OST
Ostschweizer
Fachhochschule

1 Graph-RAG

Master of Science in Engineering
HS 2025

Reader — Machine Learning im Modul FTP_MachLe (WUCH)

Inhaltsverzeichnis

Standard Retrieval-Augmented Generation (RAG) systems rely heavily on vector embeddings to retrieve relevant context. While effective for local semantic queries, these systems often fail to capture global structural information and multi-hop relationships across large corpora. This article introduces **Graph RAG**, a paradigm shift that integrates Knowledge Graphs (KGs) into the retrieval pipeline. We explore the architectural differences, the mathematical formulation of graph-based retrieval, and the Community Summary approach for global reasoning.

1 Introduction to RAG Systems

Large Language Models (LLMs) are powerful reasoning engines, but they suffer from two critical limitations: they are "frozen in time" (limited to their training data cutoff) and they lack access to proprietary or private knowledge. Retrieval-Augmented Generation (RAG) is a hybrid architecture designed to bridge this gap.

RAG fundamentally alters the interaction with an LLM by shifting from a purely generative paradigm to a *Retrieve-then-Generate* workflow. When a user submits a query, the system does not immediately send it to the LLM. Instead, it first acts as an intelligent librarian, searching a vast dataset to retrieve relevant text chunks. These chunks are then appended to the original query as "context." The LLM is instructed to answer the question using *only* the provided information, thereby grounding the response in factual data and significantly reducing hallucinations.

To retrieve relevant information, the system must understand the "meaning" of both the user's query and the stored documents. Since computers cannot inherently understand language, we must translate text into a mathematical format. This is achieved through an **Embedding Model**, a neural network trained to map discrete text inputs into continuous numerical representations.

Formally, an embedding function f_θ transforms a text sequence T into a vector $\mathbf{v}$ within a high-dimensional vector space $\mathbb{R}^d$:

$$f_\theta(T) \to \mathbf{v} \in \mathbb{R}^d \tag{1}$$

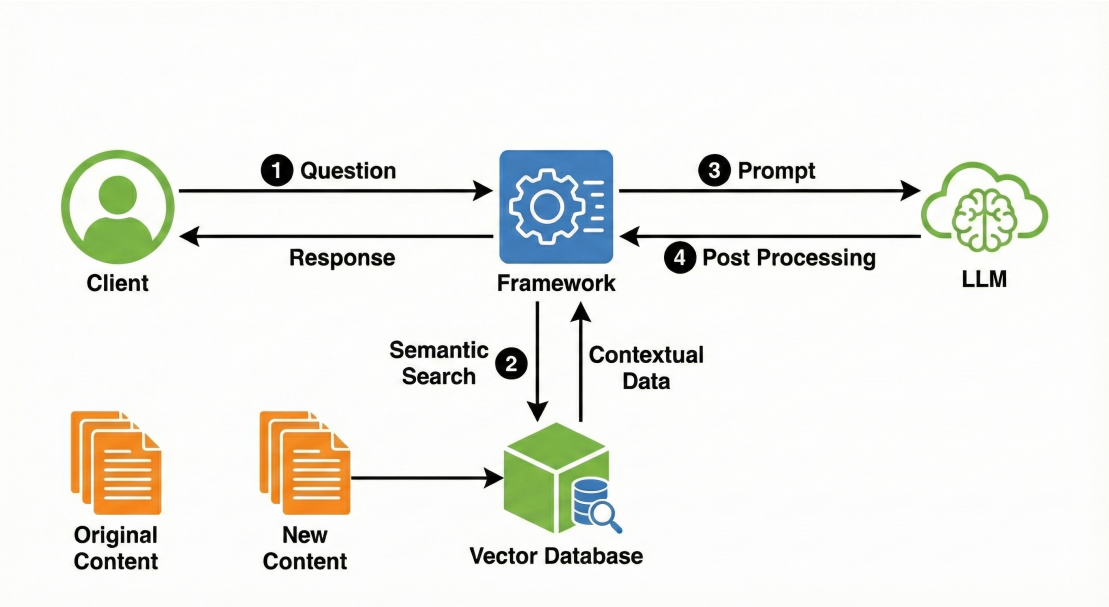

Abbildung 1: Retrieval Augmented Generation (RAG) architecture

The dimensionality d is typically high (e.g., 768 or 1536 dimensions), allowing the model to capture subtle nuances of language.

