Efficient Vector-Based Louvain Algorithm for Massive Low-Rank Graphs

Tengkai Yu
University of Victoria
Victoria, BC, Canada
yutengkai@uvic.ca

Venkatesh Srinivasan
Santa Clara University
Santa Clara, CA, USA
vsrinivasan4@scu.edu

Alex Thomo
University of Victoria
Victoria, BC, Canada
thomo@uvic.ca

Abstract

The Louvain algorithm is a widely used community detection method for producing multi-level community partitions with high modularity. However, existing implementations often struggle to scale efficiently on large graphs, limiting their usability in high-volume applications. In real-world applications, numerous high-volume low-rank graphs arise, such as similarity graphs constructed from embeddings in recommendation systems or natural language processing tasks. If edge weights are calculated and stored explicitly for such graphs, the resulting memory costs can become prohibitively large. In this paper, we introduce VLouvain, a novel approach that eliminates the need for explicit graph construction and bypasses iterating over edges while remaining mathematically identical to the original Louvain algorithm. In our experiments on real-world datasets, VLouvain was the only approach to handle low-rank graphs with millions of nodes in a few thousand seconds, where all others failed, showing its scalability for large graphs arising from node similarity computation in several critical applications.

Keywords

Social Networks, Community Detection, Louvain Algorithm, Low-Rank Graphs, Vector Based Computation

1 Introduction

Community detection is a fundamental task in graph analysis, where the goal is to partition a graph into subgroups (communities) such that nodes are densely connected within the same group and more sparsely connected across groups [10, 14, 17]. Among the most widely used methods are the Louvain [2] and Leiden [25] algorithms, both of which uncover these communities while also producing multi-level partitions. This hierarchical nature is highly beneficial, from social network analysis to recommender systems [11, 12, 15, 22] and graph-based retrieval-augmented generation (RAG) [7], as it reveals structure at multiple scales.

In this work, 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 each level. While powerful and widely adopted, Louvain faces significant scalability issues when applied to large graphs. Over the past decade, many methods have aimed to improve Louvain's efficiency [13, 18, 21], yet the need for explicit graph construction remains a major bottleneck for large graphs. When edges and their weights are derived from node feature vectors (e.g., similarity graphs based on cosine similarity among embeddings), the number of edges may grow quadratically with the node count. Formally, we say that a graph is low-rank if its $n \times n$ adjacency matrix A is of the form $A = VV^T$ for an $n \times d$ matrix V and d is much smaller than n. Storing and processing edges of such a graph in memory is often infeasible, particularly in domains like recommender systems, where user or item embeddings result in very dense networks.

In practice, node feature similarities (e.g., cosine similarities) can generate a near-complete graph Traditional approaches construct or store the entire graph before running the Louvain algorithm. For large, low-rank graphs, building and managing its structure is expensive in both computation and memory. Libraries like iGraph [5] and NetworKit [23] can handle sizable networks but still struggle with dense graphs. Even efficient variants like GVE-Louvain [21] face constraints once the graph becomes too dense.

We propose **VLouvain**[1], 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 substantial reduction in memory usage.

In this paper, we make the following contributions:

- **Memory Efficiency.** Our method dispenses with storing large graphs, keeping only node features and minimal aggregated statistics. This substantially lowers memory requirements in low-rank settings.
- **Matrix-Based Implementation.** Our approach replaces traditional nested node-edge loops with matrix or vector-based aggregation. This enables vectorized supernode representation for faster multi-level comparisons of communities and efficient GPU/parallel acceleration.
- **Mathematical Equivalence.** Although our implementation avoids explicit edge storage, it follows the Louvain framework [2] and is guaranteed to produce the same partitioning results as the original algorithm.

2 Related Work

Community detection identifies groups of nodes within a network that are more connected to each other than to the rest of the network. A common way to evaluate the quality of these groups is by using modularity, a measure that compares the density of links inside communities to what would be expected at random [14, 17]. Many methods have been proposed to find high-modularity partitions, and several surveys provide overviews of these techniques, their applications, and their limitations [4, 10, 16, 20, 26].

[1] Vector-Based **Louvain** algorithm

EDBT '26, Tampere (Finland)

10.48786/edbt.2026.43

Among these methods, the Louvain algorithm [2] stands out due to its efficiency and good results across a range of real-world networks. Louvain works by repeatedly merging nodes into communities if this merging increases modularity, and then building a smaller network of these communities to repeat the process. As a result, it can quickly reveal a hierarchical structure of communities. Later improvements and extensions, such as the Leiden algorithm, have focused on ensuring well-connected communities and more stable results [25], while other variants of modularity-based algorithms have aimed for near-linear time complexity [3, 19].

Despite its popularity, the Louvain algorithm and other modularity based methods face scalability challenges on very large networks. Researchers have explored various strategies, including parallelization in shared-memory settings [18], distributed computation across clusters [14, 20], and GPU acceleration to reduce execution time [9, 23]. Libraries such as NetworKit [26] and iGraph [5] further improve performance through optimized implementations and efficient data structures. Beyond direct Louvain optimization, others use implicit community definitions or graph embeddings to lower the cost of handling explicit structures [3, 8, 24]. While these approaches show promise, issues such as communication overhead, high memory demands, and handling implicit or context-dependent edges persist. Consequently, achieving both high-quality community detection and scalability on large networks remains an open challenge.

