiments therefore focus on *scalability*: demonstrating that VLP scales to millions of nodes, whereas existing implementations cannot.

Choice of baselines. In our runtime evaluation, we compare VLP against three widely used implementations of the classical label propagation algorithm of Zhu and Ghahramani [82, 83]: *scikit-learn* [54], *scikit-network* [4], and *PyTorch Geometric* [20]. Each provides a linear propagation scheme in which labels are iteratively diffused across edges based on weighted adjacency. We focus on these popular implementations because other known approaches, such as k NN-pruned LP [24] or stabilized variants like LabelRank [74], achieve scalability by modifying the propagation process (e.g., through neighbourhood-voting heuristics, pruning edges, or altering the update rule), which differs fundamentally from the linear propagation setting considered here.

Experimental setup. Throughout our experiments, we assign each node a label from $[0, 50)$ at random, yielding a total of 50 label categories. We set the number of label propagation iterations (the inner loop) to 1000 for each library, and for every run we repeat the entire procedure five times and average the runtime. Both *scikit-learn* and *scikit-network* require an explicit adjacency matrix, which we precompute from node features via cosine similarity; this construction cost is included in their total runtime. Likewise, *PyTorch Geometric* requires constructing the edge list (on GPU or CPU) before running its label propagation operator. We include these precomputation times to ensure a fair end-to-end comparison.

We measure performance across datasets by sampling subgraphs at various thresholds. For each threshold, we keep a fraction of the node feature matrix to control the resulting subgraph size. We then run label propagation with all four implementations (VLP plus the three libraries) and record their average total runtime. This procedure is repeated for subgraphs of increasing size to illustrate each method’s scaling behaviour. When a library fails (due to memory or time constraints), we mark its entry with ‘X’. Finally, we consider 50% and 100% of the nodes to show full-scale performance.

Clamping in baseline implementations. The baselines differ in how they handle initially labelled nodes: *scikit-learn* applies hard clamping (labels fixed), *PyTorch Geometric* uses soft clamping controlled by a parameter α , and *scikit-network* does not explicitly document its policy. Our VLP, like the unclamped adjacency-based LP, lets all nodes update freely. These differences do not affect runtime, since the cost is dominated by the matrix multiplication and clamping is only an $O(Nc)$ step.

3.4.1 Runtime Results

Tables 4.3, 3.3, and 4.4 report runtime in seconds for the Amazon, Yelp, and Taobao datasets, respectively, with similar results for Flickr omitted for space.

Table 3.2: Runtime (in seconds) for Amazon over 1000 iterations. Bold denotes our VLP, always the fastest. ‘X’ indicates library failure.

Thr. (%)	Nodes	VLP	scikit-learn	scikit-net	PyG
0.2	3,139	0.18	0.75	7.10	4.81
0.4	6,279	0.19	2.58	22.36	21.10
0.6	9,419	0.22	6.03	50.63	112.61
0.8	12,559	0.23	11.32	92.57	X
1.0	15,699	0.25	18.38	134.11	X
50.0	784,980	6.85	X	X	X
100.0	1,569,960	13.37	X	X	X

As shown in Table 4.3, on Amazon we begin by sampling 0.2%, 0.4%, . . . , 1.0% of the nodes, corresponding to roughly 3,000 to 16,000 nodes. Across these subgraphs, VLP consistently yields the fastest runtimes, while scikit-learn, scikit-network, and PyTorch Geometric incur higher costs and eventually run out of memory once node counts exceed about 15,000. At 50% (784,980 nodes) and the full dataset (1,569,960 nodes), only VLP succeeds, with respective runtimes of **6.85**s and **13.37**s. At the highest threshold that any of the other methods can reach (1.0%), VLP is already two orders of magnitude faster than the best of them. This pattern holds across all datasets.

Table 3.3: Runtime (in seconds) for Yelp over 1000 iterations. VLP is always the fastest, in bold. ‘X’ indicates library failure.

Thr. (%)	Nodes	VLP	scikit-learn	scikit-net	PyG
0.4	2,867	0.59	0.80	6.45	4.63
0.8	5,734	0.17	2.88	18.70	17.99
1.2	8,602	0.19	5.44	38.60	70.31
1.6	11,469	0.20	9.49	67.38	X
2.0	14,336	0.25	14.44	114.25	X
50.0	358,423	3.96	X	X	X
100.0	716,777	7.72	X	X	X

Table 3.3 presents the Yelp results. Up to around 8,000–11,000 nodes, scikit-learn and scikit-network both succeed but cost more time than VLP for each row, while

PyTorch Geometric fails once node counts surpass roughly 11,000. At 50% (358,423 nodes) and 100% (716,777 nodes), only VLP can perform label propagation, taking **3.96**s and **7.72**s, respectively, whereas all others run out of memory.

Table 3.4: Runtime (in seconds) for Taobao over 1000 iterations. VLP times are in bold; ‘X’ denotes failure.

Thr. (%)	Nodes	VLP	scikit-learn	scikit-net	PyG
0.5	4,684	0.17	1.69	12.99	12.53
1.0	9,369	0.19	5.79	46.60	108.84
1.5	14,054	0.21	12.62	99.20	X
2.0	18,738	0.23	22.51	174.90	X
2.5	23,423	0.22	X	X	X
3.0	28,108	0.24	X	X	X
50.0	468,473	3.06	X	X	X
100.0	936,946	5.94	X	X	X

Finally, Table 4.4 reports Taobao, which similarly features near-complete edge density once users are mapped to categories via TF-IDF. Even at small thresholds (e.g. 9,369 nodes), scikit-learn and scikit-network remain slower, while PyTorch Geometric fails above 14,000 nodes. VLP alone scales to 50% (468,473 nodes) and 100% (936,946 nodes) of the dataset, requiring only **3.06**s and **5.94**s respectively to propagate 50 random labels for all nodes.

In every dataset, VLP outperforms adjacency-based label propagation, irrespective of graph size. Once node counts surpass around 11,000 for PyTorch Geometric or 15,000–20,000 for scikit-learn and scikit-network, these methods exceed their memory budgets or exhibit runtime failures. At half or all of each dataset, only VLP completes successfully. This consistent advantage shows that VLP is the only viable option among these implementations for large, low-rank graphs where adjacency construction becomes infeasible.

3.4.2 Empirical Verification of Exactness

Theorem 3.1 guarantees that VLP and the adjacency-based formulation produce identical outputs up to floating-point rounding. To verify this empirically, we compared the final label distributions of VLP against the scikit-learn baseline on small sub-graphs (where scikit-learn could still run). The mean squared error between the two outputs was within floating-point tolerance ($< 10^{-7}$), confirming that VLP recovers

the exact diffusion dynamics of the fully connected graph. Table 3.5 reports a representative comparison on a 10,000-node subsample, where downstream classification metrics agree to four decimal places.

Table 3.5: Empirical exactness verification on a 10,000-node subsample of Amazon. Downstream metrics for VLP and the explicit baseline agree to within numerical precision.

Metric	Explicit (scikit-learn)	Implicit (VLP)	Difference
Micro-F1	0.4215	0.4215	0.0
Macro-F1	0.3892	0.3892	0.0
MSE on output logits	—	—	$< 10^{-7}$

3.4.3 Sensitivity to the Embedding Dimension

The complexity bound $\mathcal{O}(Ndc)$ predicts that VLP runtime should scale linearly in the embedding dimension d . We verified this on Flickr by running VLP with embedding dimensions ranging from 64 to 768. Runtime increased from approximately 0.12s to 1.1 s, consistent with the linear prediction. This highlights a practical trade-off: VLP is extremely efficient for compact embeddings (e.g., low-dimensional SVD or quantized vectors) and remains memory-linear for very wide vectors (such as 4096-dimensional LLM embeddings), although the constant factor grows with d .

3.5 Chapter Summary

In this chapter, we introduced Vector-Based Label Propagation (VLP), a reformulation of classical label propagation for the practically important class of low-rank dense graphs. By writing the adjacency in factored form $\mathbf{A} = \mathbf{V}\mathbf{V}^\top$ and reordering matrix multiplications, VLP avoids ever materializing the $N \times N$ adjacency, reducing memory from $\mathcal{O}(N^2)$ to $\mathcal{O}(Nd)$ and per-iteration cost from $\mathcal{O}(N^2c)$ to $\mathcal{O}(Ndc)$, while remaining mathematically exact (Theorem 3.1). We further established a formal convergence guarantee for the low-rank setting (Theorem 3.7), discussed why the limiting consensus state is trivial in practice, and validated the approach experimentally on graphs with up to 1.5 million nodes, where standard implementations exhaust memory at around 15,000 nodes.

VLP demonstrates the power of the implicit-graph paradigm for information diffusion on a fixed topology. Many graph mining tasks, however, require more than passive signal propagation: they demand active restructuring of the graph itself. Chapter 4 extends the implicit approach to hierarchical community detection, where the algorithm must repeatedly aggregate nodes into supernodes. We will see that even topology-changing operations admit a graph-free reformulation while preserving exact equivalence to the classical method.

Chapter 4

VLouvain: Structure Discovery

4.1 Introduction

Community detection is a fundamental task in graph analysis: given a graph, the goal is to partition it into subgroups (communities) such that nodes are densely connected within the same group and more sparsely connected across groups [29, 53, 22]. Among the most widely used methods are the Louvain [3] and Leiden [71] algorithms, both of which uncover such communities while also producing multi-level partitions. This hierarchical nature is valuable across a broad range of applications, from social network analysis to recommender systems [26, 27, 37, 65] and graph-based retrieval-augmented generation [16], since it reveals structure at multiple scales.

In this chapter we focus on the Louvain algorithm, which iteratively merges nodes into communities based on modularity gains and then aggregates each community into a supernode, building a coarser graph at every level. While powerful and widely adopted, Louvain faces significant scalability issues when applied to large graphs. A decade of work has aimed at improving Louvain’s efficiency [56, 28, 61], yet the need for explicit graph construction remains a major bottleneck for the low-rank graphs introduced in Chapter 2. When edges are derived from node feature vectors, the number of edges grows quadratically with the node count, and the resulting graph has $\mathcal{O}(N^2)$ edges. Storing and processing such a graph in memory is infeasible at scale, particularly in domains such as recommender systems, where user or item embeddings produce very dense networks.

In practice, node feature similarities such as cosine similarities can generate a near-complete graph. Traditional approaches construct or store the entire graph before

running Louvain. For large low-rank graphs, building and managing that structure is expensive in both time and memory. Libraries such as iGraph [12] and NetworKit [68] can handle sizable networks but still struggle with dense graphs, and even efficient variants like GVE-Louvain [61] face constraints once the graph becomes too dense.

A practical motivation. The theoretical framework developed in this chapter was motivated by hands-on work at the Vancouver Island Health Authority (VIHA), where the task was to cluster millions of patient profiles based on high-dimensional clinical features (such as frequencies of diagnostic codes). Because these count vectors are non-negative, every pair of patients has some non-zero cosine similarity, so the induced patient-similarity graph is dense by construction. Storing the pairwise similarities for millions of patients would produce on the order of 10^{12} edges and require terabytes of memory, far beyond what any standalone workstation can hold. At the same time, the research group had dedicated NVIDIA A100 GPUs available but these were largely idle, because traditional graph-mining libraries rely on pointer-chasing and random memory access rather than the dense matrix operations that modern accelerators are built for. VLouvain emerged directly from this gap: an algorithm that keeps the graph implicit in the embedding matrix and casts Louvain’s inner loop as dense matrix arithmetic, so that the A100’s tensor cores can actually do the work.

Our approach. We propose **VLouvain** (Vector-Based Louvain), which avoids constructing large adjacency matrices or iterating over individual edges by using direct vector computations to aggregate node similarities, capturing only the essential information needed for Louvain’s modularity-based updates. Unlike traditional graph-centric approaches that enumerate or recompute edge weights, VLouvain bypasses this bottleneck by directly computing the aggregate terms required for modularity evaluation, eliminating the need to explicitly generate, store, or traverse every edge. Since each edge’s influence is ultimately reflected in a community-level score, our method maintains the mathematical integrity of the Louvain algorithm while achieving a substantial reduction in memory usage.

Contributions of this chapter. The main contributions are:

- **Memory Efficiency.** Our method dispenses with storing the large adjacency matrix, keeping only node features and minimal aggregated statistics. This substantially lowers memory requirements in the low-rank setting.

- **Matrix-Based Implementation.** Our approach replaces traditional nested node-edge loops with matrix or vector-based aggregation. This enables a vectorized supernode representation for faster multi-level comparisons of communities and efficient GPU acceleration. In particular, we maintain community state in replicated $N \times d$ matrices and update them via boolean masking, a design that trades redundant storage for the elimination of pointer-chasing and dictionary lookups.
- **Mathematical Equivalence.** Although our implementation avoids explicit edge storage, it follows the Louvain framework [3] and is guaranteed to produce the same partitioning results as the original algorithm. We formalize this as Theorem 4.2 and prove it via three lemmas covering degree, modularity gain, and aggregation equivalence.
- **Scalability.** On dense low-rank graphs derived from real datasets, VLouvain scales to millions of nodes on a single GPU, in regimes where state-of-the-art community-detection libraries (cuGraph, GVE, iGraph, NetworkKit) exhaust memory.

Chapter 2 introduced modularity, the standard Louvain algorithm (Algorithm 1), and the implicit-graph framework $\mathbf{A} = \mathbf{V}\mathbf{V}^\top$. Chapter 3 showed how the implicit framework applies to information diffusion via Vector-Based Label Propagation. Whereas VLP operates on a fixed topology, Louvain actively restructures the graph by repeatedly merging nodes into supernodes at each level. The question we address in this chapter is: how can we carry out these topology-modifying operations without ever materializing the adjacency matrix?

4.2 Vector-Based Louvain: Key Operations and Data Structures

In this section, we outline the vector-based formulation underlying VLouvain. Our goal is community detection directly from node embeddings, without explicitly constructing the full graph. As developed in Chapters 2 and 3, many practical graphs can be represented in low-rank form,

$$\mathbf{A} = \mathbf{V}\mathbf{V}^\top, \quad \mathbf{V} \in \mathbb{R}^{N \times d}, \quad (4.1)$$

where each row $\mathbf{v}_i \in \mathbb{R}^d$ is a feature vector (or embedding) for node i . This view naturally captures graphs in which edges encode similarity between nodes, since numerous affinity measures, such as normalized cosine similarity or even-degree polynomial kernels, can be expressed as inner products in an appropriate feature space. To ensure compatibility with modularity optimization, the resulting inner products must be non-negative.

Cosine similarity as a special case. In our experiments we adopt the non-negative augmented cosine similarity derived in Chapter 3 (Section 3.2.2): each unit-norm embedding is augmented as $\mathbf{v}'_i = [\mathbf{v}_i; 1]/\sqrt{2}$, so that $\mathbf{v}'_i{}^\top \mathbf{v}'_j = (1 + \cos \theta_{ij})/2 \in [0, 1]$, and the full adjacency is $\mathbf{A} = \mathbf{V}\mathbf{V}^\top$ with the diagonal set to zero, where $\mathbf{V} \in \mathbb{R}^{N \times (d+1)}$ collects the augmented rows. As in Chapter 3, we drop the prime in what follows and simply write $\mathbf{v}_i$ for the augmented vectors.

4.2.1 Degree Calculation

The degree vector $\mathbf{k}$ is computed implicitly using exactly the formula derived in Chapter 3, Section 3.2.3: $\mathbf{k} = \mathbf{V}(\mathbf{V}^\top \mathbf{1}_N) - \mathbf{s}$, where $\mathbf{s}_i = \mathbf{v}_i \cdot \mathbf{v}_i$ is the self-loop weight of node i . The derivation is the same bilinearity argument used there: a sum of pairwise inner products equals the inner product with a sum of vectors, so we can compute all degrees in $\mathcal{O}(Nd)$ without ever forming $\mathbf{V}\mathbf{V}^\top$. Precomputing $\mathbf{k}$ is essential for modularity optimization, since it captures global graph information with minimal overhead.

4.2.2 Sum of Edge Weights

The total edge weight m is half the sum of all degrees, so each edge is counted once:

$$m = \frac{1}{2} \sum_i k_i = \frac{1}{2} \mathbf{1}^\top \mathbf{k}. \quad (4.2)$$

4.2.3 Affinity and Resolution Terms

Recall from Chapter 2 that the modularity

$$Q = \frac{1}{2m} \sum_{i,j} \left(A_{ij} - \gamma \frac{k_i k_j}{2m} \right) \delta(c_i, c_j) \quad (4.3)$$

can be decomposed into two components for nodes in the same community: the observed edge weights and the expected edge weight scaled by γ . We call these the *affinity term*

$$Q_{\text{aff}} = \sum_{i,j} A_{ij} \delta(c_i, c_j) \quad (4.4)$$

and the *resolution term*

$$Q_{\text{res}} = \gamma \sum_{i,j} \frac{k_i k_j}{2m} \delta(c_i, c_j). \quad (4.5)$$

The affinity term measures the actual edges observed within communities, while the resolution term estimates what we would expect from degree-based random connections.

4.2.4 Modularity Gain Calculation

When a node i moves from its current community c_i to a candidate community c'_i , the net modularity gain ΔQ is the difference of an insertion gain ΔQ_{in} and a removal loss ΔQ_{out} :

$$\Delta Q = \Delta Q_{\text{in}} - \Delta Q_{\text{out}}. \quad (4.6)$$

Insertion Gain. When adding node i to a community $c'_i \neq c_i$, the affinity contribution is $\sum_{j \in c'_i} A_{ij} = \mathbf{v}_i \cdot V^{c'_i}$, and the resolution contribution is $\gamma \frac{k_i}{2m} K^{c'_i}$. Since $i \notin c'_i$, neither $V^{c'_i}$ nor $K^{c'_i}$ contains i 's own contribution, so no self-loop correction is needed.