The resulting vector space is **semantic**, meaning that the geometric position of a vector encodes its conceptual meaning. In this space, distance correlates with semantic relatedness. For example, the mathematical vector for "Apple" will be positioned closer to "Fruit" and "Technology" than to "Car," depending on the context. This allows the system to identify relevant documents even if they do not share the exact keywords as the query, a significant improvement over traditional keyword search.

Once the query and the documents are mapped to this high-dimensional space, the retrieval task becomes a geometry problem. We need to find the document vectors that are "closest" to the query vector.

While Euclidean distance is common in 2D space, it is often suboptimal for text embeddings because the magnitude (length) of the vector can be affected by the length of the text chunk. Instead, we use **Cosine Similarity**, which measures the cosine of the angle θ between two vectors. This metric focuses purely on the *orientation* of the

vectors—i.e., whether they are pointing in the same semantic direction.
Given a query vector $\mathbf{q}$ and a document vector $\mathbf{d}$, the similarity is calculated as:

$$\text{similarity}(\mathbf{q}, \mathbf{d}) = \cos(\theta) = \frac{\mathbf{q} \cdot \mathbf{d}}{\|\mathbf{q}\|\|\mathbf{d}\|} \tag{2}$$

A result of 1 indicates identical semantic meaning (vectors overlap), while a result of 0 indicates orthogonality (completely unrelated concepts).

1.1 Scalability and Fast Vector Search

In a production environment containing millions or billions of text chunks, calculating the Cosine Similarity between the query and *every* stored vector (a linear scan, $O(N)$) is computationally prohibitive. To achieve low-latency retrieval, we employ **Approximate Nearest Neighbor (ANN)** algorithms. The industry standard for this is **HNSW (Hierarchical Navigable Small World)**. Rather than scanning every point, HNSW organizes vectors into a multi-layered graph structure, analogous to a map with both highways and local roads.

- The algorithm begins at the top layers ("highways"), where connections are sparse and span long distances, allowing the search to quickly traverse the vector space to the general neighborhood of the query.
- Once in the vicinity, the algorithm descends to lower layers ("local roads"), where connections are dense, enabling a precise search for the exact nearest neighbors.

This hierarchical approach reduces the search complexity from linear time to logarithmic time ($O(\log N)$), enabling the retrieval of the most semantically relevant information from massive datasets in milliseconds.

1.2 The Limitations of Baseline RAG

In a standard RAG pipeline, a corpus $\mathcal{D}$ is split into chunks $\{c_1, c_2, ..., c_n\}$. An embedding model f_θ maps these chunks to a vector space $\mathbb{R}^d$. Retrieval is performed via Approximate Nearest Neighbor (ANN) search:

Similarity

$$c_{\text{retrieved}} = \underset{c \in \mathcal{D}}{\operatorname{argmax}} \cos(f_\theta(q), f_\theta(c)) \tag{3}$$

where q is the query. This approach suffers from two primary deficits:

- **Context Fragmentation:** The model loses the connection between chunk c_i and chunk c_j if they are not semantically similar, even if they are logically related (e.g., via a causal chain).
- **Lack of Global Overview:** Vector RAG cannot effectively answer holistic questions (e.g., What are the changing political themes in this dataset?) because the answer is not contained in any single chunk.

2 Graph RAG Architecture

Graph RAG augments the retrieval process by constructing a directed graph $G = (V, E)$ from the source text, where V represents entities and E represents relationships.

2.1 The Indexing Phase: LLM-Driven Topology Construction

The indexing phase in Graph RAG represents a significant departure from the computationally lightweight embedding process of standard Vector RAG. While vector systems simply map text to numbers, Graph RAG acts as a structured reasoning engine that transforms unstructured text into a deterministic network topology. This process is heavily reliant on Large Language Models (LLMs) to function as intelligent extractors. The process begins by dividing the corpus into chunks, similar to standard approaches. However, rather than immediately embedding these chunks, the system passes each text chunk c_i to an LLM with a specific prompt instruction: identifying the core entities (nodes) and the relationships (edges) binding them. Unlike traditional Knowledge Graph construction, which often relies on rigid, pre-defined ontologies or fragile regular expressions, the LLM approach allows for an "open schema," dynamically discovering entity types relevant to the specific domain.