3 Problem Definition

In this section, we present the notion of modularity and give a high level description of the Louvain algorithm and the challenges.

3.1 Modularity

To measure community quality, the Louvain algorithm maximizes modularity Q, which compares the density of edges within communities to an expected baseline ([2, 17]). Formally:

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

where A_{ij} is the weight of the edge between nodes i and j, k_i and k_j are their degrees, m is the total edge weight (or number of edges), and δ is 1 when nodes share a community and 0 otherwise. Parameter γ controls resolution: higher values yield smaller communities, and lower values merge clusters into larger groups. During Louvain's iterations, each node's move is guided by the change in Q (ΔQ), indicating if the relocation improves the partition.

3.2 Louvain Algorithm

The Louvain algorithm [2] detects communities by alternating between two phases. First, each node is placed in its own community, and the algorithm iterates over all nodes to evaluate whether moving a node into a neighboring community increases modularity. This step is often called the "one-level" function: each node is reassigned if doing so yields a positive gain in modularity, and this process continues until no further improvement is possible within the current level. Second, the algorithm collapses or "generates" a new graph by merging all nodes in the same community into a single supernode. Edge weights between these supernodes reflect the sum of the original edges between their constituent nodes. The algorithm then repeats these phases on the newly formed graph, iteratively coarsening the graph until no additional modularity gain is observed, producing a hierarchical partition of the original graph.

Time and Memory Cost of the Original Louvain Algorithm. We observe that the majority of the computation occurs during the first call to the one-level function. At this stage, each node is initially placed in its own community and must visit all of its neighbors to evaluate whether moving to a new community would increase modularity. During this process, each edge is visited exactly twice—once by each of its source nodes. This makes the time complexity of the one-level function $O(\text{number of edges})$. The memory cost in this explicit graph setup is also dominated by the storage of edge weights, requiring $O(\text{number of edges})$ memory.

Challenges in low-rank Graphs. In scenarios where the graph is low-rank, the memory requirements grow rapidly. Such graphs can have up to $O(n^2)$ edges, where n is the number of nodes, making memory usage infeasible for large-scale settings like similarity-based item analysis or recommender systems. Explicitly storing edge weights in these cases becomes prohibitively expensive, rendering traditional graph-based community detection impractical.

4 Vector-Based Louvain: Key Operations and Data Structures

Now, we outline the vector-based formulation underlying VLouvain. Our goal is community detection directly from node embeddings, without explicitly constructing the full graph. Many practical graphs can be represented in low-rank form, $A = \mathbf{V}\mathbf{V}^\top$ with $\mathbf{V} \in \mathbb{R}^{n \times d}$, where each row $v_i \in \mathbb{R}^d$ 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, Gaussian, or 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 nonnegative (e.g., cosine similarities shifted to $[0, 1]$ or even-degree polynomial kernels).

Cosine similarity as a special case. In our experiments, we adopt normalized cosine similarity, a standard and interpretable measure in representation learning. Given unit-norm embeddings v_i, we define $A_{ij} = \frac{1}{2}(1 + v_i^\top v_j)$, which linearly rescales cosine similarity from $[-1, 1]$ to $[0, 1]$, ensuring nonnegativity while preserving the angular similarity signal. Equivalently, each vector can be augmented as $v_i' = \frac{[\, v_i; 1 \,]}{\sqrt{2}}$, so that $A_{ij} = v_i'^\top v_j'$. Collecting these rows yields $A = \mathbf{V}\mathbf{V}^\top$ (with zeroed diagonals for self-loops), where $\mathbf{V} = [v_1'; \ldots; v_n'] \in \mathbb{R}^{n \times (d+1)}$. For simplicity of notation, we drop the prime and denote the augmented vectors as v_i throughout the remainder of the paper. Our derivations will show that our algorithm is mathematically equivalent to the traditional Louvain algorithm.

Degree Calculation. The degree k_i of node i is the sum of the edge weights connected to it: $k_i = \sum_{j \neq i} A_{ij}$. For edge weights represented by inner products, we can rewrite k_i as $k_i = \sum_j (v_i \cdot v_j) - (v_i \cdot v_i)$, where $v_i \cdot v_i$ corresponds to the self-loop weight.

To calculate the degree vector $\mathbf{K}$ for all nodes efficiently without constructing $\mathbf{V}\mathbf{V}^T$ (which could be $O(n^2 d)$ in space), we first compute $\mathbf{s} = \mathbf{V}^T \mathbf{1}$, i.e., the row-wise sum of $\mathbf{V}$. We then let $\mathbf{K} = \mathbf{V}(\mathbf{s}^T) - \mathbf{d}$, where $\mathbf{d}$ is a vector whose i-th entry is

$v_i \cdot v_i$. Avoiding the explicit construction of $\mathbf{V}\mathbf{V}^T$ reduces the space complexity to $O(nd)$. Precomputing $\mathbf{K}$ is essential for modularity optimization, as it captures global graph information with minimal overhead.