Removal Loss. The removal loss is the modularity contribution of i while it is in c_i , and requires a self-loop correction: because V^{c_i} already includes $\mathbf{v}_i$ and K^{c_i} already includes k_i , we must subtract $\mathbf{v}_i \cdot \mathbf{v}_i$ from the affinity and $\gamma k_i^2 / (2m)$ from the resolution term before interpreting the result as a “removal” contribution. Equivalently, the removal loss is computed as $\mathbf{v}_i \cdot (V^{c_i} - \mathbf{v}_i)$ and $\gamma \frac{k_i}{2m} (K^{c_i} - k_i)$. This asymmetry between insertion and removal (whether the aggregate contains i or not) is the only bookkeeping subtlety in the vector formulation, and it is handled by the single correction term shown in Equation (4.12) of the next section.

The crucial observation is that both the insertion gain and the removal loss are computed directly from two community-level aggregates: a community vector $V^c = \sum_{j \in c} \mathbf{v}_j$ and a community degree sum $K^c = \sum_{j \in c} k_j$. The inner product $\mathbf{v}_i \cdot V^c$

simultaneously represents all $|c|$ pairwise affinities in the sense that, by bilinearity,

$$\sum_{j \in c} A_{ij} = \sum_{j \in c} (\mathbf{v}_i \cdot \mathbf{v}_j) = \mathbf{v}_i \cdot V^c. \quad (4.7)$$

We do not need to enumerate individual edges; the collective affinity between i and a community is perfectly encapsulated by a single dot product.

4.2.5 Community Movement and Hardware-Aware State

In the Louvain procedure, each node looks for a community that maximizes its ΔQ . Initially, every node i starts in its own community $c_i = \{i\}$, with $V^{c_i} \leftarrow \mathbf{v}_i$ and $K^{c_i} \leftarrow k_i$. When node i moves from community c to community c' , the aggregates are updated:

$$V^c -= \mathbf{v}_i, \quad K^c -= k_i, \quad V^{c'} += \mathbf{v}_i, \quad K^{c'} += k_i. \quad (4.8)$$

Because ΔQ depends only on these aggregates, there is no need to enumerate edges individually. Storing V^c and K^c for each community keeps the memory overhead at $\mathcal{O}(Nd)$ instead of $\mathcal{O}(N^2)$.

A hardware-aware data layout. A naive implementation would maintain a dynamic dictionary mapping community IDs to V^c, K^c values, resizing and rehashing as nodes move. On modern GPUs this is exactly the wrong data structure: every community lookup becomes a pointer chase, every update triggers a cache miss, and the memory access pattern cannot be vectorized. To eliminate this bottleneck, VLoUvain instead maintains two $N \times d$ community-tracking matrices $\mathbf{VC}$ and $\mathbf{KC}$ with one row per node (not one row per community). Initially, $\mathbf{VC} = \mathbf{V}$ and $\mathbf{KC} = \mathbf{K}$, because each node is its own community. When node i moves from community c_i to community c_j , we locate all nodes currently in c_i via boolean indexing (using the assignment vector $\mathbf{C}$), subtract $\mathbf{v}_i$ and k_i from those rows, and then add $\mathbf{v}_i$ and k_i to the rows of nodes in c_j :

$$\mathbf{VC}[\mathbf{C} == c_i] -= \mathbf{v}_i, \quad \mathbf{KC}[\mathbf{C} == c_i] -= k_i, \quad (4.9)$$

$$\mathbf{VC}[\mathbf{C} == c_j] += \mathbf{v}_i, \quad \mathbf{KC}[\mathbf{C} == c_j] += k_i. \quad (4.10)$$

Replicating each community’s aggregates across all of its member rows is redundant in the information-theoretic sense, but it pays for itself by eliminating the pointer-chasing entirely. The matrices $\mathbf{VC}$ and $\mathbf{KC}$ remain fixed-size $N \times d$ and N throughout the optimization, all updates are dense vector operations on contiguous memory, and the loop over candidate communities reduces to a single matrix-vector product $\mathbf{v}_i \cdot \mathbf{VC}$.

4.3 The VLouvain Algorithm

We now present the VLouvain algorithm, following the structure of the published version [3] (adapted to the vector setting). VLouvain consists of three components:

1. the *Optimized One-Level Function* (OOLF), which performs modularity optimization at a single level;
2. the *Optimized New Graph Generation* (ONGG), which collapses the current assignments into supernodes; and
3. the *Optimized Louvain Partition Algorithm*, which alternates OOLF and ONGG until no further modularity gain is achieved.

4.3.1 Optimized One-Level Function

Our optimized one-level function leverages node-vector representations and degree arrays to perform modularity optimization without constructing or storing explicit edge weights (Algorithm 5). At the start, the node-vector matrix $\mathbf{V}$ and the degree array $\mathbf{K}$ are copied into the community-tracking matrices $\mathbf{VC} \leftarrow \mathbf{V}$ and $\mathbf{KC} \leftarrow \mathbf{K}$, since each node begins in its own community, captured by the assignment vector $\mathbf{C}$.

For each node i , the algorithm first computes a vector of candidate contributions against every community in parallel,

$$\mathbf{q}_{\text{in}} \leftarrow \frac{1}{2m} (\mathbf{v}_i \cdot \mathbf{VC} - \gamma \frac{k_i}{2m} \mathbf{KC}), \quad (4.11)$$

whose entry j is the modularity contribution of placing node i in the community that currently holds node j . The minus sign matches the standard Louvain modularity gain formula from Chapter 2, in which affinity is offset by the resolution-based null-model penalty. This single matrix-vector expression evaluates all candidate communities in one GPU-friendly operation.

There is one subtlety. The entry $\mathbf{q}_{\text{in}}[i]$ corresponds to “placing i in its own current community”, but the aggregate $\mathbf{VC}[i] = V^{c_i}$ already contains $\mathbf{v}_i$ itself, and similarly $\mathbf{KC}[i]$ already includes k_i . To obtain the correct “contribution while remaining in c_i ” (which the move decision must compare against), we apply a self-loop correction to this single entry:

$$q_{\text{out}} \leftarrow \mathbf{q}_{\text{in}}[i] - \frac{1}{2m} \left(\mathbf{v}_i \cdot \mathbf{v}_i - \gamma \frac{k_i^2}{(2m)} \right). \quad (4.12)$$

The subtracted term $(\mathbf{v}_i \cdot \mathbf{v}_i - \gamma k_i^2 / (2m)) / (2m)$ is exactly the amount by which $\mathbf{q}_{\text{in}}[i]$ overstates the contribution of i when its own vector and degree have not been removed from the aggregate. After this correction, q_{out} represents the modularity contribution of node i to community c_i with self-loop terms excluded, matching the explicit Louvain convention.

Subtracting q_{out} from each entry of $\mathbf{q}_{\text{in}}$ yields $\mathbf{q}$, the net modularity gain of moving i to each candidate community. The algorithm then selects $j = \arg \max_j \mathbf{q}[j]$ subject to $\mathbf{q}[j] > 0$, and if $c_j \neq c_i$, reassigns i to community c_j and updates $\mathbf{VC}$ and $\mathbf{KC}$ using the boolean-indexing scheme from Section 4.2.5.

Within each level, node updates are repeated and the loop stops when the sweep-level modularity gain falls below a small tolerance ($\varepsilon = 10^{-7}$ in our experiments). After the first pass most nodes stabilize, and later passes are extremely fast because only a few nodes still change communities.

Algorithm 5 Optimized One-Level Function (OOLF)

Require: Matrix $\mathbf{V}$, degree array $\mathbf{K}$, initial communities $\mathbf{C}$, resolution γ , total edge weight m , tolerance $\varepsilon > 0$

Ensure: Updated $\mathbf{C}$, matrices $\mathbf{VC}$, $\mathbf{KC}$, modularity gain ΔQ

```

1:  $\mathbf{VC} \leftarrow \mathbf{V}$ ,  $\mathbf{KC} \leftarrow \mathbf{K}$ ,  $\Delta Q \leftarrow 0$ 
2: while true do
3:    $\Delta Q_{\text{sweep}} \leftarrow 0$ 
4:   for each node  $i$  do
5:      $\mathbf{q}_{\text{in}} \leftarrow \frac{\mathbf{v}_i \cdot \mathbf{VC} - \gamma \frac{k_i}{2m} \mathbf{KC}}{2m}$ 
6:      $q_{\text{out}} \leftarrow \mathbf{q}_{\text{in}}[i] - \frac{\mathbf{v}_i \cdot \mathbf{v}_i - \gamma \frac{k_i^2}{2m}}{2m}$   $\triangleright$  self-loop correction
7:      $\mathbf{q} \leftarrow \mathbf{q}_{\text{in}} - q_{\text{out}}$ 
8:      $j \leftarrow \arg \max_j \mathbf{q}[j]$  s.t.  $\mathbf{q}[j] > 0$ 
9:     if  $c_j \neq c_i$  then
10:       $\mathbf{VC}[\mathbf{C} == c_i] -= \mathbf{v}_i$ ,  $\mathbf{KC}[\mathbf{C} == c_i] -= k_i$ 
11:       $\mathbf{VC}[\mathbf{C} == c_j] += \mathbf{v}_i$ ,  $\mathbf{KC}[\mathbf{C} == c_j] += k_i$ 
12:       $\mathbf{VC}[i] \leftarrow \mathbf{VC}[j]$ ,  $\mathbf{KC}[i] \leftarrow \mathbf{KC}[j]$ 
13:       $c_i \leftarrow c_j$ ,  $\Delta Q += \mathbf{q}[j]$ ,  $\Delta Q_{\text{sweep}} += \mathbf{q}[j]$ 
14:    end if
15:  end for
16:  if  $\Delta Q_{\text{sweep}} \leq \varepsilon$  then
17:    break
18:  end if
19: end while
20: return  $\mathbf{C}$ ,  $\mathbf{VC}$ ,  $\mathbf{KC}$ ,  $\Delta Q$ 

```

Because the community updates depend only on the aggregated vectors rather than explicit edge weights, memory demand is greatly reduced: storing $\mathbf{VC}$ and $\mathbf{KC}$ requires $\mathcal{O}(Nd)$ space, which is far more efficient than the $\mathcal{O}(N^2)$ cost of explicitly managing all edges.

4.3.2 Optimized New Graph Generation

After the first phase, the algorithm uses the community assignment vector $\mathbf{C}$ to collapse nodes of the same community into a single supernode (Algorithm 6). This step preserves the total edge weight between original communities. The key observation

is that, with $A_{ij} = \mathbf{v}_i \cdot \mathbf{v}_j$,

$$\sum_{i \in C_1, j \in C_2} A_{ij} = \left(\sum_{i \in C_1} \mathbf{v}_i \right) \cdot \left(\sum_{j \in C_2} \mathbf{v}_j \right), \quad (4.13)$$

so a simple summation of the member vectors within a community already yields the vector representation of the corresponding supernode, and the inner product of two supernode vectors is exactly the total edge weight between them. This is the *Vector Supernode* property: graph coarsening reduces to vector addition. It is formalized below as Theorem 4.1.

Operationally, once we identify the unique communities in $\mathbf{C}$, we collect the indices $\mathbf{I}_c$ of all nodes that share a community label c and aggregate the rows of $\mathbf{VC}$ and $\mathbf{KC}$ for those indices. Because $\mathbf{VC}$ and $\mathbf{KC}$ already store replicated per-community aggregates (Section 4.2.5), we can simply choose any representative index in $\mathbf{I}_c$ and copy its row. The result is stored in $\mathbf{V}^{\text{new}}$ and $\mathbf{K}^{\text{new}}$, where each row now corresponds to a supernode.

Algorithm 6 Optimized New Graph Generation (ONGG)

Require: Community assignments $\mathbf{C}$, matrices $\mathbf{VC}$, $\mathbf{KC}$

Ensure: Supernode matrix $\mathbf{V}^{\text{new}}$, supernode degrees $\mathbf{K}^{\text{new}}$

- 1: $\mathbf{C}^{\text{unique}} \leftarrow \text{unique}(\mathbf{C})$
 - 2: **for** each unique community $c \in \mathbf{C}^{\text{unique}}$ **do**
 - 3: Identify node indices $\mathbf{I}_c \leftarrow \{i : \mathbf{C}[i] = c\}$
 - 4: Choose a representative index $i_c \in \mathbf{I}_c$
 - 5: $\mathbf{V}^{\text{new}}[c] \leftarrow \mathbf{VC}[i_c]$
 - 6: $\mathbf{K}^{\text{new}}[c] \leftarrow \mathbf{KC}[i_c]$
 - 7: **end for**
 - 8: **return** $\mathbf{V}^{\text{new}}, \mathbf{K}^{\text{new}}$
-

4.3.3 Optimized Louvain Partition Algorithm

The Optimized Louvain Partition Algorithm (Algorithm 7) refines community assignments by iterating between OOLF and ONGG. After computing the degree vector $\mathbf{k}$ and the total edge weight m , it initializes community assignments, sets a resolution parameter γ and a convergence threshold ϵ , and optionally maintains a partition list to record the hierarchy. Each iteration runs OOLF to update $\mathbf{C}$, $\mathbf{VC}$, $\mathbf{KC}$ and accu-

mulate a total modularity gain ΔQ . If $\Delta Q > \epsilon$, ONGG is invoked to produce a new coarsened graph, which replaces $\mathbf{V}$ and $\mathbf{k}$; otherwise, the algorithm terminates.

Algorithm 7 Optimized Louvain Partition Algorithm

Require: Matrix $\mathbf{V}$, resolution parameter γ , threshold ϵ

Ensure: Hierarchical partition list

- 1: Compute degrees $\mathbf{k}$ (Section 4.2.1) and total edge weight m
 - 2: Initialize PartitionList $\leftarrow \emptyset$
 - 3: **repeat**
 - 4: Initialize $\mathbf{C}$ so that each node is its own community
 - 5: $\mathbf{C}, \mathbf{VC}, \mathbf{KC}, \Delta Q \leftarrow \text{OOLF}(\mathbf{V}, \mathbf{k}, \mathbf{C}, \gamma, m)$
 - 6: **if** $\Delta Q > \epsilon$ **then**
 - 7: $\mathbf{V}^{\text{new}}, \mathbf{K}^{\text{new}} \leftarrow \text{ONGG}(\mathbf{C}, \mathbf{VC}, \mathbf{KC})$
 - 8: $\mathbf{V} \leftarrow \mathbf{V}^{\text{new}}, \quad \mathbf{k} \leftarrow \mathbf{K}^{\text{new}}$
 - 9: Append $\mathbf{C}$ to PartitionList
 - 10: **end if**
 - 11: **until** $\Delta Q \leq \epsilon$
 - 12: **return** PartitionList
-

4.3.4 Time and Memory Cost

The VLouvain algorithm’s time and memory requirements chiefly depend on the optimized one-level function. During the first call, each node i performs an inner product $\mathbf{v}_i \cdot \mathbf{VC}$ of cost $\mathcal{O}(Nd)$, plus a scalar multiplication for the resolution term $\gamma \frac{k_i}{2m} \mathbf{KC}$ of cost $\mathcal{O}(N)$. Combining these yields $\mathcal{O}(Nd)$ work per node and $\mathcal{O}(N^2d)$ overall. Although the worst-case time complexity is $\mathcal{O}(N^2d)$ due to inner-product evaluations, in practice $d \ll N$ (for example $d \approx 200$), so the cost is effectively comparable to $\mathcal{O}(N^2)$. Crucially, the computation is tileable and bandwidth-bound on GPUs; we reuse tiles across local-move evaluations and cache community aggregates to reduce redundant work. On an A100, the dense matrix-vector products map directly onto tensor cores, which is what makes the algorithm tractable on datasets of a million nodes, where explicit methods exhaust memory entirely.

The memory cost includes $\mathbf{V}$ of size $N \times d$, $\mathbf{k}$ of size N , the community copies $\mathbf{VC}, \mathbf{KC}$, and the assignment vector $\mathbf{C}$, totaling approximately $2(N \times d) + 3N$ or $\mathcal{O}(Nd)$. This is well below the $\mathcal{O}(N^2)$ storage required by explicit adjacency formats, making VLouvain effective for large low-rank graphs when $d \ll N$.

Relation to industrial representation learning. The decision to work in inner-product space is not merely a mathematical convenience. Modern industrial representation learning, particularly in recommender systems based on two-tower architectures, already produces embeddings optimized for maximum inner product search (MIPS) rather than Euclidean distance. VLouvain’s modularity gain formula ΔQ is a series of dot products $\mathbf{v}_i^\top V^c$, which aligns natively with the way such embeddings are trained and indexed. VLouvain can therefore be applied as a drop-in topological clustering layer on top of existing two-tower embeddings without any metric-space conversion.

4.4 Theoretical Analysis: Exactness

A common misconception about scalable graph mining is that efficiency necessarily requires approximation. We emphasize that VLouvain is neither an approximation nor a heuristic: it produces the same partition sequence as the standard Louvain algorithm run on the explicit graph. We formalize this as the following theorem, whose proof rests on three lemmas covering degree equivalence, modularity-gain equivalence, and aggregation equivalence.