Formally, for a given text chunk c_i, the extraction function yields a set of triples:

$$\text{Extract}(c_i) \rightarrow \{(h, r, t) \mid h, t \in V, r \in E\} \tag{4}$$

where h is the head entity (source), t is the tail entity (target), and r represents the specific semantic relationship between them.
However, a graph consisting solely of sparse triples (e.g., *Einstein, studied, Physics*) often fails to capture the richness of the source text. A strictly topological graph loses the subtle context of *how* or *why* a relationship exists. To address this, modern Graph RAG implementations (such as the methodology proposed by Microsoft Research) introduce a layer of natural language annotation.
In this enhanced indexing workflow, the LLM generates a dense natural language description, denoted as D_v, for every identified node and edge. For example, rather than simply linking "Company A" to "Product B," the edge attribute might contain a summary paragraph explaining that "Company A acquired the rights to Product B following a hostile takeover in 2023." This hybrid approach preserves the nuanced semantic information that strict triples typically discard, ensuring that the graph retains the fidelity of the original text while providing the structural advantages of network analysis.

2.2 Hierarchical Clustering and Community Detection

To enable global search, the graph G must be partitioned. We employ modularity-based community detection algorithms, such as the **Leiden Algorithm**.
Let $\mathcal{P} = \{C_1, ..., C_k\}$ be a partition of V. The Leiden algorithm maximizes modularity Q:

Modularity

$$Q = \frac{1}{2m} \sum_{ij} \left(A_{ij} - \frac{k_i k_j}{2m} \right) \delta(c_i, c_j) \tag{5}$$

where A_{ij} is the adjacency matrix weight, k_i is the degree of node i, m is the total edge weight, and δ is the Kronecker delta.
This process creates a hierarchical structure of communities. The system then uses an LLM to generate a **summary** for each community C_k, creating a high-level index of the data's topology.

3 Retrieval Modalities

Graph RAG introduces dual retrieval capabilities that standard RAG lacks.

3.1 Local Search (Multi-hop Reasoning)

For queries requiring specific details about an entity (e.g., How is Algorithm A related to Researcher B?), the system:

(a) Identifies entry nodes in G based on query entities.

(b) Traverses edges E (e.g., via Breadth-First Search or personalized PageRank) to find connected concepts k hops away.

(c) Aggregates the text descriptions of the traversed path into the context window.

3.2 Global Search (Map-Reduce Summarization)

For broad queries (e.g., What are the main conflicts in the story?), standard RAG fails because the answer exists in the *aggregate*, not in a chunk. Graph RAG solves this by:

(a) **Map:** Selecting community summaries from the hierarchical level appropriate for the query's granularity.

(b) **Reduce:** Feeding these summaries to the LLM to synthesize a global answer.

3.3 Computational Complexity and Trade-offs

While Graph RAG significantly improves sense-making capabilities, it introduces latency and cost.

Metric	**Standard RAG**	**Graph RAG**
Indexing Cost	$O(N)$ (Embedding)	$O(N \times \text{LLM}_{\text{extract}})$ (High)
Query Latency	Milliseconds (ANN Search)	Seconds (Graph Traversal/Map-Reduce)
Recall	High for explicit similarity	High for implicit/structural links