Sum of Edge Weights. The total edge weight m is half the sum of all degrees, so each edge is counted only once: $m = \frac{\sum_i k_i}{2}$. In matrix form, m can be written as $\frac{1}{2}\,\mathbf{1}^T\mathbf{K}$, where $\mathbf{1}$ is a vector of ones.

4.1 Affinity and Resolution Terms

We have already introduced the modularity formula and its parameters in section 3.1. Here, we focus on the separate contributions within that formula (i.e., Affinity Term and Resolution Term) and how they factor into gain calculations.

4.1.1 Affinity Term (Q_{aff}) and Resolution Term (Q_{res}). We recall that modularity Q sums over two main components for nodes in the same community: observed edge weights (Affinity) and an expected edge weight scaled by γ (Resolution). The Affinity component $Q_{\text{aff}} = \sum_{i,j} A_{ij}\,\delta(c_i, c_j)$ relates to the actual edges observed, while the Resolution component $Q_{\text{res}} = \gamma \sum_{i,j} \frac{k_i k_j}{2m}\,\delta(c_i, c_j)$ estimates what we would expect from degree-based connections.

4.1.2 Modularity Gain Calculation. When a node i moves from one community to another, we compute the net modularity gain ΔQ as a difference of insertion and removal terms. Specifically, if node i moves to a new community c_i', the insertion gain ΔQ_{in} (i.e., adding i to c_i') and the removal loss ΔQ_{out} (i.e., removing i from its old community) combine into $\Delta Q = \Delta Q_{\text{in}} - \Delta Q_{\text{out}}$.

Insertion Gain. When adding node i to community c_i', the affinity term $\Delta Q_{\text{in,aff}}$ can be written (in vector form) as $v_i \cdot \sum_{j \in c_i'} v_j$. The resolution term involves $\sum_{j \in c_i'} k_j$, multiplied by $\gamma\,\frac{k_i}{2m}$.

Removal Loss. Upon leaving its original community c_i, node i drops its affinity to members of c_i. We subtract $v_i \cdot \sum_{j \in c_i} v_j$, adjusted for the self-loop $v_i \cdot v_i$, and similarly correct the resolution term based on $\sum_{j \in c_i} k_j$.

4.2 Community Movement Optimization

In the Louvain procedure, each node looks for a community that maximizes its ΔQ. Although evaluating ΔQ for every possible community can be expensive, maintaining node and community vectors can streamline this step. Initially, each node i starts in its own community c_i. We assign the node's feature vector v_i and degree k_i to that community's aggregates:

$$V^{c_i} \leftarrow v_i, \quad K^{c_i} \leftarrow k_i,$$

where V^c and K^c respectively represent the summed feature vector and the total degree of a community c. Hence, each community is simply the node's own vector and degree at the start. During optimization, when node i moves from community c to community c', these aggregated vectors are updated:

$$V^c \leftarrow V^c - v_i, \quad K^c \leftarrow K^c - k_i;$$
$$V^{c'} \leftarrow V^{c'} + v_i, \quad K^{c'} \leftarrow K^{c'} + k_i.$$

Because ΔQ computations depend only on these updated aggregates, there is no need to enumerate edges individually. By storing V^c and K^c for each community rather than the entire adjacency structure, we preserve Louvain's modularity gain procedure while significantly reducing space requirements, keeping the memory overhead at $O(n \cdot d)$ instead of $O(n^2)$.

5 VLouvain Algorithm

In this section, we present our optimized implementation of the Louvain algorithm in the vector setting. The VLouvain algorithm consists of two main components: the optimized one-level function and the optimized new graph generation function. It then combines these components into an iterative process that refines community assignments until no further modularity gain is observed.

5.1 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 (see Algorithm 1). At the beginning of this function, the node vector matrix $\mathbf{V}$ and the degree array $\mathbf{K}$ are copied into community-specific matrices $\mathbf{VC} \leftarrow \mathbf{V}$ and $\mathbf{KC} \leftarrow \mathbf{K}$ as each node starts in its own community, captured by the initial community assignment $\mathbf{C}$. Here, $\mathbf{C}$ is a vector that tracks each node's community membership, and c_i denotes the community of node i.