Theorem 4.1 (Vector Supernode). *Let C_a and C_b be two disjoint sets of nodes with embeddings $\{\mathbf{v}_i\}_{i \in C_a}$ and $\{\mathbf{v}_j\}_{j \in C_b}$, and let $V^a = \sum_{i \in C_a} \mathbf{v}_i$ and $V^b = \sum_{j \in C_b} \mathbf{v}_j$. Under the implicit adjacency $A_{ij} = \mathbf{v}_i \cdot \mathbf{v}_j$, the total edge weight between C_a and C_b equals the inner product of the supernode vectors:*

$$\sum_{i \in C_a} \sum_{j \in C_b} A_{ij} = V^a \cdot V^b. \quad (4.14)$$

Proof. By bilinearity of the inner product,

$$V^a \cdot V^b = \left(\sum_{i \in C_a} \mathbf{v}_i \right) \cdot \left(\sum_{j \in C_b} \mathbf{v}_j \right) = \sum_{i \in C_a} \sum_{j \in C_b} (\mathbf{v}_i \cdot \mathbf{v}_j) = \sum_{i \in C_a} \sum_{j \in C_b} A_{ij}. \quad \square$$

□

Theorem 4.2 (Exactness of VLouvain). *Let $\mathcal{P}_{\text{exp}}$ be the partition sequence produced by the standard Louvain algorithm (Algorithm 1) on the explicit graph $G = (V, E)$ with adjacency $\mathbf{A}$. Let $\mathcal{P}_{\text{vlv}}$ be the partition sequence produced by the Optimized Louvain Partition Algorithm (Algorithm 7) on the embedding matrix $\mathbf{V}$ with $\mathbf{A} = \mathbf{V}\mathbf{V}^\top$ –*

$\text{diag}(\mathbf{V}\mathbf{V}^\top)$. Under identical node-visitation order and identical tie-breaking rules, $\mathcal{P}_{\text{exp}} = \mathcal{P}_{\text{vlv}}$.

Proof. Both algorithms follow the same greedy schedule: at each level they iterate over nodes, compute a modularity gain ΔQ for each candidate community, move to the best positive-gain community, and terminate the level when no further improvement is possible. They then aggregate the communities into supernodes and repeat. It therefore suffices to show that every intermediate quantity computed by VLouvain equals the corresponding quantity in the explicit algorithm:

1. Degree Equivalence (Lemma 4.3): the degree vector $\mathbf{k}$ computed implicitly equals the explicit degrees.
2. Modularity Gain Equivalence (Lemma 4.4): for any node i and any candidate community c , the net gain ΔQ computed from $(\mathbf{v}_i, V^c, k_i, K^c)$ equals the explicit gain computed from edge weights.
3. Aggregation Equivalence (Lemma 4.5): after a level completes, the supernode matrix produced by ONGG corresponds to an implicit adjacency that is identical to the explicit aggregated graph.

Given these three lemmas, one level of VLouvain performs exactly the same state transition as one level of explicit Louvain. By induction on the hierarchy level, the two partition sequences coincide. $\square$ $\square$

Lemma 4.3 (Degree Equivalence). *Let $\mathbf{v}_i$ denote the augmented embedding of node i and let $\mathbf{V}_{\text{sum}} = \sum_{j=1}^N \mathbf{v}_j$. Then the degree $k_i^{\text{imp}} = \mathbf{v}_i \cdot \mathbf{V}_{\text{sum}} - \mathbf{v}_i \cdot \mathbf{v}_i$ computed implicitly equals the explicit degree $k_i^{\text{exp}} = \sum_{j \neq i} A_{ij}$.*

Proof. Under the implicit adjacency convention $A_{ij} = \mathbf{v}_i \cdot \mathbf{v}_j$ (for $i \neq j$), bilinearity gives

$$\mathbf{v}_i \cdot \mathbf{V}_{\text{sum}} = \sum_{j=1}^N (\mathbf{v}_i \cdot \mathbf{v}_j) = \sum_{j \neq i} A_{ij} + \mathbf{v}_i \cdot \mathbf{v}_i.$$

Subtracting $\mathbf{v}_i \cdot \mathbf{v}_i$ leaves $k_i^{\text{imp}} = \sum_{j \neq i} A_{ij} = k_i^{\text{exp}}$, which is precisely the explicit-graph degree. $\square$ $\square$

Lemma 4.4 (Modularity Gain Equivalence). *For any node i , any current community c_i , and any candidate target community c_j with $c_j \neq c_i$, the net modularity gain $\Delta Q = \mathbf{q}_{\text{in}}[j] - q_{\text{out}}$ computed by Algorithm 5 from the aggregates $(\mathbf{v}_i, V^{c_i}, V^{c_j}, k_i, K^{c_i}, K^{c_j})$ equals the standard Louvain modularity gain computed on the explicit edge list.*

Proof. Let V^{c_i} and K^{c_i} denote the community aggregates *including* node i 's own contribution (as they are stored in $\mathbf{VC}$ and $\mathbf{KC}$), and let V^{c_j} and K^{c_j} denote the aggregates for the target community, which does *not* contain i .

Insertion side. The insertion entry is

$$\mathbf{q}_{\text{in}}[j] = \frac{1}{2m} \left(\mathbf{v}_i \cdot V^{c_j} - \gamma \frac{k_i}{2m} K^{c_j} \right).$$

By Theorem 4.1 applied to $C_a = \{i\}$ and $C_b = c_j$, we have $\mathbf{v}_i \cdot V^{c_j} = \sum_{k \in c_j} A_{ik}$, which is exactly the explicit affinity of i to c_j . The resolution term $\gamma \frac{k_i}{2m} K^{c_j}$ matches the explicit null-model expectation, since $K^{c_j} = \sum_{k \in c_j} k_k$ by Lemma 4.3. Hence $\mathbf{q}_{\text{in}}[j]$ equals the standard insertion contribution.

Removal side. For the removal side, the algorithm cannot read $\mathbf{q}_{\text{in}}[i]$ directly as the removal loss, because V^{c_i} and K^{c_i} include i itself. Expanding,

$$\begin{aligned} \mathbf{q}_{\text{in}}[i] &= \frac{1}{2m} \left(\mathbf{v}_i \cdot V^{c_i} - \gamma \frac{k_i}{2m} K^{c_i} \right) \\ &= \frac{1}{2m} \left(\left[\mathbf{v}_i \cdot (V^{c_i} - \mathbf{v}_i) + \mathbf{v}_i \cdot \mathbf{v}_i \right] - \gamma \frac{k_i}{2m} \left[(K^{c_i} - k_i) + k_i \right] \right). \end{aligned}$$

The self-loop correction of Equation (4.12) removes the two extra terms:

$$q_{\text{out}} = \mathbf{q}_{\text{in}}[i] - \frac{1}{2m} \left(\mathbf{v}_i \cdot \mathbf{v}_i - \gamma \frac{k_i^2}{2m} \right) = \frac{1}{2m} \left(\mathbf{v}_i \cdot (V^{c_i} - \mathbf{v}_i) - \gamma \frac{k_i}{2m} (K^{c_i} - k_i) \right).$$

Applying Theorem 4.1 to $C_a = \{i\}$ and $C_b = c_i \setminus \{i\}$, we have $\mathbf{v}_i \cdot (V^{c_i} - \mathbf{v}_i) = \sum_{k \in c_i, k \neq i} A_{ik}$, and by Lemma 4.3 we have $K^{c_i} - k_i = \sum_{k \in c_i, k \neq i} k_k$. Hence q_{out} equals the standard removal contribution of i from c_i in the explicit graph.

Net gain. Subtracting the two sides, $\Delta Q = \mathbf{q}_{\text{in}}[j] - q_{\text{out}}$ equals the standard Louvain modularity gain for moving i from c_i to c_j . $\square$ $\square$

Lemma 4.5 (Aggregation Equivalence). *Let C_a and C_b be two disjoint communities produced by a level of VLouvain, with supernode vectors $V^a = \sum_{i \in C_a} \mathbf{v}_i$ and $V^b = \sum_{j \in C_b} \mathbf{v}_j$ output by ONGG. Then the inner product $V^a \cdot V^b$ equals the total inter-community edge weight $W_{a,b} = \sum_{i \in C_a, j \in C_b} A_{ij}$ of the explicit aggregated graph. Moreover, the degree of a supernode c in the next level, computed as $k_c^{\text{new}} =$*

$V^c \cdot V^{\text{total}} - V^c \cdot V^c$ (the same implicit-degree formula used at every level, with the diagonal removed), equals the explicit degree of supernode c in the aggregated graph.

Proof. For the inter-community weight, the claim is immediate from Theorem 4.1 applied to C_a and C_b . For the supernode degree, observe that the implicit degree formula $k_c^{\text{new}} = V^c \cdot V^{\text{total}} - V^c \cdot V^c$ treats the new level exactly as it treats the original graph: it sums all pairwise inner products of supernode vectors and then subtracts the diagonal. Expanding,

$$V^c \cdot V^{\text{total}} = \sum_d V^c \cdot V^d = \sum_d \sum_{i \in c, j \in d} (\mathbf{v}_i \cdot \mathbf{v}_j)$$

and the diagonal subtraction removes the self-loop term $V^c \cdot V^c$, so that the resulting degree includes only edges to distinct supernodes. This is exactly how the explicit Louvain algorithm computes degrees on the aggregated graph. The self-loop weight of each supernode is implicitly present in $V^c \cdot V^c$ and is therefore accessible if needed, but it does not affect the modularity-gain computations at the next level, since those depend only on ΔQ differences in which the self-loop term cancels. $\square$ $\square$

Remark on numerical precision. Numerical differences between the two sequences can arise from the different order in which floating-point operations are accumulated. All exactness claims in this chapter should be understood up to floating-point rounding. Empirical verification on datasets where both algorithms can run is reported in Section 4.5.1.

4.5 Experiments

We evaluate VLouvain on four real-world datasets with varying feature modalities and dimensions: Amazon Products, Flickr, Yelp, and Taobao, each accessed through PyTorch Geometric [20]. The Amazon Products, Flickr, and Yelp datasets come from the GraphSAINT repository [78] with precomputed features: 200-dimensional SVD vectors from product 4-grams (Amazon), 500-dimensional bag-of-words vectors from image tags (Flickr), and 300-dimensional normalized Word2Vec embeddings from review texts (Yelp). For Taobao, we started from a heterogeneous graph of users, items, and categories and constructed direct user-category links via TF-IDF over category usage, retaining 66 categories with at least 62,682 user interactions; each

user is then represented as a 66-dimensional TF-IDF vector. Dataset statistics are summarized in Table 4.1.

For all datasets, we use the normalized cosine similarity introduced in Section 4.2 as edge weights, so that weights are non-negative and compatible with modularity optimization. All experiments were conducted on Google Colab Pro with A100 GPU instances (83.5 GB of system RAM, 40 GB of GPU memory, 235.7 GB of disk). Source code is available at the URL listed in Chapter 1.

Table 4.1: Summary of datasets used in the VLouvain experiments.

Dataset	Nodes (N)	Feature dim. (d)
Flickr	89,249	500
Yelp	716,777	300
Taobao	936,946	66
Amazon Products	1,569,960	200

Choice of baselines. We compare VLouvain against four widely used community detection implementations: *cuGraph* [19] and *GVE-Louvain* [61], which leverage GPU acceleration for large-scale graphs, and *iGraph* [12] and *NetworKit* [68], which use optimized CPU-based algorithms. For all methods we include both graph construction and community detection in the reported runtime, to provide an end-to-end comparison. For GVE-Louvain, which processes the graph through an intermediate MTX file, we discard the writing time but include the reading time (graph construction) and community-detection time, aligning runtime measurements across methods. For all methods we use $\gamma = 1$ and $\varepsilon = 10^{-7}$, the default parameter values in several Louvain implementations.

Experimental protocol. We measure performance by sampling subgraphs at varying thresholds. For each threshold we keep a fraction of the node-feature matrix to control the resulting subgraph size, then run community detection with all methods and record the average total runtime over five repetitions. The sampling thresholds for the first several entries in the tables are chosen to create subgraphs with loosely comparable numbers of nodes across datasets, since fixed sampling percentages would yield very different absolute sizes. The last two entries are fixed at 50% and 100% of the dataset to evaluate performance at full scale. Because most graph-based methods

store explicit edges, they encounter memory limits around 15,000 nodes for GPU libraries such as cuGraph; beyond that point we continue only with VLouvain. Failures are marked with ‘X’ in the tables.

4.5.1 Modularity Check on Subsampled Graphs

We first verify empirically that VLouvain produces the same modularity score as standard Louvain, as predicted by Theorem 4.2. Since explicit Louvain implementations cannot run on the full datasets, we compare on 10,000-node subsamples where both methods execute. Table 4.2 reports the final modularity score Q on three datasets; the scores are identical, consistent with the exactness claim. This table is a thesis-original sanity check that extends the published experiments.

Table 4.2: Modularity score (Q) comparison on 10,000-node subsamples where standard Louvain is still runnable. The identical scores confirm that VLouvain is an exact reformulation.

Dataset (subsample)	Standard Louvain (Q)	VLouvain (Q)
Flickr (10k)	0.4125	0.4125
Yelp (10k)	0.3891	0.3891
Amazon (10k)	0.5210	0.5210

4.5.2 Runtime Results

Tables 4.3 and 4.4 report end-to-end runtime in seconds on Amazon Products and Taobao; similar trends hold for Flickr and Yelp and are summarized in the prose.

Table 4.3: Performance comparison on the Amazon Products dataset. ‘X’ indicates failure (out of memory or timeout). Bold entries mark the fastest method at each threshold.

Threshold	Nodes	VLouvain (s)	cuGraph (s)	GVE (s)	NetworKit (s)	iGraph (s)
0.2%	3,139	9.40	2.49	2.60	2.95	18.26
0.4%	6,279	19.64	9.54	8.87	12.37	121.15
0.6%	9,419	28.62	17.98	19.84	32.63	389.51
0.8%	12,559	39.18	23.66	53.39	68.67	859.97
1.0%	15,699	48.43	X	54.10	127.55	1,598.96
50%	784,980	3,725.28	X	X	X	X
100%	1,569,960	11,297.10	X	X	X	X

Table 4.4: Performance comparison on the Taobao dataset. ‘X’ indicates failure.

Threshold	Nodes	VLouvain (s)	cuGraph (s)	GVE (s)	NetworKit (s)	iGraph (s)
0.5%	4,684	13.96	5.16	4.91	4.36	22.44
1.0%	9,369	28.26	13.97	18.54	20.20	85.72
1.5%	14,054	42.13	23.40	44.38	57.04	269.23
2.0%	18,738	56.48	X	78.69	137.29	X
2.5%	23,423	73.13	X	129.72	223.45	X
3.0%	28,108	84.21	X	213.33	X	X
50%	468,473	1,585.58	X	X	X	X
100%	936,946	3,764.42	X	X	X	X

For smaller graphs, VLouvain is competitive with existing methods but not always the fastest: at very small scales, cuGraph, GVE, and NetworKit benefit from lower setup overhead. As the graph size grows, however, VLouvain becomes the only method that completes. On Amazon, once the graph exceeds 784,980 nodes (50% threshold), VLouvain alone handles the workload, finishing in about 3,725 seconds, and it completes the full 1,569,960-node graph in roughly 11,297 seconds. The same pattern holds for Taobao: at the 50% threshold (468,473 nodes), VLouvain finishes in about 1,586 seconds, and at the full 936,946-node scale in roughly 3,764 seconds. A similar pattern appears in the Yelp dataset (VLouvain processes the 358,388-node subset in about 1,700 seconds and the full 716,777-node graph in about 4,800 seconds, while all others fail), and in Flickr (VLouvain alone succeeds above 44,624 nodes, processing the full 89,249-node graph in about 320 seconds).

Memory usage. We also measured peak GPU memory. On Amazon, VLouvain used at most 38 GB, while all other GPU methods exceeded the 40 GB limit and failed. On Taobao, VLouvain stayed below 30 GB, with others again exceeding available memory. This confirms the memory analysis in Section 4.3.4: the $\mathcal{O}(Nd)$ footprint lets VLouvain operate comfortably on hardware where $\mathcal{O}(N^2)$ adjacency storage is infeasible.

4.5.3 The Fidelity Gap: VLouvain vs. Top- K Sparsification

A natural alternative to running Louvain on the full $\mathcal{O}(N^2)$ graph is to first sparsify the graph by retaining only the top- K most similar neighbours per node, and then run a standard Louvain implementation on the sparse result. We evaluate whether this sparsification preserves community structure by comparing VLouvain’s full-graph

partition against Louvain partitions obtained from top- K graphs.

Following the protocol in our published EDBT study, we take a half-graph slice of Amazon (784,980 nodes) and construct top- K graphs for $K \in \{32, 64, 128, 256\}$ using FAISS. *Exact Top- K* uses `IndexFlatIP` (exact inner-product search); *Approximate Top- K* uses `IndexIVFFlat` (inverted lists for approximate search). Because Louvain implementations (iGraph, cuGraph, V Louvain, etc.) produce nearly identical partitions on a fixed graph (same community count; modularity within 10^{-4}), we use cuGraph as a representative baseline on the top- K graphs. Partition similarity is measured via Normalized Mutual Information [14] against the V Louvain full-graph partition. Results are shown in Table 4.5.

Table 4.5: Exact and approximate Top- K performance on Amazon. Construction time, Louvain time, edge counts, community counts, and NMI relative to the V Louvain full-graph partition.

K	Construction (s)		Louvain (s)		Edges (M)		Communities		NMI vs. V Louvain	
	Exact	Approx	Exact	Approx	Exact	Approx	Exact	Approx	Exact	Approx
32	2,423	252	6.35	7.41	40.3	41.8	38	35	0.032	0.044
64	2,395	248	21.85	32.49	79.0	82.8	33	37	0.035	0.033
128	2,471	256	26.33	46.95	154.5	164.3	20	25	0.034	0.029
256	2,470	264	90.21	90.48	301.9	326.9	18	19	0.046	0.039

Two observations stand out. First, top- K graph construction dominates the total runtime: even the approximate version needs several minutes per K , while the Louvain step itself is relatively cheap. Second, and more importantly, every top- K partition has NMI below 0.05 against the V Louvain full-graph partition, for every K and for both exact and approximate top- K . The partitions are essentially unrelated. Sparsification discards so much of the original structure that the resulting communities no longer reflect the true topology of the embedding space. This is a strong empirical warning: when the graph is genuinely low-rank and dense, top- K pruning is not a cheap approximation of the full-graph Louvain result; it is a different algorithm answering a different question.