Tabelle 1: Comparison of Efficiency vs. Capability

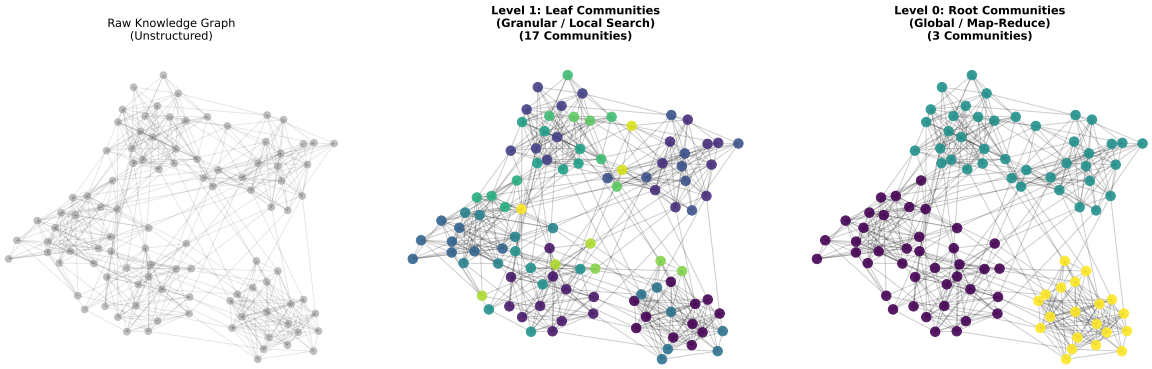

Abbildung 2: **Center Plot** (Level 1): With a high resolution (1.5), the algorithm detects tight, smaller communities. In Graph RAG, the LLM would summarize these to answer questions like "What specific protocol caused the error?". **Right Plot** (Level 0): With a low resolution (0.5), the algorithm merges those smaller communities into broader super-communities. In Graph RAG, the LLM summarizes these to answer " What are the overarching themes in this dataset? "

Graph RAG represents the convergence of symbolic AI (Knowledge Graphs) and connectionist AI (LLMs). It addresses the reasoning gap in standard RAG. Future work involves *Graph-Vector Hybrids*, where retrieval is performed simultaneously in vector space and graph space to maximize both precision and breadth.

4 Hierarchical Clustering and the Leiden Algorithm

To enable the Global Search capability—wherein the system answers queries regarding broad themes across the entire corpus—the Knowledge Graph $G = (V, E)$ must be organized into a semantic hierarchy. We achieve this through recursive modularity optimization. While the Louvain algorithm is a common standard, Graph RAG implementations prefer the **Leiden Algorithm** due to its guarantee of generating connected communities and its superior convergence speed.

4.1 Mathematical Formulation of Modularity

The objective function for community detection is the maximization of **Modularity** (Q). Modularity measures the density of links inside communities compared to links between communities.
For a weighted graph, Modularity Q is defined as:

Modularity

$$Q = \frac{1}{2m} \sum_{i,j} \left(A_{ij} - \frac{k_i k_j}{2m} \right) \delta(\sigma_i, \sigma_j) \tag{6}$$

Where:

- A_{ij} represents the edge weight between nodes i and j.
- $k_i = \sum_j A_{ij}$ is the weighted degree of node i.
- $m = \frac{1}{2} \sum_{i,j} A_{ij}$ is the total edge weight in the graph.
- σ_i denotes the community to which node i belongs.
- $\delta(\sigma_i, \sigma_j)$ is the Kronecker delta function, equal to 1 if $\sigma_i = \sigma_j$, and 0 otherwise.

The term $\frac{k_i k_j}{2m}$ represents the expected edge weight between nodes i and j in a random graph with the same degree distribution. Thus, we maximize the deviation of the actual graph structure from the null model.

4.2 The Leiden Algorithm Mechanics

The Leiden algorithm improves upon the Louvain method by addressing a critical flaw: Louvain can produce arbitrarily badly connected (or even disconnected) communities. Leiden introduces a **Refinement Phase** to guarantee connectivity. The algorithm iterates through three distinct phases:

4.2.1 Phase 1: Fast Local Moving

Similar to Louvain, this phase iterates through all nodes. For each node i, we calculate the gain in modularity ΔQ if i were moved from its current community to the community of one of its neighbors.

$$\Delta Q(i \to C) = \left[\frac{\Sigma_{in} + 2k_{i,in}}{2m} - \left(\frac{\Sigma_{tot} + k_i}{2m} \right)^2 \right] - \left[\frac{\Sigma_{in}}{2m} - \left(\frac{\Sigma_{tot}}{2m} \right)^2 - \left(\frac{k_i}{2m} \right)^2 \right] \quad (7)$$

Node i is moved to the community that maximizes ΔQ, provided $\Delta Q > 0$.

4.2.2 Phase 2: Refinement

This is the distinguishing feature of Leiden. Instead of aggregating the communities directly from Phase 1, we refine them.