The procedure then sets $\Delta Q \leftarrow 0$. For each node i, based on the explanation in section 4.1, the algorithm calculates $q_{\text{out}} \leftarrow \frac{\mathbf{v}_i \cdot \mathbf{VC}[i] + \frac{\gamma k_i}{2m}\mathbf{KC}[i]}{2m}$, representing the modularity effect of removing i from its existing community. Next, it computes a vector $\mathbf{q}_{\text{in}}$, where each entry corresponds to the modularity effect of placing i into the community of another node j: $\mathbf{q}_{\text{in}} \leftarrow \frac{\mathbf{v}_i \cdot \mathbf{VC} + \frac{\gamma k_i}{2m}\mathbf{KC}}{2m}$. Subtracting q_{out} from each element of $\mathbf{q}_{\text{in}}$ yields $\mathbf{q}$, which captures the net gain of moving i. The algorithm then finds

$$j = \arg\max_j \mathbf{q}[j] \quad \text{subject to} \quad \mathbf{q}[j] > 0,$$

indicating the best alternative community among those that offer a positive gain. If $c_j \neq c_i$, the node i is reassigned to community c_j. In this process, $\mathbf{VC}$ and $\mathbf{KC}$ are updated by subtracting $\mathbf{v}_i$ and k_i from the old community's entries and adding them to the new community's entries. Concretely, the algorithm locates all nodes belonging to the old community c_i (using Boolean indexing by $\mathbf{C} == c_i$) and subtracts $\mathbf{v}_i$ and k_i from those entries in $\mathbf{VC}$ and $\mathbf{KC}$. It then locates the nodes in the new community c_j and adds $\mathbf{v}_i$ and k_i to the corresponding entries, so the aggregates accurately represent the removal of i from c_i and its addition to c_j.

The algorithm also increases modularity gain ΔQ by $\mathbf{q}[j]$. After all nodes have been processed, the updated community assignments $\mathbf{C}$, matrices $\mathbf{VC}$ and $\mathbf{KC}$, and the accumulated ΔQ are returned.

Within each level we repeat node updates and stop when the sweep-level modularity gain falls below a small threshold (e.g., $\varepsilon = 10^{-7}$). After the first pass most nodes stabilize, so we use a batched check that tests in parallel whether any node in a block has a positive-gain move—skipping whole blocks with no movers—which makes later passes extremely fast due to GPU vector operations.

Since these 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 $O(n \cdot d)$ space, which is far more efficient than the $O(n^2)$ cost of explicitly managing all edges.

5.2 Optimized New Graph Generation

Upon completing the first phase (the one-level function), the algorithm uses the community assignment array $\mathbf{C}$ to collapse

Algorithm 1 Optimized One-Level Function (OOLF)

Input: Matrix $\mathbf{V}$, array $\mathbf{K}$, initial communities $\mathbf{C}$, resolution γ, total edge weight m, tolerance $\varepsilon > 0$
Output: Updated $\mathbf{C}$, matrices $\mathbf{VC}$, $\mathbf{KC}$, modularity gain ΔQ

Initialize $\mathbf{VC} \leftarrow \mathbf{V}$, $\mathbf{KC} \leftarrow \mathbf{K}$, $\Delta Q \leftarrow 0$
while true **do**
  $\Delta Q_{\text{sweep}} \leftarrow 0$
  for each node i **do**
    $q_{\text{out}} \leftarrow \frac{v_i \cdot \mathbf{VC}[i] + \gamma \frac{k_i}{2m} \mathbf{KC}[i]}{2m}$
    Compute $\mathbf{q}_{\text{in}} \leftarrow \frac{\left(v_i \cdot \mathbf{VC} + \gamma \frac{k_i}{2m} \mathbf{KC}\right)}{2m}$
    Define $\mathbf{q} \leftarrow \mathbf{q}_{\text{in}} - q_{\text{out}}$
    $j \leftarrow \arg\max_j \mathbf{q}[j] \quad \text{s.t. } \mathbf{q}[j] > 0$
    if $c_j \neq c_i$ **then**
      Update $\mathbf{VC}, \mathbf{KC}$ for c_i and c_j:
      $\mathbf{VC}[\mathbf{C} == c_i] \mathrel{-}= v_i, \quad \mathbf{KC}[\mathbf{C} == c_i] \mathrel{-}= k_i$
      $\mathbf{VC}[\mathbf{C} == c_j] \mathrel{+}= v_i, \quad \mathbf{KC}[\mathbf{C} == c_j] \mathrel{+}= k_i$
      $\mathbf{VC}[i] \leftarrow \mathbf{VC}[j], \quad \mathbf{KC}[i] \leftarrow \mathbf{KC}[j]$
      $c_i \leftarrow c_j, \quad \Delta Q \mathrel{+}= \mathbf{q}[j]$
      $\Delta Q_{\text{sweep}} \mathrel{+}= \mathbf{q}[j]$
  if $\Delta Q_{\text{sweep}} \leq \varepsilon$ **then**
    break
return $\mathbf{C}$, $\mathbf{VC}$, $\mathbf{KC}$, ΔQ

Algorithm 2 Optimized New Graph Generation (ONGG)

Input: Community assignments $\mathbf{C}$, matrices $\mathbf{VC}$, $\mathbf{KC}$
Output: $\mathbf{V}^{\text{new}}, \mathbf{K}^{\text{new}}$