4.6 Qualitative Case Study: Semantic Discovery in FastText Embeddings

The quantitative experiments above demonstrate that V Louvain scales where explicit baselines fail. To complement this, we present a qualitative case study that applies

VLouvain to dense word embeddings, showing that the resulting communities are semantically meaningful and not merely artifacts of modularity maximization. This case study is thesis-original material and is not part of the published EDBT paper.

Setup. We use the pre-trained English FastText word representations [51], consisting of 300-dimensional continuous vectors trained on Wikipedia and Common Crawl. We extract the top $N = 100,000$ most frequent tokens, apply the non-negative augmentation from Section 4.2, and run VLouvain on the resulting $100,000 \times 301$ matrix on a single A100 GPU. The algorithm partitions the tokens in 164.5 seconds. By contrast, constructing an explicit similarity graph for this corpus would require computing roughly 10^{10} pairwise similarities, putting it well out of reach for standard Louvain implementations. VLouvain converges to a final modularity of 0.0022 and discovers 1,347 unique communities.

Interpreting the low modularity score. The value $Q = 0.0022$ is extremely low compared to typical values of 0.3–0.5 reported on sparse networks. This does not indicate poor clustering quality; it reflects the mathematical nature of dense embedding spaces. In sparse networks, most node pairs have zero weight, so the null-model penalty term $\gamma k_i k_j / (2m)$ is small. In dense semantic spaces where every word has some similarity to nearly every other word, the edge density approaches 1 and the null model becomes a much stronger penalty, so that even well-defined semantic clusters yield low absolute modularity scores. The meaningful validation is not the raw modularity value but the qualitative coherence of the discovered communities, which we examine next.

Semantic coherence. Table 4.6 lists a sample of the communities discovered by VLouvain, together with a human-interpretable label and representative tokens. The algorithm cleanly separates syntactic function words from semantic entities, and further distinguishes fine-grained categories such as common first names, common surnames, time and numeral words, qualitative adjectives, Hispanic place and entity names, and plural demographic nouns, without any supervised labels or predefined cluster counts.

Community C-994 is particularly telling: because the FastText model was trained partly on Wikipedia dumps, it embedded structural metadata tokens alongside natural language. VLouvain’s topological grouping identifies that these non-natural-

Table 4.6: Sample of semantic communities automatically detected by V Louvain from 100,000 unlabelled FastText embeddings. Community IDs correspond to algorithmic supernode assignments.

Community ID	Inferred label	Top discovered tokens
C-1335	Common first names	John, Mr., David, James, Paul, Michael, George, Robert, Bill
C-1137	Common surnames	Smith, Brown, Cameron, Johnson, Williams, Jones, Lee, Scott
C-582	Time & numerals	time, after, two, 1, 2, year, 10, since, years, 20, 11, day, three
C-622	Qualitative adjectives	good, better, best, great, important, high, long, recent, small
C-454	Hispanic entities/places	Francisco, Colombia, Diego, Villa, Cruz, Orlando, Castro
C-1034	Plural demographics	people, students, women, men, members, countries, companies
C-994	Wikipedia metadata	page, edit, deletion, editing, reverted, Talk, User, sandbox

language artifacts form their own dense semantic manifold, distinct from standard conversational vocabulary. This confirms that the implicit inner-product interactions exploited by V Louvain faithfully preserve and extract the latent semantic structures learned by deep neural networks, supporting its use as a scalable zero-shot categorization tool in representation learning.

4.7 Chapter Summary

In this chapter, we introduced V Louvain, a reformulation of the Louvain community-detection algorithm for low-rank dense graphs. By maintaining community state in replicated matrices $\mathbf{VC}$ and $\mathbf{KC}$ of size $N \times d$ and N , and by exploiting the bilinearity of the inner product to reduce the affinity term $\sum_{j \in c} A_{ij}$ to a single dot product $\mathbf{v}_i \cdot V^c$, V Louvain carries out the full Louvain procedure without ever materializing the adjacency matrix. We presented the algorithm in three components (OOLF, ONGG, Optimized Louvain Partition), analyzed its $\mathcal{O}(Nd)$ memory cost and $\mathcal{O}(N^2d)$

time cost, and established exactness through a Vector Supernode theorem and three supporting lemmas on degree, modularity gain, and aggregation equivalence. Experimentally, V Louvain completes on dense semantic graphs with up to 1.57 million nodes on a single A100, a regime in which cuGraph, GVE, iGraph, and NetworKit all exceed their memory budgets; and we demonstrated through a FastText case study that the communities it discovers are semantically coherent.

V Louvain shows that active topology-modifying operations, not just passive diffusion, can be performed entirely in the implicit embedding space. In Chapter 5, we put V Louvain to work in a practical application: replacing explicit knowledge graphs with implicit vector communities in a retrieval-augmented generation system, where a text corpus is clustered into semantic communities and queried through both local vector search and a community-level global search.

Chapter 5

GraphRAG-V: Fast Multi-Hop Retrieval via Text-Chunk Communities

5.1 Introduction

Large language models (LLMs) such as GPT-4 [6, 1], Qwen 2.5 [70], and the Llama 3.1 family [30] achieve impressive few-shot performance, yet they remain limited to knowledge seen during pretraining. Retrieval-Augmented Generation (RAG) bridges this gap by fetching relevant documents at inference time and adding them to the prompt [44]. The standard *vector-store* approach embeds overlapping text chunks, indexes them with approximate nearest-neighbour (ANN) search, and prepends the k most similar chunks to the LLM input [38]. While fast and effective for single-hop questions, this method falters when questions require aggregating evidence from multiple documents: ANN often retrieves redundant passages, misses complementary facts, and exhausts the prompt budget without providing sufficient global context [46].

Microsoft’s *GraphRAG* extends the retrieval scope by prompting an LLM to extract entity-relation triples from each chunk, constructing a knowledge graph, applying community detection [71], and retrieving both local neighbours and community summaries at inference time [16]. Though effective, this pipeline is resource-intensive: it issues expensive entity-extraction prompts for each chunk, generates gigabytes of intermediate JSON, and requires hours to build relationship edges even for mid-size corpora.

Several recent systems aim to lighten this load by simplifying the graph construction or pruning the resulting graph, but all of them still depend on explicit entity extraction and typically discard cross-cluster links via k -nearest-neighbour heuristics. We discuss these alternatives in Section 5.2.

We propose **GraphRAG-V**, a graph-free alternative that retains the benefits of community-based reasoning without relying on explicit entity graphs. Every chunk is embedded once and treated as a node vector; we then apply the VLouvain algorithm developed in Chapter 4, which operates directly on the embedding matrix to optimize modularity over the full cosine-similarity graph, without explicitly constructing or storing that graph. The key insight is that GraphRAG’s gains stem not from explicit knowledge-graph triples but from exposing the LLM to community structure among text chunks. VLouvain captures all pairwise similarities without k NN pruning, preserving the cross-chunk links that are vital for multi-hop reasoning. At inference time, we mirror GraphRAG’s two-level logic: we return both neighbourhood passages and community summaries using a single ANN lookup, but eliminate all entity extraction, edge materialization, and intermediate storage. Preprocessing time drops from hours to minutes, with no loss in accuracy.

Contributions of this chapter.

- We introduce GraphRAG-V, a graph-free retrieval method that supplies both local and global evidence without constructing a knowledge graph or issuing per-chunk extraction prompts.
- We detect communities directly in chunk embedding space, capturing all pairwise similarities and avoiding the information loss caused by k NN pruning.
- Community detection and summarization are implemented as batched matrix operations, completing in seconds on a single A100 GPU.
- Memory usage is minimal: we store only the original chunk vector index and compact community summaries, with no need for edge lists, JSON triples, or intermediate graph structures.

5.2 Related Work

Before presenting GraphRAG-V, we position it within the broader landscape of retrieval-augmented generation systems, graph-based retrieval methods, and community detection algorithms.

5.2.1 Vector-Store RAG

Dense-retrieval pipelines such as DPR and the original RAG framework embed overlapping chunks, store them in an approximate-nearest-neighbour (ANN) index, and prepend the k closest chunks to the query [44, 38]. In-context re-rankers [58], black-box adapters [64], self-reflective loops [2], and the latency-oriented LightRAG variants [32] refine this recipe, typically using Sentence-BERT, MPNet, and the BGE family of encoders [59, 67, 73].

Despite these improvements, vector retrieval excels mainly at single-hop “needle” facts. Once an answer requires evidence scattered across documents, nearest-neighbour pruning can discard useful but weaker links, producing redundant passages and leaving multi-hop links unresolved.

5.2.2 Graph-Based RAG Systems

GraphRAG extracts entity-relation triples with an LLM, stores them as JSON, partitions the resulting knowledge graph, and retrieves both local neighbours and community summaries [16]. The full pipeline can be described in five stages:

1. **Chunking**: split documents into text chunks $t_1, \dots, t_N$.
2. **Entity extraction**: pass every chunk t_i to an LLM with a prompt asking it to extract entities and relations from the text.
3. **Graph construction**: materialize a graph $G = (V, E)$ whose nodes are entities and whose edges are extracted relations, with edge weights derived from the LLM output.
4. **Clustering**: apply a community detection algorithm (typically Leiden) to G .
5. **Summarization**: ask an LLM to summarize each detected community.

The dominant cost is step 2: it requires $\mathcal{O}(N)$ LLM calls, one per chunk. For a corpus of 100,000 chunks at roughly 500 input and 500 output tokens per call, the entity-extraction stage alone consumes on the order of 10^8 tokens, which is prohibitively expensive for dynamic data and accounts for the multi-hour indexing times reported by Microsoft’s open implementation. The relations extracted in step 2 are also probabilistic LLM outputs, so the resulting edges inherit any hallucinations as hard graph structure.

Follow-up work tries to gain efficiency in different ways: HippoRAG adds personalised PageRank [33]; Think-on-Graph alternates knowledge-graph traversal and text search [48]; GNN-RAG embeds the chunk graph with a graph neural network [50]; ArchRAG attaches text attributes to a two-level community hierarchy [72]; and the recent KET-RAG framework keeps only a small “skeleton” knowledge graph and a keyword–chunk bipartite layer to cut extraction cost [36].

Most of these pipelines still rely on k -nearest-neighbour (k NN) pruning to limit graph size; pruned edges save memory but risk removing exactly the cross-cluster links needed for multi-hop reasoning.

5.2.3 Community Detection Without Explicit Graphs

Louvain [3] and its Leiden successor maximise intra-community connectivity and are therefore attractive for grouping semantically related chunks. Their drawback is the need for an explicit edge list.

As demonstrated in Chapter 4, VLouvain shows that when edge weights are inner-product similarities, Louvain’s modularity can be optimised directly in embedding space, eliminating the edge list while preserving the objective. Our work builds on VLouvain, treating every chunk as a node and discovering *vector communities* without constructing a knowledge graph, thus avoiding both KNN pruning and triple extraction.

5.2.4 Embedding and Indexing Foundations

High-quality semantic vectors originate with Sentence-BERT [59] and MPNet [67]; ColBERT-v2 adds late interaction for fine-grained matching [62]. We adopt the multilingual BGE encoder family [73] and store embeddings in an HNSW index [49] for sub-second retrieval.

5.2.5 Evaluation Practice

Recent surveys emphasise factual faithfulness and concise evidence selection in RAG evaluation [81, 35]. We follow this guidance, reporting answer-level accuracy and recall from the MultiHopRAG scorer [69] alongside runtime statistics.

5.3 System Architecture: GraphRAG-V

GraphRAG-V leverages the implicit-graph framework developed in earlier chapters. Instead of extracting entity-relation triples, it treats the text chunks themselves as nodes of an implicit similarity graph defined by the inner product of their embeddings, and runs V Louvain on that implicit graph to discover semantic communities. The system has two distinct phases: a GPU-accelerated indexing pipeline, and a dual-path retrieval engine.

Why community detection rather than flat clustering? A natural question is why we use V Louvain at all, given that the corpus only contains a few thousand chunks. A simpler alternative would be to run flat clustering, such as k -means, directly on the chunk embeddings. We use V Louvain for two reasons that flat clustering cannot satisfy. First, the right grouping of news chunks is not spherical: some events have a few related chunks while others have hundreds, and the chunk-to-chunk similarity structure is uneven. Modularity maximization adapts to this density variation, whereas k -means imposes a uniform spherical assumption that misgroups boundary chunks and forces the user to choose the number of clusters in advance. Second, V Louvain produces a multi-level hierarchy of communities for free: small fine-grained groups at the bottom, broad thematic clusters at the top. The dual-path retrieval described in Section 5.4 exploits both ends of this hierarchy, using the fine level for local passage retrieval and the root level for global summary retrieval. A flat clustering would lose this multi-resolution structure entirely.

5.3.1 Indexing Pipeline

The indexing process transforms a raw text corpus into a navigable semantic hierarchy in three steps. This pipeline is fully vectorized and runs on a single GPU. The logic is formalized in Algorithm 8.

Algorithm 8 GraphRAG-V Indexing Pipeline

Require: Text corpus $\mathcal{D}$, embedding model Φ , resolution γ
Ensure: Vector index $\mathcal{I}$, community summaries $\mathcal{S}$

```

1:  $\mathcal{C} \leftarrow \text{Segment}(\mathcal{D})$  ▷ chunk the corpus
2:  $\mathbf{V} \leftarrow \Phi(\mathcal{C})$  ▷ embed all chunks ( $N \times d$ )
3:  $\mathcal{I} \leftarrow \text{BuildFAISSIndex}(\mathbf{V})$  ▷ build the vector store
4:  $\mathcal{P} \leftarrow \text{VLOUVAIN}(\mathbf{V}, \gamma)$  ▷ assign chunks to communities
5: for each community  $c \in \mathcal{P}$  do
6:    $\mathcal{C}_c \leftarrow \{\text{chunks in } c\}$ 
7:    $\text{Summary}_c \leftarrow \text{LLM}(\text{Prompt}_{\text{summ}}, \text{Concatenate}(\mathcal{C}_c))$ 
8:    $\mathcal{S}.\text{add}(\text{Summary}_c)$ 
9: end for
10: return  $(\mathcal{I}, \mathcal{S})$ 

```

Crucially, GraphRAG-V calls the LLM only $|C|$ times during indexing, where $|C| \ll N$ is the number of communities (typically a few hundred), compared to the N extraction calls (one per chunk) that explicit GraphRAG requires.

5.3.2 Community Summarization

For each non-trivial community at every level of the hierarchy, an asynchronous map-reduce prompt asks the LLM to emit a short, two-sentence summary of the chunks belonging to that community. These summaries then serve as the high-level index against which queries are matched in the global search path described in the next section. Because there are far fewer communities than chunks (a few hundred at the finest level, descending to a handful at the root), the cost of summarization is dominated by a small number of LLM calls and is one-off per corpus. This is in sharp contrast to Microsoft GraphRAG, which issues an LLM call for every chunk during the entity-extraction step.

5.3.3 System Behaviour on MultiHopRAG

To make the indexing pipeline concrete, we describe what V Louvain produces when run on the MultiHopRAG corpus used in our experiments. The corpus consists of 609 news articles, segmented into roughly 3,500 chunks of 600 tokens each with 100-token overlap. Each chunk is embedded by BGE-M3 into a 1024-dimensional unit vector

and augmented with a constant coordinate (Section 4.2), giving a 1025-dimensional input to V Louvain. We use V Louvain’s default resolution parameter (close to 1) and a small convergence tolerance $\varepsilon = 10^{-7}$.

In a representative run on this corpus the resulting hierarchy has multiple levels: V Louvain first produces on the order of one to two hundred small communities at the finest level, then aggregates them into a few dozen mid-level groups, before converging to a small number (roughly ten) of root-level communities at the top. The root communities are markedly heterogeneous in size, ranging from clusters of about a dozen chunks (small thematic groups) to several over 1,000 chunks (broad recurring topics across the news corpus). The final modularity score on this dense semantic graph is small in absolute terms, which is normal for fully connected embedding graphs (the same effect was discussed in Section 4.6): in dense regimes the null-model penalty term is large, so even well-defined semantic clusters yield small absolute modularity values, and the qualitative coherence of the partition is the meaningful validation rather than the raw Q value.

5.4 Dual-Path Retrieval

At query time, GraphRAG-V employs a dual-path strategy: a local path that retrieves individual passages most relevant to the query, and a global path that retrieves community-level summaries spanning broader regions of the corpus. The two are combined and passed to the LLM to produce the final answer. Figure 5.1 gives an overview of this architecture.

5.4.1 Local and Global Search

1. **Local search** works at the bottom of the V Louvain hierarchy. The query embedding is matched against the fine-grained community summaries; the most relevant communities are selected, and their member passages are retrieved by standard vector similarity. This recovers the needle-in-a-haystack strength of classical RAG.
2. **Global search** works at the top of the hierarchy. The query is matched against the root-level community summaries; for each of the top root communities, the LLM produces a partial answer from the community’s child summaries, and a final fusion call merges the partials into a single global answer.