- Let $\mathcal{P}_{local}$ be the partition found in Phase 1.
- The algorithm initializes a refined partition $\mathcal{P}_{refined}$ where every node is a singleton.
- It attempts to merge nodes in $\mathcal{P}_{refined}$ only if they belong to the same community in $\mathcal{P}_{local}$ **and** are well-connected.
- This effectively splits the communities from Phase 1 into well-connected sub-communities.

4.2.3 Phase 3: Aggregation

The graph is coarsened based on $\mathcal{P}_{refined}$. Each community in the refined partition becomes a single super-node in the new aggregate graph. Edges between super-nodes are weighted by the sum of weights of the edges between the constituent nodes.

4.3 Algorithmic Representation

The process is recursive. The output of the aggregation phase serves as the input for the next iteration, building the hierarchical levels required for Graph RAG (Level 0 $\rightarrow$ Level 1 $\rightarrow$ ... $\rightarrow$ Root).

Algorithm 1 Leiden Algorithm for Hierarchical Graph Partitioning

Require: Graph $G = (V, E)$

Ensure: Hierarchy of partitions $\mathcal{H}$

$G_{current} \leftarrow G$

$\mathcal{H} \leftarrow \emptyset$

while $G_{current}$ is not sufficiently reduced **do**

    $\mathcal{P}_{local} \leftarrow \text{SingletonPartition}(G_{current})$

    ▷ **Phase 1: Local Move**

    repeat

        for all $v \in V(G_{current})$ **do**

            $C_{best} \leftarrow \underset{C \in \text{Neighbors}(v)}{\text{argmax}} \Delta Q(v \to C)$

            if $\Delta Q > 0$ **then**

                Move v to C_{best} in $\mathcal{P}_{local}$

            end if

        end for

    until Convergence

    ▷ **Phase 2: Refinement**

    $\mathcal{P}_{refined} \leftarrow \text{SingletonPartition}(G_{current})$

    for all $v \in V(G_{current})$ **do**

        if v is well-connected within $\mathcal{P}_{local}(v)$ **then**

            Merge v into a cluster in $\mathcal{P}_{refined}$ within $\mathcal{P}_{local}(v)$

            (Selection based on randomized greedy optimization)

        end if

    end for

    ▷ **Phase 3: Aggregation**

    $G_{next} \leftarrow \text{Aggregate}(G_{current}, \mathcal{P}_{refined})$

    Add $\mathcal{P}_{refined}$ to $\mathcal{H}$

    $G_{current} \leftarrow G_{next}$

end while

return $\mathcal{H}$

4.4 Relevance to Graph RAG Summarization

In the context of Graph RAG, the resulting hierarchy $\mathcal{H}$ allows for Map-Reduce summarization:

(a) **Leaf Communities (Level N):** Contain granular details (e.g., specific interactions between two entities).

(b) **Intermediate Communities (Level 1):** Summarize groups of leaf communities (e.g., Conflict between Department A and B).

(c) **Root Community (Level 0):** Provides the global summary of the entire dataset.

5 Code Example

To demonstrate the *hierarchical* nature of Graph RAG (Level 0, Level 1, etc.) using the Leiden Algorithm, this code uses the `resolution_parameter`.

- **High Resolution**: Detects small, dense "Leaf" communities (local context).
- **Low Resolution**: Detects large, broad "Summary" communities (global context).
- Graph Generation (Stochastic Block Model): We generate a graph that mimics a real Knowledge Graph with clusters. In a real scenario, these clusters might represent topics like "Politics," "Technology," or "Healthcare." The nodes within them are densely connected (related concepts), while connections between clusters are sparse.
- `leidenalg.RBConfigurationVertexPartition`: This is the key line. We use the *Reichardt-Bornholdt* method, which is a variation of Modularity that accepts a `resolution_parameter` (γ).

```
import networkx as nx
import igraph as ig
import leidenalg
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np