$(\mathbf{C}^{\text{unique}}, \mathbf{I}^*) \leftarrow \text{unique}(\mathbf{C}, \text{return\_inverse=True})$
for each unique community index $c \in \mathbf{C}^{\text{unique}}$ **do**
  Identify node indices: $\mathbf{I}_c \leftarrow \mathbf{I}^*[\mathbf{C} == c]$
  Choose a representative index $i_c \in \mathbf{I}_c$ (e.g., the first element of $\mathbf{I}_c$)
  $\mathbf{V}^{\text{new}}[c] \leftarrow \mathbf{VC}[i_c]$
  $\mathbf{K}^{\text{new}}[c] \leftarrow \mathbf{KC}[i_c]$
return $\mathbf{V}^{\text{new}}, \mathbf{K}^{\text{new}}$

Algorithm 3 Optimized Louvain Partition Algorithm

Input: Matrix $\mathbf{V}$, resolution parameter γ, threshold ϵ
Output: PartitionList

Compute degrees $\mathbf{K}$ and total edge weights m
Initialize PartitionList $\leftarrow$ empty list
repeat
  Initialize community assignments $\mathbf{C}$, where each node is in a community by itself.
  $\mathbf{C}, \mathbf{VC}, \mathbf{KC}, \Delta Q \leftarrow \text{OOLF}(\mathbf{V}, \mathbf{K}, \mathbf{C}, \gamma)$
  if $\Delta Q > \epsilon$ **then**
    $\mathbf{V}^{\text{new}}, \mathbf{K}^{\text{new}} \leftarrow \text{ONGG}(\mathbf{C}, \mathbf{VC}, \mathbf{KC})$
    Update $\mathbf{V} \leftarrow \mathbf{V}^{\text{new}}, \mathbf{K} \leftarrow \mathbf{K}^{\text{new}}$
    Append $\mathbf{C}$ to PartitionList
until $\Delta Q \leq \epsilon$
return PartitionList

nodes within the same community into a single supernode (Algorithm 2). This step preserves the total edge weight between original communities. The process begins by identifying all unique communities in $\mathbf{C}$, then aggregating the vector representations $\mathbf{VC}$ and the degree array $\mathbf{KC}$ for nodes belonging to each community. Summing the rows in $\mathbf{VC}$ for nodes in one community produces the vector representation for the corresponding supernode while summing the corresponding entries in $\mathbf{KC}$ yields that supernode's degree. Because each new edge weight between supernodes C_1 and C_2 is $\sum_{i \in C_1, j \in C_2} A_{ij} = \left(\sum_{i \in C_1} v_i\right) \cdot \left(\sum_{j \in C_2} v_j\right)$, a simple summation of the vectors in each community suffices to reconstruct community-level connections.

We use $\mathbf{I}$ to denote the global set of node indices, and $\mathbf{C}$, as before, to store the community to which each node belongs. 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. This index collection allows vector operations on $\mathbf{VC}$ and $\mathbf{KC}$ only for the nodes belonging to each community. We store the aggregated results in $\mathbf{V}^{\text{new}}$ and $\mathbf{K}^{\text{new}}$, where each row corresponds to a supernode.

Mathematical Justification.
Defining $A_{C_1C_2} = \sum_{i \in C_1, j \in C_2} A_{ij}$ and using $A_{ij} = v_i \cdot v_j$ yields $A_{C_1C_2} = \left(\sum_{i \in C_1} v_i\right) \cdot \left(\sum_{j \in C_2} v_j\right)$. Hence, summing vectors within a community precisely yields the supernode representation that captures the correct total edge weight.

5.3 Optimized Louvain Partition Algorithm

The Optimized Louvain Partition Algorithm refines community assignments by iterating between the Optimized One-Level Function (OOLF) and the Optimized New Graph Generation (ONGG). After computing the degree of each node $\mathbf{K}$ and the total sum of edge weights m, the algorithm initializes community assignments $\mathbf{C}$, sets a resolution parameter γ and a threshold ϵ, and optionally maintains a partition list PartitionList.

During each iteration, OOLF runs to update $\mathbf{C}, \mathbf{VC}, \mathbf{KC}$, and computes the total modularity gain ΔQ. If $\Delta Q > \epsilon$, the ONGG is invoked to produce $\mathbf{V}^{\text{new}}$ and $\mathbf{K}^{\text{new}}$, which replace $\mathbf{V}$ and $\mathbf{K}$. The current community assignments can be appended to PartitionList. If $\Delta Q \leq \epsilon$, the algorithm terminates.

5.4 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 $v_i \cdot \mathbf{V}^c$ of cost $O(n \cdot d)$ plus a scalar multiplication for the resolution term, $\gamma \frac{k_i}{2m} \mathbf{K}^c$, of cost $O(n)$. Combining these yields $O(n \cdot d)$ work per node and $O(n^2 \cdot d)$ overall.

Although the worst-case time complexity is $O(n^2 \cdot d)$ due to inner-product evaluations, in practice $d \ll n$ (e.g., $d = 200$), making the cost effectively comparable to $O(n^2)$. Moreover, the computation is tileable and bandwidth-bound on GPUs; we reuse tiles across local-move evaluations and cache community aggregates to reduce redundant work.