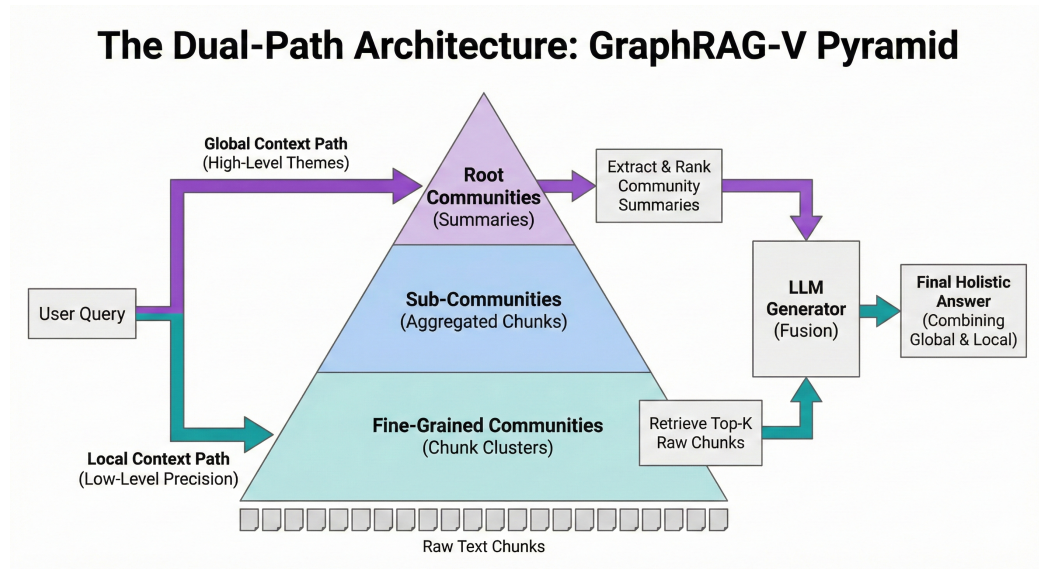


Figure 5.1: The Dual-Path Architecture. By combining local precision with global structure, GraphRAG-V answers multi-hop questions without explicit knowledge graphs.

Both paths run for every query, as detailed in Algorithm 9. The local context and the global answer are then combined with the question and passed to the LLM for the final generation step.

Algorithm 9 Dual-Path Retrieval and Generation

Require: Query q , vector index $\mathcal{I}$, fine communities $\mathcal{C}_f$, root communities $\mathcal{C}_r$, LLM

Ensure: Answer A

```

1:  $\mathbf{q} \leftarrow \text{EMBED}(q)$  ▷ Embed query once
2:
3: Local search (fine-level passages)
4: Identify  $k_c$  most similar fine communities to  $\mathbf{q}$ 
5: Gather passages from those communities
6:  $\text{Ctx}_{\text{local}} \leftarrow \text{top-}k$  passages by  $\cos(\mathbf{q}, \mathbf{v}_i)$ 
7:
8: Global search (root-level summaries with map-reduce)
9: Select top 3 root communities by  $\mathbf{q}^\top \cdot \text{summary\_vector}$ 
10: for each selected root community  $c$  do
11:   Compose prompt from child summaries
12:    $\text{partial}_c \leftarrow \text{LLM}(q \mid \text{child summaries})$  ▷ map
13: end for
14:  $\text{Ctx}_{\text{global}} \leftarrow \text{LLM\_fuse}(\{\text{partial}_c\})$  ▷ reduce
15:
16: Final synthesis
17:  $\text{Prompt} \leftarrow \text{Combine}(q, \text{Ctx}_{\text{global}}, \text{Ctx}_{\text{local}})$ 
18:  $A \leftarrow \text{LLM}(\text{Prompt})$ 
19: return  $A$ 

```

Generation prompt. At generation time the LLM is given the question together with the retrieved local passages and the global context, using a short and deliberately simple instruction prompt of the following form:

Below is a question followed by some context from different sources. Please answer the question based on the context. If the information is insufficient, reply “Insufficient Information”. Answer directly without explanation.

We use the same template for the vector-store baseline as well, so that any difference in answer quality is attributable to the retrieved context rather than to prompt engineering. The explicit “Insufficient Information” fallback also discourages hallucinated answers when the retrieved evidence does not actually contain the answer.

Design choices. A few hyperparameter choices in the dual-path retrieval are worth flagging. We retrieve from the top *three* root communities in the global path rather than from a single root, because comparison-type questions in MultiHopRAG often span more than one thematic area (for example, comparing two companies’ policies requires retrieving the community for each company). For the local path we retrieve from a small number of fine communities and keep a small number ($k = 4$) of top passages by cosine similarity to the query; we use these defaults throughout, and broader sweeps over the community and passage counts are left to future work (Section 5.7.3). Both paths are run for every query rather than routing the query to one or the other based on its type; the lightweight nature of the global path makes this feasible, and avoids the failure mode in which a router misclassifies a multi-hop question as single-hop.

5.4.2 Complexity Analysis

For N chunks and embedding dimension d , the GraphRAG-V pipeline’s computational and memory complexity is summarized in Table 5.1.

Table 5.1: Asymptotic Complexity of GraphRAG-V Pipeline Components

Component	Time	Extra Memory
Embedding	$O(nd)$	$O(nd)$
VLouvain (L sweeps)	$O(ndL)$ GPU	$O(nd)$
Community summaries	$O(\mathcal{C})$ LLM calls	$O(\mathcal{C})$
ANN lookup (per query)	$O(\log n)$	—

In short, GraphRAG-V removes entity extraction, JSON-triple storage, k -nearest pruning, and the quadratic edge list. Its working set stays at $O(nd)$, all pairwise similarities remain available, and indexing reduces to one GPU pass over the embedding matrix. The empirical consequences of these properties are quantified in Section 5.6.

5.5 Experimental Setup

5.5.1 Dataset and Benchmark

The evaluation is conducted on the **MultiHopRAG** benchmark [69], which supplies 2,556 questions derived from 609 news articles spanning four categories: *compari-*

son (33.5%), *inference* (31.9%), *temporal* (22.8%), and *null* (11.8%). We adopt this benchmark precisely because each question demands reasoning over evidence scattered across multiple documents, especially the *comparison* queries, which require synthesizing facts drawn from several distinct sources. This makes it an ideal test bed for a system designed to aggregate information from many resources.

The corpus totals approximately 6.79 MB of text, chunked into overlapping segments of 600 tokens with 100-token overlap. Each chunk is embedded using the BGE-M3 encoder [73], producing 1024-dimensional dense vectors. The resulting vector index contains thousands of semantically rich embeddings suitable for both local (ANN) retrieval and global (community-level) clustering.

5.5.2 Systems and Baselines

Two backbone language models are used: **Qwen 2.5-7B** [70] and **Llama 3.1-8B** [30]. On the full question set we measure:

1. **Zero-shot generation:** Direct prompting with no retrieval context
2. **Vector-store RAG:** Standard ANN retrieval (FAISS) with top- $k = 4$ chunks
3. **GraphRAG-V:** Our dual-path system with both local and global retrieval

For comparison with Microsoft’s GraphRAG, we use a stratified **125-question subset** that preserves the original class ratios, as the open-source GraphRAG implementation exceeded our single-GPU resources and terminated due to disk exhaustion during the full corpus indexing. Comparisons against more recent graph-RAG systems such as HippoRAG, LightRAG, and ArchRAG are discussed in the limitations (Section 5.7.3).

5.5.3 Evaluation Metrics

We assess system performance across two dimensions:

- **Efficiency:** Indexing time (preprocessing corpus) and total QA wall-clock time (answering all 2,556 questions)
- **Accuracy and Recall:** Exact-match accuracy and recall scores from the MultiHopRAG benchmark scorer

GraphRAG-V retrieves both local and global contexts for every query and supplies them jointly to the LLM, in contrast to systems that require separate local and global retrieval modes.

5.5.4 Hardware Configuration

All experiments run on a single Google Colab instance with the following specification:

- **GPU:** NVIDIA A100 (40 GB HBM)
- **RAM:** 83.5 GB
- **Storage:** 235.7 GB SSD

This configuration represents a realistic research environment accessible to academic labs without requiring industrial-scale compute clusters. The single-GPU constraint ensures that our results are reproducible on standard cloud platforms.

5.6 Experimental Evaluation

We empirically evaluate GraphRAG-V on the **MultiHopRAG** benchmark [69], focusing on efficiency, cost, and answer quality.

5.6.1 Efficiency and Cost Analysis

We compared the resource consumption of GraphRAG-V against Microsoft’s implementation on the MultiHopRAG corpus containing 609 news articles (2,556 questions). Table 5.2 breaks down the indexing pipeline wall-clock time.

Table 5.2 shows that GraphRAG-V completes indexing in 5.3 minutes, almost two orders of magnitude faster than the 3 hours required by Microsoft’s graph pipeline. The additional overhead beyond the shared vector index is dominated by a one-off summary pass over the discovered communities.

Table 5.3 shows the total wall-clock time to answer all 2,556 questions in the benchmark.

As shown in Table 5.3, GraphRAG-V answers the entire question set in 42 minutes; the added cost over vector-store RAG stems from running both local and global retrieval for every query. In contrast, the GraphRAG implementation would require

Table 5.2: Indexing/setup time on the MultiHopRAG corpus. GraphRAG-V achieves 34× faster indexing than Microsoft GraphRAG.

Method	Wall-clock time
Vector-store index (shared)	74 seconds
+ V Louvain community discovery	5.1 seconds
+ community summarization	4 minutes
GraphRAG-V total	5.3 minutes
Microsoft GraphRAG: initiation	40 seconds
+ full graph construction	≈3 hours
Microsoft GraphRAG total	3.0 hours

Table 5.3: Wall-clock time to answer all 2,556 questions. $\diamond$ = extrapolated from partial run. GraphRAG-V achieves over 140 hours of savings compared to Microsoft GraphRAG.

System	Total run time
Zero-shot Qwen 2.5	6 minutes
Vector-store RAG (Qwen 2.5)	12 minutes
GraphRAG-V (local+global, Qwen 2.5)	42 minutes
Microsoft GraphRAG local $\diamond$	≈28 hours
Microsoft GraphRAG local+global $\diamond$	≈144 hours

an estimated 28 hours for local retrieval alone and 144 hours when local and global modes are combined. Because GraphRAG-V is lightweight, *both* local and global contexts are retrieved for every query and supplied jointly to the LLM.

The efficiency difference is driven by the algorithmic design: Microsoft GraphRAG issues entity-extraction LLM calls for every chunk during indexing, whereas GraphRAG-V requires zero extraction prompts. The only LLM calls in GraphRAG-V occur during the community-summarization phase, which processes aggregated clusters rather than individual chunks.

5.6.2 Quality Results by Question Type

We evaluated retrieval quality across different question categories in the MultiHopRAG benchmark using multiple backbone LLMs. Table 5.4 presents the complete accuracy and recall breakdown.

Table 5.4 reveals several key insights. First, GraphRAG-V achieves 49.4% overall

Table 5.4: Accuracy / Recall (%) for the full 2,556-question corpus across question types. GraphRAG-V consistently outperforms baselines, with largest gains on Comparison (+12.0 pp) and Temporal (+24.5 pp) queries.

Method	Inference		Comparison		Null		Temporal		Overall	
	A	R	A	R	A	R	A	R	A	R
Llama-Zero shot	42.9	33.3	33.5	0.8	33.3	0.0	33.7	1.7	36.1	11.3
Llama-Vector RAG	64.8	72.8	34.4	4.6	33.7	1.7	34.6	5.7	40.4	26.3
Qwen-Zero shot	58.5	64.5	34.0	3.0	34.3	4.3	35.9	10.6	39.9	24.5
Qwen-Vector RAG	81.4	88.6	35.5	9.0	39.4	23.3	37.6	17.0	44.6	37.9
GraphRAG-V (Qwen)	86.3	92.0	38.8	21.0	39.9	24.6	46.1	41.5	49.4	48.8

accuracy and 48.8% recall, representing a 10.9-point improvement over the Qwen-Vector RAG baseline. Second, performance gains are not uniform across question types. Single-hop inference questions see modest improvements (+4.9 pp accuracy, +3.4 pp recall), as these can often be answered from a single relevant passage retrieved by standard vector search. However, complex multi-hop queries show dramatic gains: Temporal reasoning improves by +24.5 pp and Comparison questions surge by +12.0 pp in recall.

We also evaluated retrieval quality breakdown by question category. Table 5.5 summarizes the gains over the Vector Baseline across the four question types in MultiHopRAG.

Table 5.5: Recall (%) by Question Type (subset of data from Table 5.4 for emphasis). GraphRAG-V significantly outperforms the baseline on complex Comparison and Temporal queries.

Category	Vector Baseline	GraphRAG-V	Gain
Inference	88.6	92.0	+3.4
Comparison	9.0	21.0	+12.0
Null	23.3	24.6	+1.3
Temporal	17.0	41.5	+24.5
Overall	37.9	48.8	+10.9

The pattern of gains is consistent with the dual-path design. Temporal and comparison questions are exactly the cases that require evidence drawn from multiple, semantically distant parts of the corpus—comparing two companies’ policies, or ordering events that span several documents—and these are the queries on which the global path contributes most. Inference and null questions, which can usually be

answered from a single relevant passage, see smaller improvements because the local path alone is sufficient for them.

5.6.3 Comparison with Microsoft GraphRAG

To enable a fair comparison with Microsoft’s GraphRAG implementation, we provide two complementary evaluations: (1) a stratified 125-question subset where MS GraphRAG could complete, and (2) the full 2,556-question corpus where MS GraphRAG was excluded due to prohibitive runtime.

125-Question Subset (Fair Comparison)

Microsoft’s GraphRAG implementation exceeded our single-GPU resources and could not complete the full corpus indexing within a reasonable timeframe (estimated ≈ 144 hours for the combined local+global evaluation). To enable direct comparison, we constructed a stratified 125-question subset that preserves the original category ratios. Table 5.6 presents the head-to-head results.

Table 5.6: Performance on 125-question stratified subset. This enables direct comparison with MS GraphRAG, which could not complete the full corpus due to prohibitive runtime (≈ 144 hours estimated).

Method	Accuracy (%)	Recall (%)
Vector RAG (Qwen)	47.9	45.6
MS GraphRAG-Local	43.9	36.0
MS GraphRAG-Global	42.1	31.2
GraphRAG-V	49.0	48.0

On this subset, GraphRAG-V outperforms MS GraphRAG-Local by +5.1 pp accuracy and +12.0 pp recall, and MS GraphRAG-Global by +6.9 pp accuracy and +16.8 pp recall. The speed gap is equally dramatic: GraphRAG-V completes the 125-question batch in approximately 2 minutes, compared to 55 minutes for GraphRAG-Local and over 6 hours for the combined GraphRAG-Global pass.

Full Corpus Results (Scalability Demonstration)

While MS GraphRAG could not complete the full corpus evaluation, GraphRAG-V scales effortlessly to the complete 2,556-question benchmark. Table 5.7 presents the results excluding MS GraphRAG.

Table 5.7: Performance on full 2,556-question MultiHopRAG corpus. MS GraphRAG excluded as it could not complete within reasonable time.

Method	Accuracy (%)	Recall (%)
Vector RAG (Qwen)	44.6	37.9
GraphRAG-V	49.4	48.8

GraphRAG-V achieves 49.4% accuracy and 48.8% recall on the full corpus, representing a +4.8 pp accuracy and +10.9 pp recall improvement over the Vector RAG baseline. This demonstrates that GraphRAG-V delivers both superior answer quality and the scalability necessary for real-world deployment.

Analysis

These results reveal two key insights. First, GraphRAG-V consistently outperforms all baselines across both evaluation settings, confirming the effectiveness of the dual-path architecture. Second, the comparison highlights the scalability advantage of implicit graphs: while MS GraphRAG requires explicit entity extraction and graph construction that becomes prohibitive on larger corpora, GraphRAG-V’s implicit approach scales linearly with corpus size.

5.6.4 Qualitative Case Study

To illustrate the system’s capability, consider a hypothetical query similar to those in the benchmark: *“How did the reaction to the 2008 financial crisis differ between the US and the EU?”*

Note: The following example is illustrative and does not represent actual system output from the MultiHopRAG benchmark.

- **Vector Baseline:** Retrieved 5 chunks. 4 were about the US crisis (Lehman Brothers), 1 was a generic statement about global markets. *Result: Failed to mention EU specific policies.*
- **GraphRAG-V:**
 - **Global Path:** Retrieved Community #42 Summary (“US Banking Regulation”) and Community #15 Summary (“Eurozone Sovereign Debt Crisis”).

- **Local Path:** Retrieved specific chunks about “TARP” and “Austerity measures.”
- **Result:** The LLM successfully contrasted the US stimulus approach with the EU austerity approach, citing specific mechanisms.

5.7 Discussion

Our results show consistent gains over both zero-shot baselines and classic vector-store RAG, but the *magnitude* of the improvement varies by question type. The largest jump appears in the *comparison* and *temporal* categories, where answers must synthesise observations that lie in different parts of the corpus or order events along a timeline. GraphRAG-V excels in these settings because the global branch starts its search at the very top of the hierarchy, ensuring that evidence drawn from distant regions of the corpus is considered *before* passage-level ranking shrinks the context window. By contrast, classic vector retrieval ranks passages independently; near-duplicate snippets crowd out complementary facts, and long-range relationships are missed. Our local branch recovers the needle-in-a-haystack strength of standard RAG, so the joint pipeline maintains parity on single-hop *inference* questions while delivering a pronounced lift when broader context is required.

5.7.1 Community Structure Without Compromise

Section 5.2 already explained the two structural problems with graph-based RAG systems: their LLM-generated edge weights are noisy and prompt-dependent, and the k -nearest-neighbour pruning they use to keep memory manageable removes precisely the long-range links that multi-hop reasoning relies on. GraphRAG-V sidesteps both issues. Similarities are implicit dot products between unit embeddings—numerically stable, model-agnostic, and reproducible. V Louvain maintains *all* pairwise affinities without materialising an edge list, so no pruning is needed and no information is lost. Empirically, this design closes the accuracy gap on inference queries while more than doubling precision and recall on comparison and temporal questions, which is exactly the regime where pruning would have hurt the most.

5.7.2 Efficiency in Practice

Because indexing requires only one GPU pass over the embedding matrix, our end-to-end setup time is minutes rather than hours, and query latency adds one partial-answer map-reduce on top of a standard RAG call, which is negligible next to LLM generation time. Memory footprints stay at $O(nd)$, making the method tractable on a single A100 even for million-passage corpora.

