4. HubLink: A new KGQA Retrieval Approach

4.1. Concept

4.1.1. Definition of a Hub

To properly use the HubLink Retriever, it is necessary to define beforehand what constitutes a hub. This is important, because during retrieval, the information from each hub are consolidated and used to generate an answer. Therefore, if a question asks for information that is not inside of a hub, the HubLink Retriever will not be able to generate an answer. There are multiple ways to define a hub. In the following, we will introduce some possibilities:

Entity Type If possible, the hubs should be defined by their type defined in the graph. When doing this, the types of entities that are relevant for the QA domain are specified and then used for the classification of hubs. For example, if the QA setting is in the literature research, the objects of interest are publications. In this case, if the KG provides it, it would be straightforward to define publications as a hub type.

Edge Count The simplest way to determine if an entity is a hub is to count the number of edges that are connected to the entity. If the number is above a certain threshold, the entity is classified as a hub. This can be useful, for example, if the information in a graph is highly diverse making it difficult to define all possible types of hubs.

Length of the HubPaths It is also important to consider the length of the paths a hub contains. If the paths are too long, the information from a hub might be diverse and not semantically connected. In this case, it might be useful to add another definition of the hub to split the information into smaller parts. However, if the paths are too short, the information might not be detailed enough to answer a question. Therefore, it is important to monitor the length of the paths when defining a hub.

Defining hubs in a graph where the HubLink retriever is to be applied is an ongoing process. We recommend choosing a hybrid approach from the methods suggested above and continually checking the coverage of the graph in the indexing. It should be ensured that all entities relevant to the given QA setting are captured within a hub.

4.2. HubLink Algorithm

In the following sections, an explanation of the HubLink algorithm is provided. Initially, we introduce the data models employed in the algorithm in subsection 4.2.1. Subsequently, subsection 4.2.2 outlines the indexing procedure executed before retrieval. Following this, subsection 4.2.3 describes the process of identifying hubs within the graph, and subsection 4.2.4 details how these hubs are constructed and stored in the vector store. Then, in subsection 4.2.5, we describe the retrieval process based on a given natural language query using two different strategies. Finally, subsection 4.2.6 presents the process of answer generation.

4.2.1. Data Models

Data Models 1 Data Models Overview

Class: `Entity`	▹ Represents a node in the graph
Class: `Triple`	▹ Represents a whole triple in the graph
Class: `EntityWithDirection`	
Entity: `Entity`	▹ Represents a node in the graph
IsLeft: `Boolean`	▹ Indicates the current direction
PathToTopic: `List` **of** `Node`	▹ The path from the topic entity to this entity
Class: `Hub`	
Root: `EntityWithDirection`	▹ Root node of the Hub
Paths: `List` **of** `HubPath`	▹ Represents a list of the paths in the Hub
Class: `HubPath`	
PathText: `String`	▹ A textual description of the path
Embedding: `Array`	▹ The embedding of the PathText
PathHash: `String`	▹ The unique hash value of the path
Path: `List` **of** `Triple`	▹ The path consisting of triples
Class: `HubPathWithScore`	
Path: `HubPath`	▹ The HubPath to score
Score: `Float`	▹ The relevance score towards the question
Class: `HubScoringSummary`	
HubRoot: `EntityWithDirection`	▹ Root node of the Hub
Paths: `List` **of** `HubPathWithScore`	▹ The HubPaths with score
Score: `Float`	▹ The relevance score towards the question

The HubLink algorithm operates with a set of data models. These models are used for exchanging data between the different steps of the algorithm. In the following, we describe each model in detail.

Entity The `Entity` model represents a node in the graph. In our implementation, an entity can be either a literal or an object with a unique ID. Objects also have a class and a label that can be accessed via edge traversal.

Triple The `Triple` model represents a triple in the graph. We follow the RDF schema which is also used in the ORKG [18]. According to this schema, a triple consists of a subject, a predicate, and an object. Both the subject and object are models of type `Entity`, while the predicate is a string.

EntityWithDirection The graph on which the HubLink Retriever is applied is a directed graph. Therefore, when traversing the graph, a direction can always be specified. This direction is stored in the `EntityWithDirection` model together with the entity itself. This allows for a direction-aware traversal. Additionally, this data model records the complete path through the graph from the starting entity of the search to the current entity.