The memory cost includes $\mathbf{V}$ of size $n \times d$, $\mathbf{K}$ of size n, their community copies $\mathbf{V}^c, \mathbf{K}^c$, and the assignment array $\mathbf{C}$. This corresponds to about $2(n \times d) + 3n$ overall, or $O(n \cdot d)$. This figure remains considerably below the $O(n^2)$ storage, making VLouvain effective for large, low-rank graphs when the embedding dimension d is much smaller than n.

6 Experiments

All experiments were conducted using Google Colab Pro with A100 GPU instances. The system specifications include 83.5 GB of RAM, 40 GB of GPU memory, and a total disk space of 235.7 GB. Our implementation is in Python using PyTorch https://github.com/yutengkai/VLouvain.

Datasets. We evaluated our approach on four datasets from PyTorch Geometric[2] [9], summarized in Table 1. The Amazon Products, Flickr, and Yelp datasets are from GraphSAINT [28] and include precomputed node 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). We used these features as-is; VLouvain treats the embedding space as fixed input, and any representational limitations apply equally across all methods. The embedding dimension d is fixed by the upstream embedding process and lies outside VLouvain's control.

For Taobao, starting with a heterogeneous graph of users, items, and categories, we constructed direct user-category links via TF-IDF over category usage (treating users as documents). We retained 66 categories with at least 62,682 user interactions (15% of the most frequent one). Each user was then represented by a 66-dimensional TF-IDF vector over these categories.

Table 1: Summary of Datasets

Dataset	n	Dimensionality
Flickr	89,249	500
Yelp	716,777	300
Taobao	936,946	66
Amazon Products	1,569,960	200

For all datasets, we use the normalized cosine similarity as edge weights, defined as $A_{ij} = \frac{1+(v_i \cdot v_j)}{2}$, where v_i and v_j are unit-normalized feature vectors. This is equivalent to the inner product of augmented vectors $\mathbf{v}'_i = \frac{[v_i;1]}{\sqrt{2}}$, so that edge weights are non-negative and compatible with modularity optimization in the Louvain algorithm.

Runtime Data Collection. We compare our approach against cuGraph [8], GVE [21], iGraph [5], and NetworKit [23]. These methods were chosen because they are popular, efficient implementations of community detection: cuGraph and GVE leverage GPU acceleration for large-scale graphs, while iGraph and NetworKit use optimized CPU-based algorithms and data structures. For all methods, we include both the graph construction phase and the community detection step to provide a fair runtime comparison. For GVE, 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" to reflect its actual runtime cost. Thus, we align runtime measurements across all methods, comparing the full path from input vectors to final clustering.

For all methods, we use $\gamma = 1$ and $\epsilon = 10^{-7}$, which are the default parameter values in several Louvain implementations.

Performance Across Graph Sizes. Our results for the two large datasets, Amazon and Taobao, are shown in Tables 2 and 3. We observed similar trends for the other two datasets, Flickr and Yelp. Due to space constraints, we omit detailed tables for Yelp and Flickr but summarize their key results in this section. In our setting, the resulting graphs are almost complete, with edges between nearly all pairs of nodes. This observation holds true even for datasets like Taobao, where we use TF-IDF and original cosine similarities. We examine how each method scales by creating subgraphs that include a fraction of the nodes from the original graph. To compare performance across datasets with varying sizes and structures, we sample subgraphs using different percentage thresholds for each dataset. The sampling thresholds for the first five entries in the tables are chosen to create subgraphs with (loosely) similar numbers of nodes, allowing for reasonably fair runtime comparisons across datasets. This way we account for the fact that fixed sampling percentages in the initial entries would result in significantly different graph sizes across datasets, leading to imbalanced comparisons. The last two entries, however, are fixed at 50% and 100% of the dataset size to evaluate performance at higher scales regardless of dataset size.

We record runtime at each fraction. As most graph-based methods store explicit edges, they frequently encounter memory limits, especially above about 15,000 nodes for GPU-based libraries like cuGraph. Consequently, smaller subgraphs (ranging up to several thousand or around 15,000 nodes) allow comparisons among all methods, while larger fractions typically surpass memory constraints for competing approaches. After those methods fail, we continue with our method at 50% and 100% of the graph to demonstrate its ability to handle large-scale data. Each trial is repeated five times, and we use the average runtime for consistent reporting.

Observations from Experimental Results. For smaller graphs, VLouvain performs comparably to existing methods, with small variations depending on the dataset. However, as the graph size increases, particularly at thresholds of 50% or above, VLouvain becomes the only method that can complete execution.

In the Amazon dataset, once the graph exceeds 784,980 nodes (50% threshold), VLouvain alone handles the load, finishing in about 3,800 seconds, and it also completes the full 1,569,960-node graph in roughly 11,200 seconds (See Table 2). The Taobao dataset shows the same trend; by 50% threshold (468,473 nodes), VLouvain finishes in about 1600 seconds, and at the full 936,946-node graph, in about 3800 seconds—again, the only successful method (See Table 3).