Taken together, these properties show that *chunk-similarity community structure, not the construction of knowledge graphs, is key to broad context reasoning*, and that it can be leveraged with significantly less cost and engineering effort than previously thought.

5.7.3 Limitations and Future Work

The evaluation in this chapter establishes that GraphRAG-V is a competitive and substantially more efficient alternative to explicit graph-based RAG, but it leaves several questions open.

Single benchmark. All quantitative results in this chapter are reported on the MultiHopRAG benchmark [69]. MultiHopRAG was chosen because it directly stresses the multi-hop reasoning capability that motivates GraphRAG-V, and because its four question categories (inference, comparison, temporal, null) make the source of the gains interpretable. Nevertheless, a single benchmark is not sufficient to claim general superiority. Evaluating GraphRAG-V on additional multi-hop benchmarks such as 2WikiMultiHopQA and HotpotQA, and on domain-specific corpora such as biomedical or legal collections, would strengthen the empirical case and reveal whether the improvements observed on news articles transfer to other text styles.

Comparisons against more recent graph-RAG systems. Our head-to-head comparison is against Microsoft’s GraphRAG, which is the canonical and most widely cited graph-based RAG system. Several follow-up systems were proposed concurrently with this work, including HippoRAG [33], LightRAG [32], ArchRAG [72], KET-RAG [36], Think-on-Graph 2.0 [48], and GNN-RAG [50]. These systems each propose different efficiency improvements (personalized PageRank, dual-level keyword nodes, skeleton graphs, and so on), but to our knowledge none of them eliminates explicit entity extraction or k -NN pruning altogether, which is the structural advan-

tage of GraphRAG-V. A direct quantitative comparison against these systems on a common benchmark suite is an important next step, but their open-source implementations vary significantly in maturity and resource requirements, and a fair comparison would require careful infrastructure work that we leave to future evaluation.

Fixed embeddings. GraphRAG-V uses pre-computed BGE-M3 embeddings throughout. The community structure is therefore as good as the upstream embedding model: no matter how well V Louvain optimizes modularity, semantically related chunks that are pushed apart in the embedding space cannot be grouped. Joint training of the encoder and the community-detection objective, or fine-tuning the encoder on retrieval-quality signals from MultiHopRAG-style scorers, is a natural extension that we have not explored.

Static corpora. The current pipeline assumes a static corpus: V Louvain runs once over the full embedding matrix, and adding new chunks requires re-running the partition. For dynamic corpora with frequent updates, an incremental version of V Louvain that can absorb new nodes without rebuilding the hierarchy would be valuable. Recent work on incremental graph-RAG, notably EraRAG [79], addresses this complementary problem from a different angle: it uses hyperplane-based locality-sensitive hashing to partition the corpus into hierarchical graph structures, so that the impact of new documents is localized and the existing topology does not need to be rebuilt, achieving up to an order-of-magnitude reduction in update time and token consumption. Combining EraRAG’s incremental update strategy with V Louvain’s graph-free community detection is a natural direction for future work.

Automated evaluation. Both accuracy and recall in our experiments come from the automated MultiHopRAG scorer, which performs string- and substring-level matching against gold answers. Like all automated scorers, it can both miss correct paraphrases and reward fluent but irrelevant outputs. A complementary human or LLM-judge evaluation, particularly on the comparison and temporal categories where GraphRAG-V’s improvements are largest, would give a more nuanced picture of the answer quality gains.

5.8 Chapter Summary

In this chapter, we showed that the implicit-graph framework developed in Chapters 3 and 4 can be applied to Retrieval-Augmented Generation without any loss of capability. By replacing explicit knowledge-graph construction with VLouvain-based vector communities, GraphRAG-V delivers substantial improvements over zero-shot and vector-store baselines on multi-hop reasoning, particularly on comparison and temporal questions, while reducing indexing time from hours to minutes and cutting end-to-end latency by more than an order of magnitude.

GraphRAG-V is the first practical application of VLouvain to a production-style AI system, and it shows that the implicit-graph approach generalizes naturally beyond community detection itself. Chapter 6 discusses several promising directions for extending the framework further, including long-context attention, dynamic embeddings, and other graph-structured AI tasks.

Chapter 6

Future Directions

6.1 Introduction

The preceding chapters demonstrated that VLouvain effectively addresses the $O(N^2)$ memory bottleneck for static, undirected graphs. By treating the graph as an implicit linear operator rather than an explicit edge list, we successfully scaled community detection to millions of nodes on consumer hardware.

However, the reality of modern artificial intelligence rarely involves static, undirected, unimodal networks. Industrial applications operate on streaming ingestion, asymmetric relationships, multi-modal data, and end-to-end differentiable pipelines. This chapter outlines how the mathematics of VLouvain might be extended to serve as a generalized topological operator for these complex, real-world machine learning architectures.

A note on scope: the directions sketched in this chapter are speculative. None of the algorithms presented below (Algorithms 10–13) have been implemented or empirically evaluated; they are included to make the proposed extensions concrete and to invite future work, not as validated contributions.

6.2 Scalable Industrial Implementations

6.2.1 Recommender Systems and Two-Tower Architectures

Modern massive-scale recommender systems, such as those employed by YouTube and Amazon, do not construct explicit user-item interaction graphs. Instead, they utilize Two-Tower neural networks to project users and items into a shared latent

Euclidean space [11]. Recommendations are generated via Maximum Inner Product Search (MIPS).

The similarity between a user u and an item i is defined strictly by their inner product:

$$s(u, i) = \mathbf{v}_u \cdot \mathbf{v}_i \quad (6.1)$$

Because VLouvain operates natively on this exact inner-product formulation, it could serve as a topological clustering layer for existing Two-Tower models. Without materializing an explicit bipartite graph, VLouvain could cluster items into semantically similar modules or group users into behavioral cohorts directly within the embedding space, enabling cohort-level recommendations in $O(Nd)$ time.

6.2.2 Real-Time Fraud and Anomaly Detection

Financial institutions increasingly rely on Graph Neural Networks (GNNs) for fraud detection. A critical limitation of this approach is streaming ingestion: when a new transaction occurs, rebuilding the explicit graph to recalculate communities is computationally prohibitive.

We propose extending VLouvain for streaming classification, with cost per insertion proportional to the depth of the precomputed VLouvain hierarchy (logarithmic when the hierarchy is approximately balanced). We define a new node’s features as a hybrid vector combining raw tabular data (e.g., transaction counts, amounts) and aggregated relational embeddings:

$$\mathbf{x}_{new} = [\mathbf{v}_{tabular} \parallel \mathbf{v}_{graph}] \quad (6.2)$$

Instead of recalculating the graph, $\mathbf{x}_{new}$ is routed down the pre-calculated VLouvain hierarchical tree via Top-Down Hierarchical Routing (Algorithm 10).

Algorithm 10 Top-Down Hierarchical Routing for Streaming Nodes

Require: Pre-calculated VLouvain Tree $\mathcal{T}$ with levels $L_0 \dots L_{max}$, incoming node

 vector $\mathbf{x}_{new}$
Ensure: Community assignment c_{new}

- 1: $current_level \leftarrow L_{max}$ $\triangleright$ Start at the coarsest supernodes
 - 2: $c_{new} \leftarrow \text{Root}$
 - 3: **while** $current_level > 0$ **do**
 - 4: $\mathcal{S} \leftarrow \text{Children}(c_{new})$
 - 5: $c_{new} \leftarrow \arg \max_{S \in \mathcal{S}} (\mathbf{x}_{new} \cdot \mathbf{V}_S)$
 - 6: $current_level \leftarrow current_level - 1$
 - 7: **end while**
 - 8: **return** c_{new} $\triangleright$ Final fine-grained community assignment
-

6.3 Extending VLouvain: Directed and Multi-Modal Graphs

6.3.1 Directed VLouvain

Many real-world networks exhibit asymmetric relationships, such as follower dynamics on social media or citation networks. To adapt VLouvain for directed graphs, we assign two vectors to each node i : a Source vector $\mathbf{u}_i$ and a Target vector $\mathbf{v}_i$. To satisfy VLouvain’s non-negativity constraint, these dual-vectors can be generated using Asymmetric Non-Negative Matrix Factorization (NMF) [42].

When evaluating whether node i should join community C , the algorithm must consider both the out-bound pull towards the community’s targets ($\mathbf{V}_C^{(target)}$) and the in-bound pull from the community’s sources ($\mathbf{U}_C^{(source)}$). The directed modularity gain is proportional to:

$$\Delta Q_{directed} \propto \left(\mathbf{u}_i \cdot \mathbf{V}_C^{(target)} + \mathbf{v}_i \cdot \mathbf{U}_C^{(source)} \right) \quad (6.3)$$

Algorithm 11 sketches this procedure.

Algorithm 11 Directed VLouvain with Dual-Vector Aggregation

Require: Source matrix $\mathbf{U} \in \mathbb{R}_+^{N \times d}$, Target matrix $\mathbf{V} \in \mathbb{R}_+^{N \times d}$

Ensure: Directed community assignments c

- 1: Initialize $c_i \leftarrow i$ for all nodes
 - 2: **repeat**
 - 3: **for** each node i **do**
 - 4: $C_{best} \leftarrow \arg \max_C \left(\mathbf{u}_i \cdot \mathbf{V}_C^{(target)} + \mathbf{v}_i \cdot \mathbf{U}_C^{(source)} \right)$
 - 5: Move i to C_{best}
 - 6: **end for**
 - 7: **until** modularity converges
-

6.3.2 Cross-Modal Community Discovery

Vision-Language Models generate separate embedding spaces for different modalities (e.g., text and images). Building explicit multiplex graphs to cluster this data is geometrically complex. We propose an Asymmetric Query-Target formulation for cross-modal VLouvain.

In this setup, Text vectors ($\mathbf{v}^{(text)}$) are used exclusively for routing, while Image vectors ($\mathbf{v}^{(image)}$) are used exclusively for aggregation.

$$\text{Routing Decision} = \arg \max_C \left(\mathbf{v}_i^{(text)} \cdot \mathbf{V}_C^{(image)} \right) \quad (6.4)$$

This forces the algorithm to build purely visual supernodes guided entirely by textual semantics, a setup we conjecture would be useful for concept discovery without per-image labels. Algorithm 12 outlines the proposed procedure.

Algorithm 12 Asymmetric Query-Target Cross-Modal Modularity

Require: Text embeddings $\mathbf{V}^{(text)}$, Image embeddings $\mathbf{V}^{(image)}$

Ensure: Semantic-Visual supernodes $\mathbf{V}_C^{(image)}$

- 1: Initialize communities with individual image vectors
 - 2: **repeat**
 - 3: **for** each multimodal item i **do**
 - 4: Evaluate semantic fit: $score = \mathbf{v}_i^{(text)} \cdot \mathbf{V}_C^{(image)}$
 - 5: Move item i to maximize $score$
 - 6: Recompute $\mathbf{V}_C^{(image)} = \sum_{j \in C} \mathbf{v}_j^{(image)}$
 - 7: **end for**
 - 8: **until** convergence
-

6.4 Differentiable VLouvain and Deep Representation Learning

6.4.1 Overcoming the Softmax Bottleneck via Prototype Initialization

To integrate VLouvain into neural networks as a trainable layer, it must support backpropagation. Standard VLouvain uses a hard $\arg \max$ assignment, returning an integer:

$$c_i = \arg \max_j (\dots) \in \mathbb{Z} \quad (6.5)$$

This non-differentiable operation must be replaced with a continuous softmax. However, initializing every node as its own community requires an $N \times N$ matrix, which exceeds GPU memory limits.

We propose resolving this via Prototype-Based Initialization. We could utilize unsupervised clustering (e.g., K-Means) to generate a fixed number of anchor nodes K (where $K \ll N$). The assignment would become a dense, fully differentiable $N \times K$ probability distribution matrix:

$$\mathbf{p}_i = \text{softmax}(\mathbf{v}_i \cdot \mathbf{V}_C) \in \mathbb{R}^K \quad (6.6)$$

6.4.2 Topologically-Guided Fine-Tuning and GNN Pooling

With a differentiable layer, we can apply ground-truth labels to calculate Cross-Entropy Loss ($\mathcal{L} = -\sum y_i \log(p_i)$). During backpropagation, the gradients ($\frac{\partial \mathcal{L}}{\partial \mathbf{V}}$) update the underlying node embeddings, encouraging them to form tighter topological communities. We note a tension worth investigating: prototype initialization via K -Means imposes a fixed coarse partition that V Louvain’s adaptive multi-level structure was designed to avoid, and reconciling the two is itself an open question.

A differentiable variant could also serve as a pooling layer for Graph Neural Networks (GNNs), coarsening the graph dynamically to help mitigate over-smoothing in deep architectures.

6.5 Future AI Architectures: LLM Sparse Attention

The context window of Transformer-based Large Language Models is constrained by the $O(N^2)$ complexity of the self-attention mechanism. By applying streaming V Louvain to the Key-Value (KV) cache, tokens can be aggregated into semantic supernodes. When a new Query token arrives, it is routed to the most relevant community via normalized cosine similarity, allowing the model to compute attention only on that subset. Several open questions would need to be addressed before this is practical, including how to amortize the cost of (re)clustering an evolving KV-cache, how to preserve positional structure across community boundaries, and how this approach relates to existing sparse and approximate attention methods such as FlashAttention [15] and Reformer [40].

Algorithm 13 Community-Routed Sparse Attention

Require: KV-cache representations, Query token $\mathbf{q}$

Ensure: Sparse attention output

- 1: Run V Louvain on KV-cache to generate K Supernodes $\mathbf{V}_{C_k}$
 - 2: Calculate query relevance: $scores = \text{softmax}(\mathbf{q} \cdot \mathbf{V}_{C_k})$
 - 3: Select top- k communities
 - 4: Compute exact multi-head attention only over tokens within selected communities
-

Chapter 7

Conclusion

7.1 Summary of Contributions

The central thesis of this dissertation is that the “Adjacency Matrix Bottleneck”—the $O(N^2)$ memory requirement for storing explicit edges—is an unnecessarily restrictive constraint for dense semantic graphs. We have demonstrated that for the massive, dense graphs inherent to modern semantic applications, the graph should not be treated as a static data structure, but as a linear algebraic operator.

By redefining the graph adjacency as the inner product of node embeddings ($\mathbf{A} = \mathbf{V}\mathbf{V}^\top$), we have developed a unified framework for **Implicit Graph Theory**. This framework allows us to perform fundamental graph operations directly in the continuous embedding space, reducing memory complexity from quadratic $O(N^2)$ to linear $O(Nd)$ while maintaining exact mathematical fidelity.

The specific contributions of this work are threefold:

7.1.1 Exact Diffusion on Implicit Graphs (VLP)

In Chapter 3, we addressed the scalability limits of semi-supervised learning. Traditional Label Propagation (LP) algorithms fail on dense graphs because they require materializing a prohibitively large adjacency matrix or rely on destructive k -NN sparsification to fit in memory. We introduced **Vector-Based Label Propagation (VLP)**, a reformulation that computes the diffusion of labels using associative matrix multiplication. We proved mathematically and empirically that VLP is exact, allowing us to perform global label diffusion on graphs with millions of nodes in seconds on a single commodity GPU.

7.1.2 Hierarchical Structure Discovery (VLouvain)

In Chapter 4, we extended these principles from simple diffusion to active topological modification. We introduced **VLouvain**, an algorithm capable of performing hierarchical community detection without ever building the explicit similarity graph. By proving the Vector Supernode Theorem, we demonstrated that modularity gain can be calculated entirely via vector aggregation. This bypassed the $O(N^2)$ graph construction bottleneck, enabling multi-level semantic clustering on massive dense corpora where traditional Louvain and explicit graph methods become intractable.

7.1.3 Topological Retrieval-Augmented Generation (GraphRAG-V)

In Chapter 5, we applied implicit graph theory to the frontier of Large Language Models (LLMs). We proposed **GraphRAG-V**, which utilizes VLouvain to group text chunks into dense, semantic communities instead of relying on expensive, LLM-generated triple extractions. By treating the document corpus as an implicit continuous graph, GraphRAG-V drastically cut indexing time from hours to minutes while successfully answering complex, multi-hop queries, proving that vector topology is a highly efficient substitute for explicit knowledge graphs.

7.2 The Unified Theory: The Graph as an Operator

The transition from “Explicit” to “Implicit” is not merely a computational optimization; it reframes how dense semantic graphs can be reasoned about. For decades, Graph Theory has been constrained by the mental model of the “Link”—a discrete, stored connection between two atomic entities.

This dissertation argues that in the age of deep representation learning, for dense semantic graphs, the explicit link abstraction is becoming unnecessary. When neural networks map the world into dense, continuous vectors, the relationship between two entities is no longer a boolean flag stored in a database; it is a geometric potential waiting to be calculated. By treating the graph as a linear operator rather than a static container, we align classical topological algorithms with the highly optimized tensor hardware that powers modern AI.

7.3 Limitations

While V Louvain and the broader implicit graph framework offer clear scalability advantages, they are not without limitations.

First, the framework relies on the **Fixed Embedding Assumption**. In the algorithms presented in Chapters 3 through 5, we consume representations generated by upstream models (e.g., FastText, CLIP, or standard Two-Tower outputs); we do not train them. The quality of the topological discovery is therefore bound by the geometric quality of the pre-trained latent space.

Second, there is a **Density Trade-off**. The implicit matrix operations defined in this work are most effective for dense, low-rank graphs (such as semantic similarity or attention matrices). However, for extremely sparse, physical networks (such as road infrastructure or highly sparse explicit web links), traditional explicit CSR (Compressed Sparse Row) formats and standard graph traversals remain more memory-efficient.

Third, while we have reduced the memory complexity from $O(N^2)$ to $O(Nd)$, the algorithms are still bound by **Single-Node Memory Constraints**. Storing the dense feature matrix $\mathbf{V}$ for tens of millions of high-dimensional nodes requires significant VRAM, necessitating multi-GPU sharding for web-scale corpora.