Hub A hub is a special concept used in the retrieval approach that is of particular significance for a specific domain or question. Hubs consolidate information and serve as pivotal points for searching relevant data. The `Hub` model stores the root entity of the hub and the associated HubPaths. The root entity is an `EntityWithDirection` object, which allows tracing the origin of the search within the graph. The HubPaths are a list of `HubPath` objects, which are described below.

HubPath A HubPath is a sub-path within the graph composed of a list of triples. We define a HubPath as a path that starts from the root entity of a hub and either leads to an end node in the graph or to another hub. HubPaths are stored in the corresponding `HubPath` model. In addition to the path itself, it also stores a textual description of the path, an embedding of this text, and a hash value of the path. The textual description of the path is used to generate an embedding which is then utilized to calculate the relevance of the path to the question during retrieval. The hash value of the path is used to retrieve already indexed paths from the cache to avoid recalculation.

HubPathWithScore The `HubPathWithScore` model stores a `HubPath` along with a relevance score. This score is used to assess the relevance of the path related to the research question during retrieval. The score ranges from 0 to 1, where 0 signifies that the path is not relevant, and 1 indicates high relevance.

HubScoringSummary The `HubScoringSummary` model stores a summary of the scores for all HubPaths of a hub. It includes the root entity of the hub, the list of HubPaths with their respective scores, and the overall score of the hub. The overall score is calculated as the average of all HubPath scores.

4.2.2. Indexing

Algorithm 1 Indexing Hubs

Input:

G: `KnowledgeGraph` ▷ The graph to search in
$\vec{R}$: `List` **of** `Entity` ▷ Entities from which the indexing is started
α: `Integer` ▷ The max depth to index

```
1:  procedure HUBINDEXING
2:      roots ← {EntityWithDirection(T, True, []) for T in R⃗}
3:      roots.insert({EntityWithDirection(T, False, []) for T in R⃗})
4:      depth ← 0                                   ▷ The current level of the search
5:      while depth ≤ α do
6:          hub_entities, next_entities ← FINDHUBROOTS(G, roots)
7:          roots ← next_entities                   ▷ The next level of entities
8:          next_entities ← BUILDHUBS(hub_entities)
9:          roots.insert(next_entities)
10:         if roots = ∅ then Break                 ▷ Stop if no more entities to process
11:         depth ← depth + 1
12:     end while
13: end procedure
```

For indexing, the HubLink Retriever expects a list of starting nodes $\vec{R}$. This design choice comes from the fact that the graph G can be very large with potentially millions of nodes. This indexing approach allows for defining a subgraph $\hat{G}$ on which the retrieval is performed, thus avoiding the need to index the entire graph. In addition, a maximum depth α is defined, which specifies how deep the index should go into G, determining the size of $\hat{G}$.

The indexing function is shown in Algorithm 1. Starting from $\vec{R}$, the graph G is scanned. The process iterates through G until either the maximum depth α is reached or no further nodes are found to traverse. In each iteration, the algorithm checks if there are additional nodes in $\vec{R}$ to continue the indexing process. If not, the indexing process terminates. If $\vec{R}$ contains nodes, `findHubRoots` is called with $\vec{R}$ to find the hub entities *hub_entities* and the next set of entities *next_entities*. The hub entities discovered by this function are then passed to the `buildHubs` function to finally construct the hubs. For subsequent traversal, the `findHubRoots` function also returns a list of entities that are appended to $\vec{R}$.

Parameter α: Maximum Depth

The maximum depth is defined by η. This parameter specifies the maximum depth at which a hub can be located to be included in the indexing process. However, since processing a hub requires indexing all its paths, the actual total indexing depth can exceed η, depending on the internal depth of the hub structure itself.