A similar pattern appears in the Yelp dataset: at 358,388 nodes (50% threshold), VLouvain processes it in about 1700 seconds, while all others fail; at the full 716,777-node scale, it completes in about 4,800 seconds. Likewise, the Flickr dataset hits a point at 44,624 nodes (50% threshold) where VLouvain alone succeeds (about 150 seconds), and it proceeds to handle the entire 89,249-node graph in roughly 320 seconds. These results show that VLouvain efficiently handles large graph datasets by scaling to graphs with millions of nodes where others fail due to memory or computational limits.

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 limits.

Normalized Cosine. As empirical support, normalized cosine similarity also underlies our GraphRAG-V system for retrieval via text-chunk communities [27], where VLouvain clustered semantically related passages for large language model retrieval. On MultiHopRAG, this approach improved recall from 37.9% to

[2]https://pytorch-geometric.readthedocs.io/en/2.6.0/modules/datasets.html

Table 2: Performance Comparison for Amazon Products Dataset

threshold	n	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
50.0%	784,980	**3,725.28**	X	X	X	X
100.0%	1,569,960	**11,297.10**	X	X	X	X

Table 3: Performance Comparison for Taobao Dataset

threshold	n	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
50.0%	468,473	**1,585.58**	X	X	X	X
100.0%	936,946	**3,764.42**	X	X	X	X

Table 4: Exact and Approx. Top K performance on Amazon.

	Construction (s)		Louvain (s)		Edges (M)		Communities		NMI wrt VL	
K	Exact	Approx	Exact	Approx	Exact	Approx	Exact	Approx	Exact	Approx
32	2423	252	6.35	7.41	40.3	41.8	38	35	0.032	0.044
64	2395	248	21.85	32.49	79.0	82.8	33	37	0.035	0.033
128	2471	256	26.33	46.95	154.5	164.3	20	25	0.034	0.029
256	2470	264	90.21	90.48	301.9	326.9	18	19	0.046	0.039

Notes: Construction (s) = Top K build (Exact: IndexFlatIP, Approx: IVF train+add+search+build).

48.8% (over Microsoft RAG) and increased exact-match accuracy by four points, confirming its practical validity.

7 Top-K Similarity Graph

We compare **VLouvain on the full similarity graph** to *Exact Top-K* (FAISS `IndexFlatIP`, exact inner product search) and *Approximate Top-K* (FAISS `IndexIVFFlat`, inverted lists for approximate search). Our goal is to assess runtime feasibility and partition fidelity of the Top-K approaches relative to the full graph.

Methodology.

Louvain implementations (iGraph, cuGraph, VLouvain, etc.) produce nearly identical partitions on a fixed graph (same community count; modularity within 10^{-4}), so we use cuGraph as representative. On the Amazon dataset, we take a half-graph slice (784,980 nodes), construct Top-K graphs for $K \in \{32, 64, 128, 256\}$ using FAISS (Exact and Approximate), run cuGraph, and compare each to the VLouvain full graph via **Normalized Mutual Information (NMI)** [6]. Table 4 summarizes construction time, Louvain time, edge counts, community counts, and NMI.

Findings. For both methods, construction is slower than Louvain itself. The runtimes of different implementations of Louvain are similar and dominated by construction, so we report only cuGraph. The main result, however, is that both methods yield partitions essentially unrelated to the full-graph VLouvain solution (NMI ≈ 0.04), showing that limiting to top-K edges discards much of the original structure, so speed gains come at a large loss in fidelity.

8 Application to GraphRAG

Retrieval-augmented generation (RAG) is a widely adopted approach for enabling large language models (LLMs) to access external, dynamic, or private corpora without retraining. Traditional RAG systems retrieve semantically similar passages based on dense embeddings and pass them to the LLM at inference time. A recent popular variant, **GraphRAG**, by Microsoft [7], extends this strategy by grouping passages into *communities* based on an LLM-generated graph of entities and relations. This allows more effective multi-hop reasoning and context-aware retrieval.

However, such enhancement comes at a significant cost. Microsoft's implementation relies on LLM calls to extract entities and infer edges with weights between them, which are then materialized into a dense graph containing up to $O(n^2)$ edges. This is not only computationally expensive but also financially intensive: anecdotal reports cite thousands of dollars in API costs for medium-scale corpora [1]. Moreover, the edge weighting is LLM-based and thus stochastic, introducing variability between runs.

We modify this pipeline using **VLouvain**, replacing the LLM-derived graph with a similarity graph computed directly from passage embeddings. Each passage becomes a node; edge weights are derived from cosine similarities. VLouvain discovers communities directly from the embedding matrix, maintaining only $O(nd)$ state and avoiding any explicit graph construction, edge pruning, or entity linking. Our implementation, detailed in [27] matches GraphRAG's local-global retrieval logic, while simplifying the pipeline.

Evaluated on MultiHopRAG [24], our method builds the index in **5.3 minutes**, nearly **100× faster** than GraphRAG's three-hour graph construction. QA runs in **42 minutes** on a single A100, with retrieval recall improving from 37.9% to 48.8%, and exact-match accuracy up four points. Token usage drops by two orders of magnitude (see [27] for details).

9 Artifacts

The source code, data, and/or other artifacts have been made available at https://github.com/yutengkai/VLouvain.