```

```
def nx_to_igraph(nx_graph):
  """
  Helper function to convert NetworkX graph to iGraph
  (required because leidenalg is optimized for iGraph).
  """
  # Relabel nodes to integers 0..N-1 for igraph consistency
  mapping = {node: i for i, node in enumerate(nx_graph.nodes())}
  H = nx.relabel_nodes(nx_graph, mapping)

  g = ig.Graph(len(H), list(H.edges()))
  return g, mapping

def plot_graph_partition(ax, G, partition_dict, title, pos):
  """
  Visualizes the graph with nodes colored by their community.
  """
  # Get unique communities and generate a color map
  communities = set(partition_dict.values())
  num_communities = len(communities)
  cmap = cm.get_cmap('viridis', num_communities)

  # Draw nodes
  node_colors = [cmap(partition_dict[n]) for n in G.nodes()]

  nx.draw_networkx_nodes(G, pos, node_size=100, node_color=node_colors, alpha
    =0.9, ax=ax)
  nx.draw_networkx_edges(G, pos, alpha=0.2, ax=ax)

  # Add title and stats
  ax.set_title(f"{title}\n({num_communities} Communities)", fontsize=12,
    fontweight='bold')
  ax.axis('off')

def run_leiden_demo():
  # ------------------------------------------------------------
  # 1. Setup: Generate a Synthetic "Knowledge Graph"
  # ------------------------------------------------------------
  print("Generating Stochastic Block Model Graph...")
  # We create a graph with inherent structure:
  # 5 clusters of 20 nodes, connected with probability 0.5 within, 0.02
    between.
  sizes = [20, 20, 20, 20, 20]
  probs = [[0.5, 0.02, 0.02, 0.02, 0.02],
  [0.02, 0.5, 0.02, 0.02, 0.02],
  [0.02, 0.02, 0.5, 0.02, 0.02],
```

```
  [0.02, 0.02, 0.02, 0.5, 0.02],
  [0.02, 0.02, 0.02, 0.02, 0.5]]

  G = nx.stochastic_block_model(sizes, probs, seed=42)

  # Calculate layout once so nodes don't move between plots
  pos = nx.spring_layout(G, seed=42, k=0.2)

  # Convert to iGraph for the Leiden algorithm
  G_ig, mapping = nx_to_igraph(G)
  reverse_mapping = {v: k for k, v in mapping.items()}

  # ------------------------------------------------------------
  # 2. The Leiden Algorithm (Simulating Graph RAG Hierarchy)
  # ------------------------------------------------------------

  # SCENARIO A: High Resolution (Leaf Communities)
  # This represents specific, granular topics (e.g., "Specific Causal Chain")
  # RBConfigurationVertexPartition implements Modularity with resolution
  part_leaf = leidenalg.find_partition(
  G_ig,
  leidenalg.RBConfigurationVertexPartition,
  resolution_parameter=4.1
  )

  # SCENARIO B: Low Resolution (Root/Summary Communities)
  # This represents the "Map-Reduce" global summary level
  part_root = leidenalg.find_partition(
  G_ig,
  leidenalg.RBConfigurationVertexPartition,
  resolution_parameter=0.3
  )

  # Convert igraph membership back to networkx dict format
  # Format: {node_id: community_id}
  dict_leaf = {reverse_mapping[i]: part_leaf.membership[i] for i in range(len
    (part_leaf.membership))}
  dict_root = {reverse_mapping[i]: part_root.membership[i] for i in range(len
    (part_root.membership))}

  # ------------------------------------------------------------
  # 3. Visualization
  # ------------------------------------------------------------
  fig, axes = plt.subplots(1, 3, figsize=(18, 6))

```

```
  # Plot 1: Raw Graph
  nx.draw_networkx_nodes(G, pos, node_size=50, node_color='gray', alpha=0.5,
    ax=axes[0])
  nx.draw_networkx_edges(G, pos, alpha=0.1, ax=axes[0])
  axes[0].set_title("Raw Knowledge Graph\n(Unstructured)", fontsize=12)
  axes[0].axis('off')

  # Plot 2: Leaf Communities (High Resolution)
  plot_graph_partition(axes[1], G, dict_leaf, "Level 1: Leaf Communities\n(
    Granular / Local Search)", pos)

  # Plot 3: Root Communities (Low Resolution)
  plot_graph_partition(axes[2], G, dict_root, "Level 0: Root Communities\n(
    Global / Map-Reduce)", pos)

  plt.tight_layout()
  plt.savefig("leiden_knowledge_graph_demo.png", dpi=600)
  plt.savefig("leiden_knowledge_graph_demo.pdf")
  plt.show()

if __name__ == "__main__":
run_leiden_demo()
```

Lösungen