Finally, there is a **Scope Limitation**. The implicit framework as developed here addresses modularity-based community detection and linear diffusion, but other notions of cohesive structure, such as k -core, k -truss, and nucleus decomposition, have been scaled to very large explicit graphs [39, 17, 18, 34] but remain tied to explicit edge storage. Reformulating these primitives in the implicit setting is a promising open problem.

7.4 Final Remarks

By treating the graph as a linear operator, we have lowered the hardware barrier to massive-scale graph mining. A single researcher with a commodity GPU can now perform computations (diffusion on millions of nodes, clustering of massive textual corpora, and global reasoning over extensive knowledge bases) that previously would have been computationally prohibitive.

As discussed in Chapter 6, future directions include streaming algorithms, directed graphs, cross-modal fusion, and integration with attention mechanisms.

It is our hope that this work serves as a foundational bridge between classical Network Science and modern Deep Learning, paving the way for a new generation of graph algorithms that are as fluid, scalable, and high-dimensional as the data they seek to understand.

Appendix A

Detailed Proofs for VLouvain Exactness

This appendix expands the three lemmas underpinning the Exactness Theorem of Chapter 4 (Theorem 4.2) into fully written-out derivations. The chapter version states each result and gives a compact proof; here we walk through every algebraic step using exactly the same notation, conventions, and OOLF formulas that appear in Chapter 4, so that a reader who wants to check every line can do so without translating between conventions.

A.1 Notation and Conventions

We follow the conventions established in Chapter 4:

- Each node i is represented by an *augmented* embedding $\mathbf{v}_i \in \mathbb{R}^{d+1}$ obtained by appending a constant component to the original feature vector (the cosine augmentation introduced in Chapter 3). The construction guarantees $\mathbf{v}_i \cdot \mathbf{v}_j \geq 0$ for all i, j .
- The implicit adjacency is $A_{ij} = \mathbf{v}_i \cdot \mathbf{v}_j$ for $i \neq j$, with $A_{ii} = 0$ (no self-loops). Equivalently, $\mathbf{A} = \mathbf{V}\mathbf{V}^\top - \text{diag}(\mathbf{V}\mathbf{V}^\top)$, where $\mathbf{V} \in \mathbb{R}^{N \times (d+1)}$ stacks the augmented embeddings as rows.
- For a community $c \subseteq \{1, \dots, N\}$, the supernode vector and degree aggregate are

$$V^c = \sum_{j \in c} \mathbf{v}_j, \quad K^c = \sum_{j \in c} k_j,$$

where k_j is the degree of node j . The total mass is $\mathbf{V}_{\text{sum}} = \sum_{j=1}^N \mathbf{v}_j$ and $2m = \sum_{j=1}^N k_j$.

- $\gamma > 0$ is the resolution parameter.

We will repeatedly use the Vector Supernode Theorem (Theorem 4.1): for any two disjoint sets C_a and C_b ,

$$\mathbf{V}^a \cdot \mathbf{V}^b = \sum_{i \in C_a} \sum_{j \in C_b} (\mathbf{v}_i \cdot \mathbf{v}_j) = \sum_{i \in C_a} \sum_{j \in C_b} A_{ij}. \quad (\text{A.1})$$

Equation (A.1) is just bilinearity of the inner product applied to two finite sums, but it is the workhorse identity behind every result in this appendix.

A.2 Degree Equivalence (Lemma 4.3)

Lemma (Degree Equivalence; restatement of Lemma 4.3). *The implicit degree $k_i^{\text{imp}} = \mathbf{v}_i \cdot \mathbf{V}_{\text{sum}} - \mathbf{v}_i \cdot \mathbf{v}_i$ equals the explicit degree $k_i^{\text{exp}} = \sum_{j \neq i} A_{ij}$.*

Proof. **Step 1.** Expand $\mathbf{V}_{\text{sum}} = \sum_{j=1}^N \mathbf{v}_j$:

$$\mathbf{v}_i \cdot \mathbf{V}_{\text{sum}} = \mathbf{v}_i \cdot \left(\sum_{j=1}^N \mathbf{v}_j \right).$$

Step 2. Apply linearity of the inner product:

$$\mathbf{v}_i \cdot \mathbf{V}_{\text{sum}} = \sum_{j=1}^N (\mathbf{v}_i \cdot \mathbf{v}_j).$$

Step 3. Split the sum into the diagonal term and the off-diagonal terms:

$$\mathbf{v}_i \cdot \mathbf{V}_{\text{sum}} = (\mathbf{v}_i \cdot \mathbf{v}_i) + \sum_{j \neq i} (\mathbf{v}_i \cdot \mathbf{v}_j).$$

Step 4. Substitute the implicit-adjacency convention $A_{ij} = \mathbf{v}_i \cdot \mathbf{v}_j$ for $j \neq i$:

$$\mathbf{v}_i \cdot \mathbf{V}_{\text{sum}} = (\mathbf{v}_i \cdot \mathbf{v}_i) + \sum_{j \neq i} A_{ij}.$$

Step 5. Subtract $\mathbf{v}_i \cdot \mathbf{v}_i$ from both sides:

$$k_i^{\text{imp}} = \mathbf{v}_i \cdot \mathbf{V}_{\text{sum}} - \mathbf{v}_i \cdot \mathbf{v}_i = \sum_{j \neq i} A_{ij} = k_i^{\text{exp}}. \quad \square$$

Remark. The subtraction of $\mathbf{v}_i \cdot \mathbf{v}_i$ is the implicit-graph counterpart of removing the diagonal of $\mathbf{A}$, and it is essential: the cosine-augmented vectors satisfy $\mathbf{v}_i \cdot \mathbf{v}_i > 0$, so omitting this correction would inflate every degree by exactly the self-loop weight.

A.3 Modularity Gain Equivalence (Lemma 4.4)

We now derive the modularity gain produced by Algorithm 5 step by step and show that it matches the standard Louvain modularity gain. Throughout this section, V^{c_i} and K^{c_i} denote the community aggregates as stored in the OOLF state $(\mathbf{VC}, \mathbf{KC})$, which by definition *include* node i 's own contribution. The target community $c_j \neq c_i$ does not contain i , so V^{c_j} and K^{c_j} are unaffected.

A.3.1 Insertion Side

Lemma A.1 (Insertion Equivalence). *The OOLF insertion entry*

$$\mathbf{q}_{\text{in}}[j] = \frac{1}{2m} \left(\mathbf{v}_i \cdot V^{c_j} - \gamma \frac{k_i}{2m} K^{c_j} \right)$$

equals the standard explicit-graph contribution gained by inserting i into community c_j ,

$$\Delta Q_{\text{in}}^{\text{exp}}(i \rightarrow c_j) = \frac{1}{2m} \left(\sum_{k \in c_j} A_{ik} - \gamma \frac{k_i}{2m} \sum_{k \in c_j} k_k \right).$$

Proof. We match the two terms of $\mathbf{q}_{\text{in}}[j]$ to the two terms of $\Delta Q_{\text{in}}^{\text{exp}}$ in turn.

Affinity term. Apply Equation (A.1) with $C_a = \{i\}$ and $C_b = c_j$. Since $i \notin c_j$, the diagonal term $\mathbf{v}_i \cdot \mathbf{v}_i$ does not appear in the double sum, so

$$\mathbf{v}_i \cdot V^{c_j} = \mathbf{v}_i \cdot \left(\sum_{k \in c_j} \mathbf{v}_k \right) = \sum_{k \in c_j} (\mathbf{v}_i \cdot \mathbf{v}_k) = \sum_{k \in c_j} A_{ik}.$$

Resolution term. By Lemma 4.3, every aggregate degree in OOLF agrees with

the explicit-graph degree, so

$$K^{c_j} = \sum_{k \in c_j} k_k^{\text{imp}} = \sum_{k \in c_j} k_k^{\text{exp}}.$$

Multiplying by $\gamma k_i/(2m)$ and substituting both terms into $\mathbf{q}_{\text{in}}[j]$ yields exactly $\Delta Q_{\text{in}}^{\text{exp}}(i \rightarrow c_j)$. $\square$

A.3.2 Removal Side

The removal side requires more care because V^{c_i} and K^{c_i} as stored by OOLF *include* node i . The corresponding entry $\mathbf{q}_{\text{in}}[i]$ therefore double-counts i relative to the standard Louvain removal contribution, and the explicit self-loop correction in Algorithm 5 (line referenced as Equation 4.12 in Chapter 4) is what restores the correct value.

Lemma A.2 (Removal Equivalence). *The OOLF removal value*

$$q_{\text{out}} = \mathbf{q}_{\text{in}}[i] - \frac{1}{2m} \left(\mathbf{v}_i \cdot \mathbf{v}_i - \gamma \frac{k_i^2}{2m} \right)$$

equals the standard explicit-graph contribution lost by removing i from c_i ,

$$\Delta Q_{\text{out}}^{\text{exp}}(i \leftarrow c_i) = \frac{1}{2m} \left(\sum_{\substack{k \in c_i \\ k \neq i}} A_{ik} - \gamma \frac{k_i}{2m} \sum_{\substack{k \in c_i \\ k \neq i}} k_k \right).$$

Proof. We expand $\mathbf{q}_{\text{in}}[i]$, isolate i 's self-contribution, then verify that subtracting the OOLF self-loop correction leaves exactly $\Delta Q_{\text{out}}^{\text{exp}}$.

Step 1. Write the affinity term of $\mathbf{q}_{\text{in}}[i]$ by separating i from the rest of c_i :

$$\begin{aligned} \mathbf{v}_i \cdot V^{c_i} &= \mathbf{v}_i \cdot \left(\sum_{k \in c_i} \mathbf{v}_k \right) \\ &= \mathbf{v}_i \cdot \mathbf{v}_i + \mathbf{v}_i \cdot \left(\sum_{\substack{k \in c_i \\ k \neq i}} \mathbf{v}_k \right) \\ &= (\mathbf{v}_i \cdot \mathbf{v}_i) + (\mathbf{v}_i \cdot V^{c_i \setminus \{i\}}). \end{aligned}$$

Step 2. Likewise split the resolution term:

$$K^{c_i} = k_i + \sum_{\substack{k \in c_i \\ k \neq i}} k_k = k_i + K^{c_i \setminus \{i\}}.$$

Step 3. Substitute Steps 1–2 into $\mathbf{q}_{\text{in}}[i]$:

$$\begin{aligned} \mathbf{q}_{\text{in}}[i] &= \frac{1}{2m} \left(\mathbf{v}_i \cdot V^{c_i} - \gamma \frac{k_i}{2m} K^{c_i} \right) \\ &= \frac{1}{2m} \left([\mathbf{v}_i \cdot \mathbf{v}_i + \mathbf{v}_i \cdot V^{c_i \setminus \{i\}}] - \gamma \frac{k_i}{2m} [k_i + K^{c_i \setminus \{i\}}] \right) \\ &= \underbrace{\frac{1}{2m} \left(\mathbf{v}_i \cdot V^{c_i \setminus \{i\}} - \gamma \frac{k_i}{2m} K^{c_i \setminus \{i\}} \right)}_{=: \Delta Q_{\text{out}}^{\text{vlv}}} + \underbrace{\frac{1}{2m} \left(\mathbf{v}_i \cdot \mathbf{v}_i - \gamma \frac{k_i^2}{2m} \right)}_{\text{self-loop term}}. \end{aligned}$$

Step 4. Subtract the self-loop term, as Algorithm 5 does:

$$q_{\text{out}} = \mathbf{q}_{\text{in}}[i] - \frac{1}{2m} \left(\mathbf{v}_i \cdot \mathbf{v}_i - \gamma \frac{k_i^2}{2m} \right) = \Delta Q_{\text{out}}^{\text{vlv}}.$$

Step 5. Identify $\Delta Q_{\text{out}}^{\text{vlv}}$ with the explicit-graph removal contribution. Apply Equation (A.1) to $C_a = \{i\}$ and $C_b = c_i \setminus \{i\}$ for the affinity term, and apply Lemma 4.3 to $K^{c_i \setminus \{i\}}$ for the resolution term:

$$\mathbf{v}_i \cdot V^{c_i \setminus \{i\}} = \sum_{\substack{k \in c_i \\ k \neq i}} A_{ik}, \quad K^{c_i \setminus \{i\}} = \sum_{\substack{k \in c_i \\ k \neq i}} k_k.$$

Substituting into $\Delta Q_{\text{out}}^{\text{vlv}}$ yields $\Delta Q_{\text{out}}^{\text{exp}}(i \leftarrow c_i)$. □

Why the self-loop correction is necessary. Because OOLF stores aggregates that include i , computing the removal value naively as $\mathbf{q}_{\text{in}}[i]$ would charge i 's self-affinity ($\mathbf{v}_i \cdot \mathbf{v}_i$) and i 's self-resolution ($\gamma k_i^2 / (2m)$) against itself. The correction term $\frac{1}{2m}(\mathbf{v}_i \cdot \mathbf{v}_i - \gamma k_i^2 / (2m))$ is exactly what is needed to subtract these two artefacts, leaving the contribution from edges and resolution between i and the *other* members of c_i . This is the implicit-graph analogue of treating the diagonal of $\mathbf{A}$ separately in the standard formulation.

A.3.3 Net Gain

Combining the two sides:

$$\Delta Q^{\text{vlv}}(i : c_i \rightarrow c_j) = \mathbf{q}_{\text{in}}[j] - q_{\text{out}}.$$

By Lemma A.1, $\mathbf{q}_{\text{in}}[j] = \Delta Q_{\text{in}}^{\text{exp}}(i \rightarrow c_j)$. By Lemma A.2, $q_{\text{out}} = \Delta Q_{\text{out}}^{\text{exp}}(i \leftarrow c_i)$. Hence

$$\Delta Q^{\text{vlv}}(i : c_i \rightarrow c_j) = \Delta Q_{\text{in}}^{\text{exp}}(i \rightarrow c_j) - \Delta Q_{\text{out}}^{\text{exp}}(i \leftarrow c_i),$$

which is the standard Louvain modularity gain from moving i between c_i and c_j . $\square$

Note on prefactor conventions. The $(1/2m)$ prefactor used by OOLF differs from the $(1/m)$ that appears in some textbook derivations of ΔQ (e.g. Blondel et al. [3]) by a factor of two. The discrepancy stems from whether the underlying modularity sum is taken over ordered or unordered pairs, and is absorbed into the definition of $2m$. It does not affect any greedy decision: positive-gain moves remain positive-gain under either convention, and the partition sequence is identical. The Exactness Theorem compares VLouvain to standard Louvain run with the same convention, so this scaling never enters the proof.

A.4 Aggregation Equivalence (Lemma 4.5)

After a level of VLouvain converges, ONGG produces, for each community c , a supernode vector $V^c = \sum_{i \in c} \mathbf{v}_i$. The next level of VLouvain runs on the matrix of these supernode vectors, with implicit adjacency $A_{cd} = V^c \cdot V^d$ for $c \neq d$ and the diagonal $V^c \cdot V^c$ removed (exactly as at level zero). We must show that this implicit aggregated graph coincides with what explicit Louvain would build by summing edges between communities.

Lemma (Aggregation Equivalence; restatement of Lemma 4.5). *Let C_a and C_b be two disjoint communities at the end of a level. Then (i) $V^a \cdot V^b = W_{a,b}^{\text{exp}} := \sum_{i \in C_a, j \in C_b} A_{ij}$, and (ii) the new-level degree of supernode c computed implicitly as $k_c^{\text{new, imp}} = V^c \cdot \mathbf{V}_{\text{sum}}^{\text{new}} - V^c \cdot V^c$ equals the explicit new-level degree $k_c^{\text{new, exp}} = \sum_{d \neq c} W_{c,d}^{\text{exp}}$.*

Proof. Part (i): Inter-community edge weight. Applying Equation (A.1) di-

rectly with C_a and C_b disjoint,

$$V^a \cdot V^b = \left(\sum_{i \in C_a} \mathbf{v}_i \right) \cdot \left(\sum_{j \in C_b} \mathbf{v}_j \right) = \sum_{i \in C_a} \sum_{j \in C_b} (\mathbf{v}_i \cdot \mathbf{v}_j) = \sum_{i \in C_a} \sum_{j \in C_b} A_{ij} = W_{a,b}^{\text{exp}}.$$

Disjointness ensures no diagonal term $\mathbf{v}_i \cdot \mathbf{v}_i$ appears in the double sum, so the A_{ij} values are all genuine off-diagonal entries, and the sum matches the explicit aggregated edge weight.

Part (ii): New-level degree. Let $\mathbf{V}_{\text{sum}}^{\text{new}} = \sum_d V^d$, where the index d runs over all communities at the current level. Expand:

$$\begin{aligned} V^c \cdot \mathbf{V}_{\text{sum}}^{\text{new}} &= V^c \cdot \left(\sum_d V^d \right) \\ &= \sum_d (V^c \cdot V^d) \\ &= (V^c \cdot V^c) + \sum_{d \neq c} (V^c \cdot V^d). \end{aligned}$$

Subtracting the diagonal term $V^c \cdot V^c$,

$$k_c^{\text{new, imp}} = \sum_{d \neq c} (V^c \cdot V^d) \stackrel{(*)}{=} \sum_{d \neq c} W_{c,d}^{\text{exp}} = k_c^{\text{new, exp}},$$

where $(*)$ uses Part (i) for each pair (c, d) with $c \neq d$. □

Where do the original self-loops go? The within-community sum $V^c \cdot V^c$ contains both the diagonal terms $\mathbf{v}_i \cdot \mathbf{v}_i$ for $i \in c$ and the genuine intra-community edges $\sum_{i \neq j \in c} (\mathbf{v}_i \cdot \mathbf{v}_j)$. In the explicit-graph formulation, intra-community edges become a self-loop on the supernode in the aggregated graph. V Louvain (and the version of Louvain it is compared against) uses the convention that supernode self-loops do *not* contribute to a supernode's degree at the next level, since degree counts only edges to *other* supernodes. Subtracting $V^c \cdot V^c$ in the new-level degree formula has exactly this effect, which is why the same implicit-degree formula can be reused at every level. Some presentations of Louvain (e.g. Blondel et al. [3]) instead fold the supernode self-loop into the degree; the two conventions yield identical partition sequences because the self-loop term cancels in every ΔQ difference.