References

[1] Irina Adamchic. 2025. Build your hybrid-Graph for RAG & GraphRAG applications using the power of NLP. https://medium.com/@irina.karkkanen/build-your-hybrid-graph-for-rag-graphrag-applications-using-the-power-of-nlp-57219b6e2adb Accessed: 2025-06-01.

[2] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment* 2008, 10 (2008), P10008.

[3] Mingming Chen, Konstantin Kuzmin, and Boleslaw K Szymanski. 2014. Community detection via maximization of modularity and its variants. *IEEE Transactions on Computational Social Systems* 1, 1 (2014), 46–65.

[4] Aaron Clauset, Mark EJ Newman, and Cristopher Moore. 2004. Finding community structure in very large networks. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics* 70, 6 (2004), 066111.

[5] Gabor Csardi and Tamas Nepusz. 2006. The igraph software. *Complex syst* 1695 (2006), 1–9.

[6] Leon Danon, Albert Diaz-Guilera, Jordi Duch, and Alex Arenas. 2005. Comparing community structure identification. *Journal of statistical mechanics: Theory and experiment* 2005, 09 (2005), P09008.

[7] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, and Jonathan Larson. 2024. From local to global: A graph rag approach to query-focused summarization. *arXiv preprint arXiv:2404.16130* (2024).

[8] Alex Fender, Brad Rees, and Joe Eaton. 2022. Rapids cugraph. In *Massive Graph Analytics*. Chapman and Hall/CRC, 483–493.

[9] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).

[10] Santo Fortunato. 2010. Community detection in graphs. *Physics reports* 486, 3-5 (2010), 75–174.

[11] Fabio Gasparetti, Alessandro Micarelli, and Giuseppe Sansonetti. 2018. Community Detection and Recommender Systems.

[12] Fabio Gasparetti, Giuseppe Sansonetti, and Alessandro Micarelli. 2021. Community detection in social recommender systems: a survey. *Applied Intelligence* 51, 6 (2021), 3975–3995.

[13] Sayan Ghosh, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, Hao Lu, Daniel Chavarria-Miranda, Arif Khan, and Assefaw Gebremedhin. 2018. Distributed louvain algorithm for graph community detection. In *2018 IEEE international parallel and distributed processing symposium (IPDPS)*. IEEE, 885–895.

[14] Michelle Girvan and Mark EJ Newman. 2002. Community structure in social and biological networks. *Proceedings of the national academy of sciences* 99, 12 (2002), 7821–7826.

[15] Vassilis N Ioannidis, Ahmed S Zamzam, Georgios B Giannakis, and Nicholas D Sidiropoulos. 2019. Coupled graphs and tensor factorization for recommender systems and community detection. *IEEE Transactions on Knowledge and Data Engineering* 33, 3 (2019), 909–920.

[16] Andrea Lancichinetti and Santo Fortunato. 2009. Community detection algorithms: a comparative analysis. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics* 80, 5 (2009), 056117.

[17] Mark EJ Newman. 2006. Modularity and community structure in networks. *Proceedings of the national academy of sciences* 103, 23 (2006), 8577–8582.

[18] Xinyu Que, Fabio Checconi, Fabrizio Petrini, and John A Gunnels. 2015. Scalable community detection with the louvain algorithm. In *2015 IEEE international parallel and distributed processing symposium*. IEEE, 28–37.

[19] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics* 76, 3 (2007), 036106.

[20] Martin Rosvall and Carl T Bergstrom. 2008. Maps of random walks on complex networks reveal community structure. *Proceedings of the national academy of sciences* 105, 4 (2008), 1118–1123.

[21] Subhajit Sahu. 2023. GVE-Louvain: Fast Louvain Algorithm for Community Detection in Shared Memory Setting. *arXiv preprint arXiv:2312.04876* (2023).

[22] Zeinab Shokrzadeh, Mohammad-Reza Feizi-Derakhshi, Mohammad-Ali Balafar, and Jamshid Bagherzadeh Mohasefi. 2023. Graph-Based Recommendation System Enhanced by Community Detection. *Scientific Programming* 2023, 1 (2023), 5073769.

[23] Christian L Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. 2014. NetworKit: A tool suite for high-performance network analysis. *Network Science* (2014).

[24] Yixuan Tang and Yi Yang. 2024. Multihop-rag: Benchmarking retrieval-augmented generation for multi-hop queries. *arXiv preprint arXiv:2401.15391* (2024).

[25] Vincent A Traag, Ludo Waltman, and Nees Jan Van Eck. 2019. From Louvain to Leiden: guaranteeing well-connected communities. *Scientific reports* 9, 1 (2019), 1–12.

[26] Jaewon Yang and Jure Leskovec. 2012. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD workshop on mining data semantics*. 1–8.

[27] Tengkai Yu, Venkatesh Srinivasan, and Alex Thomo. 2026. GraphRAG-V: Fast Multi-hop Retrieval via Text-Chunk Communities. In *Social Networks Analysis and Mining*. Springer, 3–14.

[28] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2019. Graphsaint: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931* (2019).