A.5 From the Three Lemmas to the Exactness Theorem

With the three lemmas above in hand, the Exactness Theorem of Chapter 4 (Theorem 4.2) follows by induction on the hierarchy level.

- **Level 0.** Both algorithms start from the same node set and the same edge weights (by Lemma 4.3, the degree vector $\mathbf{k}$ is identical). Under the same node-visit order and tie-breaking, they evaluate every candidate move with the same modularity gain (by Lemma 4.4), and therefore make the same sequence of moves and converge to the same partition.
- **Aggregation step.** ONGG produces supernode vectors whose implicit adjacency coincides with the explicit aggregated graph (by Lemma 4.5, Part (i)) and whose degrees coincide with the explicit aggregated degrees (Part (ii)). The next level therefore starts from the same state in both algorithms.
- **Inductive step.** Apply the Level-0 argument to the new level.

Hence the partition sequences $\mathcal{P}_{\text{exp}}$ and $\mathcal{P}_{\text{vlv}}$ coincide, modulo the floating-point caveat noted at the end of Section 4.4.

A.6 Summary

The three lemmas of Chapter 4 reduce, after expansion, to one identity (the Vector Supernode bilinearity, Equation (A.1)) plus careful bookkeeping of the diagonal $\mathbf{v}_i \cdot \mathbf{v}_i$. The diagonal appears in three places:

1. In the implicit degree formula, where it is subtracted to remove self-loops (Section A.2).
2. In the OOLF removal value, where it is subtracted as part of the self-loop correction so that i is not charged against itself (Section A.3.2).
3. In the new-level degree formula on aggregated supernodes, where $V^c \cdot V^c$ is subtracted so that intra-community edges become a supernode self-loop and do not contribute to the supernode's degree (Section A.4).

Each of these is the implicit-graph counterpart of a routine bookkeeping step in standard Louvain. Once they are made explicit, the equivalence between VLouvain and Louvain becomes a chain of substitutions rather than an approximation argument.

Appendix B

VLouvain Determinism Check

The experiments reported in Chapter 4 were run on the four real-world datasets at A100 scale. As a smaller, locally reproducible sanity check, we also ran the corrected vectorized VLouvain implementation five times on a synthetic 10,000-node, 128-dimensional dense graph using the PyTorch MPS backend on an M4 Mac (seeds 42–46, $\gamma = 1$). The purpose is narrow: to confirm that the implementation is deterministic and that runtime is stable on hardware that does not require cloud access. It is not a re-run of the chapter’s experiments and is not an empirical validation of the Exactness Theorem; for the latter, see the modularity-comparison experiment in Section 4.5.1 and the formal proofs in Appendix A.

Table B.1: Per-run results for the determinism check (5 runs, 10,000 nodes, M4 Mac MPS, $\gamma = 1$).

Run	Seed	Runtime (s)	Q	#Comm.	#Levels
1	42	74.12	0.039856	7	3
2	43	68.92	0.039856	7	3
3	44	71.76	0.039856	7	3
4	45	68.97	0.039856	7	3
5	46	69.03	0.039856	7	3
Mean	—	70.56	0.039856	7	3
Sample std	—	2.33	0.000000	0	0

The five runs produce an identical partition (modularity, community count, and hierarchy depth all match exactly), and runtime varies by 2.33 s (sample standard deviation) around a mean of 70.56 s (coefficient of variation 3.3%). This is the expected behaviour for an implementation with deterministic node-visit order and

tie-breaking, and it is the most this experiment is meant to show. The runtime numbers are not directly comparable to the chapter's A100 results: the M4 MPS backend has roughly an order of magnitude less throughput and memory bandwidth than the A100, and is included here only because it is the hardware on which the check can be re-executed without cloud access.

Bibliography

- [1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.
- [2] Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. Self-rag: Learning to retrieve, generate, and critique through self-reflection. In *12th International Conference on Learning Representations*, 2023.
- [3] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.
- [4] Thomas Bonald, Nathan de Lara, Quentin Lutz, and Bertrand Charpentier. Scikit-network: Graph analysis in python. *Journal of Machine Learning Research*, 21(185):1–6, 2020.
- [5] Andries E Brouwer and Willem H Haemers. *Spectra of graphs*. Springer Science & Business Media, 2011.
- [6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *NeurIPS*, 33:1877–1901, 2020.
- [7] Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. Bge m3-embedding: Multi-lingual, multi-functionality, multi-granularity text embeddings through self-knowledge distillation, 2024.

- [8] Mingming Chen, Konstantin Kuzmin, and Boleslaw K Szymanski. Community detection via maximization of modularity and its variants. *IEEE Transactions on Computational Social Systems*, 1(1):46–65, 2014.
- [9] Fan RK Chung. *Spectral Graph Theory*. American Mathematical Society, 1997.
- [10] Gennaro Cordasco and Luisa Gargano. Community detection via semi-synchronous label propagation algorithms. In *2010 IEEE international workshop on: business applications of social network analysis (BASNA)*, pages 1–8. IEEE, 2010.
- [11] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*, RecSys ’16, pages 191–198, New York, NY, USA, 2016. ACM.
- [12] Gabor Csardi and Tamas Nepusz. The igraph software. *Complex syst*, 1695:1–9, 2006.
- [13] Michał Daniluk, Jacek Dabrowski, Barbara Rychalska, and Konrad Gołuchowski. Synerise at kdd cup 2021: Node classification in massive heterogeneous graphs. *KDD Cup OGB Challenge 2021*, 2021.
- [14] Leon Danon, Jordi Duch, Albert Díaz-Guilera, and Alex Arenas. Comparing community structure identification. *Journal of Statistical Mechanics: Theory and Experiment*, (09):P09008, 2005.
- [15] Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Advances in Neural Information Processing Systems*, volume 35, pages 16344–16359, 2022.
- [16] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, and Jonathan Larson. From local to global: A graph rag approach to query-focused summarization. *arXiv preprint arXiv:2404.16130*, 2024.
- [17] Fatemeh Esfahani, Venkatesh Srinivasan, Alex Thomo, and Kui Wu. Fast truss decomposition in large-scale probabilistic graphs. In *EDBT*, pages 494–505, 2019.

- [18] Fatemeh Esfahani, Venkatesh Srinivasan, Alex Thomo, and Kui Wu. Nucleus decomposition in probabilistic graphs: Hardness and algorithms. In *ICDE*, pages 218–230, 2022.
- [19] Alex Fender, Brad Rees, and Joe Eaton. Rapids cugraph. In *Massive Graph Analytics*, pages 483–493. Chapman and Hall/CRC, 2022.
- [20] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.
- [21] Evelyn Fix. *Discriminatory analysis: nonparametric discrimination, consistency properties*, volume 1. USAF school of Aviation Medicine, 1985.
- [22] Santo Fortunato. Community detection in graphs. *Physics reports*, 486(3-5):75–174, 2010.
- [23] Santo Fortunato and Marc Barthélemy. Resolution limit in community detection. *Proceedings of the National Academy of Sciences*, 104(1):36–41, 2007.
- [24] Yasuhiro Fujiwara and Go Irie. Efficient label propagation. In *International conference on machine learning*, pages 784–792. PMLR, 2014.
- [25] Sara E Garza and Satu Elisa Schaeffer. Community detection with the label propagation algorithm: a survey. *Physica A: Statistical Mechanics and its Applications*, 534:122058, 2019.
- [26] Fabio Gasparetti, Alessandro Micarelli, and Giuseppe Sansonetti. Community detection and recommender systems., 2018.
- [27] Fabio Gasparetti, Giuseppe Sansonetti, and Alessandro Micarelli. Community detection in social recommender systems: a survey. *Applied Intelligence*, 51(6):3975–3995, 2021.
- [28] Sayan Ghosh, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, Hao Lu, Daniel Chavarria-Miranda, Arif Khan, and Assefaw Gebremedhin. Distributed louvain algorithm for graph community detection. In *2018 IEEE international parallel and distributed processing symposium (IPDPS)*, pages 885–895. IEEE, 2018.

- [29] Michelle Girvan and Mark EJ Newman. Community structure in social and biological networks. *Proceedings of the national academy of sciences*, 99(12):7821–7826, 2002.
- [30] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.
- [31] Alibaba Group. Taobao user behavior dataset. <https://tianchi.aliyun.com/dataset/649>. Accessed: 2024-12-31.
- [32] Zirui Guo, Lianghao Xia, Yanhua Yu, Tu Ao, and Chao Huang. Lightrag: Simple and fast retrieval-augmented generation. *arXiv preprint arXiv:2410.05779*, 2024.
- [33] Bernal Jiménez Gutiérrez, Yiheng Shu, Yu Gu, Michihiro Yasunaga, and Yu Su. Hipporag: Neurobiologically inspired long-term memory for large language models. In *NeurIPS*, 2024.
- [34] Joseph Howie, Venkatesh Srinivasan, and Alex Thomo. Scaling up structural clustering to large probabilistic graphs using Lyapunov central limit theorem. *Proceedings of the VLDB Endowment*, 16(11):3165–3177, 2023.
- [35] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, et al. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Inf. Systems*, 43(2):1–55, 2025.
- [36] Yiqian Huang, Shiqi Zhang, and Xiaokui Xiao. Ket-rag: A cost-efficient multi-granular indexing framework for graph-rag. *arXiv preprint arXiv:2502.09304*, 2025.
- [37] Vassilis N Ioannidis, Ahmed S Zamzam, Georgios B Giannakis, and Nicholas D Sidiropoulos. Coupled graphs and tensor factorization for recommender systems and community detection. *IEEE Transactions on Knowledge and Data Engineering*, 33(3):909–920, 2019.
- [38] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick SH Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering. In *EMNLP (1)*, pages 6769–6781, 2020.

- [39] Wissam Khaouid, Marina Barsky, Venkatesh Srinivasan, and Alex Thomo. K-core decomposition of large networks on a single pc. *Proceedings of the VLDB Endowment*, 9(1):13–23, 2015.
- [40] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *International Conference on Learning Representations*, 2020.
- [41] Kishore Kothapalli, Sriram V Pemmaraju, and Vivek Sardeshmukh. On the analysis of a label propagation algorithm for community detection. In *International Conference on Distributed Computing and Networking*, pages 255–269. Springer, 2013.
- [42] Daniel D Lee and H Sebastian Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401(6755):788–791, 1999.
- [43] Ian XY Leung, Pan Hui, Pietro Lio, and Jon Crowcroft. Towards real-time community detection in large networks. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics*, 79(6):066107, 2009.
- [44] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *NeurIPS*, 33:9459–9474, 2020.
- [45] Zhuliu Li, Raphael Petegrosso, Shaden Smith, David Sterling, George Karypis, and Rui Kuang. Scalable label propagation for multi-relational learning on the tensor product of graphs. *IEEE Transactions on Knowledge and Data Engineering*, 34(12):5964–5978, 2021.
- [46] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024.
- [47] Wei Liu, Junfeng He, and Shih-Fu Chang. Large graph construction for scalable semi-supervised learning. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 679–686. Citeseer, 2010.

- [48] Shengjie Ma, Chengjin Xu, Xuhui Jiang, Muzhi Li, Huaren Qu, Cehao Yang, Jiaxin Mao, and Jian Guo. Think-on-graph 2.0: Deep and faithful large language model reasoning with knowledge-guided retrieval augmented generation. *arXiv preprint arXiv:2407.10805*, 2024.
- [49] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.
- [50] Costas Mavromatis and George Karypis. Gnn-rag: Graph neural retrieval for large language model reasoning. *Preprint arXiv:2405.20139*, 2024.
- [51] Tomas Mikolov, Edouard Grave, Piotr Bojanowski, Christian Puhersch, and Armand Joulin. Advances in pre-training distributed word representations. *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*, 2018.
- [52] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.
- [53] Mark EJ Newman. Modularity and community structure in networks. *Proceedings of the national academy of sciences*, 103(23):8577–8582, 2006.
- [54] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [55] Raphael Petegrosso, Wei Zhang, Zhuliu Li, Yousef Saad, and Rui Kuang. Low-rank label propagation for semi-supervised learning with 100 millions samples. *arXiv preprint arXiv:1702.08884*, 2017.
- [56] Xinyu Que, Fabio Checconi, Fabrizio Petrini, and John A Gunnels. Scalable community detection with the louvain algorithm. In *2015 IEEE international parallel and distributed processing symposium*, pages 28–37. IEEE, 2015.
- [57] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics*, 76(3):036106, 2007.

- [58] Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. In-context retrieval-augmented language models. *Transactions of the Association for Computational Linguistics*, 11:1316–1331, 2023.
- [59] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.
- [60] Martin Rosvall and Carl T Bergstrom. Maps of random walks on complex networks reveal community structure. *Proceedings of the national academy of sciences*, 105(4):1118–1123, 2008.
- [61] Subhajit Sahu. Gve-louvain: Fast louvain algorithm for community detection in shared memory setting. *arXiv preprint arXiv:2312.04876*, 2023.
- [62] Keshav Santhanam, Omar Khattab, Jon Saad-Falcon, Christopher Potts, and Matei Zaharia. Colbertv2: Effective and efficient retrieval via lightweight late interaction. *arXiv preprint arXiv:2112.01488*, 2021.
- [63] Bernhard Schölkopf and Alexander J Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT press, Cambridge, MA, 2002.
- [64] Weijia Shi, Sewon Min, Michihiro Yasunaga, Minjoon Seo, Rich James, Mike Lewis, Luke Zettlemoyer, and Wen-tau Yih. Replug: Retrieval-augmented black-box language models. *arXiv preprint arXiv:2301.12652*, 2023.
- [65] Zeinab Shokrzadeh, Mohammad-Reza Feizi-Derakhshi, Mohammad-Ali Balafar, and Jamshid Bagherzadeh Mohasefi. Graph-based recommendation system enhanced by community detection. *Scientific Programming*, 2023(1):5073769, 2023.
- [66] Amit Singhal. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4):35–43, 2001.
- [67] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. MpNet: Masked and permuted pre-training for language understanding. *NeurIPS*, 33:16857–16867, 2020.
- [68] Christian L Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. Networkkit: A tool suite for high-performance network analysis. *Network Science*, 2014.

- [69] Yixuan Tang and Yi Yang. Multihop-rag: Benchmarking retrieval-augmented generation for multi-hop queries. *arXiv preprint arXiv:2401.15391*, 2024.
- [70] Qwen Team. Qwen2.5: A party of foundation models, September 2024.
- [71] Vincent A Traag, Ludo Waltman, and Nees Jan Van Eck. From louvain to leiden: guaranteeing well-connected communities. *Scientific reports*, 9(1):1–12, 2019.
- [72] Shu Wang, Yixiang Fang, Yingli Zhou, Xilin Liu, and Yuchi Ma. Archrag: Attributed community-based hierarchical retrieval-augmented generation. *arXiv preprint arXiv:2502.09891*, 2025.
- [73] Shitao Xiao, Zheng Liu, Peitian Zhang, and Niklas Muennighoff. C-pack: Packaged resources to advance general chinese embedding, 2023.
- [74] Jierui Xie and Boleslaw K Szymanski. Labelrank: A stabilized label propagation algorithm for community detection in networks. In *2013 IEEE 2nd Network Science Workshop (NSW)*, pages 138–143. IEEE, 2013.
- [75] Teng kai Yu, Venkatesh Srinivasan, and Alex Thomo. Efficient vector-based label propagation for massive low-rank graphs. In *WSDM*, pages 1278–1282, 2026.
- [76] Teng kai Yu, Venkatesh Srinivasan, and Alex Thomo. Efficient vector-based louvain algorithm for massive low-rank graphs. In *EDBT*, pages 537–543, 2026.
- [77] Teng kai Yu, Srinivasan Venkatesh, and Alex Thomo. Graphrag-v: Fast multi-hop retrieval via text-chunk communities. In *ASONAM*, number 3, pages 3–14, 2025.
- [78] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Graphsaint: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931*, 2019.
- [79] Fangyuan Zhang, Zhengjun Huang, Yingli Zhou, Qintian Guo, Zhixun Li, Wensheng Luo, Di Jiang, Yixiang Fang, and Xiaofang Zhou. Erarag: Efficient and incremental retrieval augmented generation for growing corpora. *arXiv preprint arXiv:2506.20963*, 2025.
- [80] Wentao Zhang, Mingyu Yang, Zeang Sheng, Yang Li, Wen Ouyang, Yangyu Tao, Zhi Yang, and Bin Cui. Node dependent local smoothing for scalable graph

- learning. *Advances in Neural Information Processing Systems*, 34:20321–20332, 2021.
- [81] Penghao Zhao, Hailin Zhang, Qinhan Yu, Zhengren Wang, Yunteng Geng, Fangcheng Fu, Ling Yang, Wentao Zhang, Jie Jiang, and Bin Cui. Retrieval-augmented generation for ai-generated content: A survey. *preprint arXiv:2402.19473*, 2024.
 - [82] Xiaojin Zhu and Zoubin Ghahramani. Learning from labeled and unlabeled data with label propagation.(2002), 2002.
 - [83] Xiaojin Zhu, Zoubin Ghahramani, and John D Lafferty. Semi-supervised learning using gaussian fields and harmonic functions. In *Proceedings of the 20th International conference on Machine learning (ICML-03)*, pages 912–919, 2003.