4.2.3. Finding Hub Root Entities

Algorithm 2 Finding the Root Entities of a Hub

Input:

G: `KnowledgeGraph` ▷ The graph to search in
$\vec{V}$: `List` **of** `EntityWithDirection` ▷ Entities that have already been visited
$\vec{E}$: `List` **of** `EntityWithDirection` ▷ The entities to start the search from
η: `Boolean` ▷ Indicates whether the hubs need to have the same amount of hops

Output:

A tuple (`hub_entities`, `next_entities`) containing the entities that are the root of a Hub and the entities that are used to initialize the search on the next level.

```
1:  function findHubRoots
2:      Initialize queue ← Queue(Ē)                          ▷ Entities to traverse
3:      hub_entities ← ∅                                     ▷ Collection of Hub root entities found
4:      next_entities ← ∅                                    ▷ Candidates for the next level
5:      while queue ≠ ∅ do
6:          e ← queue.pop()                                  ▷ Current entity to process
7:          if e ∈ V̄ then Continue                           ▷ If entity has been visited, skip it
8:          V̄.insert(e)                                      ▷ Add entity to visited
9:          e_isHub ← isHub(e)                               ▷ Check if entity is a hub
10:         if e_isHub and e ∉ hub_entities then
11:             hub_entities.insert(e.Entity)                ▷ Add entity to the list
12:         end if
13:         if e_isHub and ¬e.Left then
14:             Continue                                     ▷ Stop traversing the current path
15:         end if
16:         all_relations ← {All relations of e considering its direction}
17:         if η then queue.insert(all_relations)
18:         else next_entities.insert(all_relations)
19:     end while
20:     return (hub_entities, next_entities)
21: end function
```

The function `findHubRoots`, depicted in **??** is responsible for finding the root entities of the hubs, starting from a list of starting entities $\vec{E}$. These entities define the points on the graph from which the search is started. The function is implemented in a way similar to breadth-first search. This means that initially, all neighbors of an entity are visited before descending deeper into the graph. To start this search, the queue is initialized with $\vec{E}$. In each iteration, an entity is taken from the queue and processed. First, a check $e \in \vec{V}$ is performed to verify whether this entity has already been visited. Subsequently, it is checked whether the entity is a hub by calling the function `isHub`.

If the entity is a hub, it is added to *hub_entities*, provided that it is not already included. Then, a check is performed to determine if the direction of the entity is left. If so, the

incoming edges of the entity are added to the queue or to the list of *next_entities* depending on η. The reason we only continue traversing in the left direction is that hubs are always processed in the direction of the graph, which is right. This means, that the entties to the right, are part of the Hub and will be processed by the `buildHubs` function.

On the other hand, if the entity is not a hub, we collect all relations of e considering the current direction of the traversal. This means, that if we are traversing against the direction of the graph, we only add relations that are inbound of e. The algorithm then proceeds to the next iteration and finishes once the queue is empty. The hub entities that have been found are stored in *hub_entities* which is returned in addition to the *next_entities* which are used for the traversal of the next level.

Parameter η: Continue Traversal Direction

The decision to add edges to the queue or to the list of next entities depends on the parameter η. This distinction affects, whether the function should continue traversing in the current direction or only consider the relations of the current entity that are one hop away. Therefore, if η is set to `False`, only Hubs are returned, that are directly connected to an entity in $\vec{E}$. If η is set to `True`, the function will continue traversing each path of the entities in $\vec{E}$ until a Hub is found. This is useful in a scenario, where Hubs that should be compared to each other, do not have the same amount of hops. Therefore, the choice of η depends on the structure of the graph.

Parameter $\vec{V}$: Visited Entities considering Direction

The parameter $\vec{V}$ is used to prevent hubs from being processed multiple times and to avoid cycles in the graph. The list $\vec{V}$ accounts for the direction of traversal in the graph, tracking the directions in which entities have already been explored. This ensures that the retrieval process can also traverse in the opposite direction of the graph.

4.2.4. Building Hubs

Building the hubs is the most time-consuming process in the HubLink Retriever. An overview of the processes that are applied during the building of the hubs is illustrated in Figure 4.1. The graphic shows four boxes which will be explained in the following:

1. In the first box, a snippet of a graph is shown. It shows a paper entity which we define as a hub type in our following example. The paper entity is associated with multiple edges that supply further information about it, such as the title or the authors of the paper. The first step of building a hub is to identify the root entity of the hub which is done in the `findHubRoots` function shown in ??.

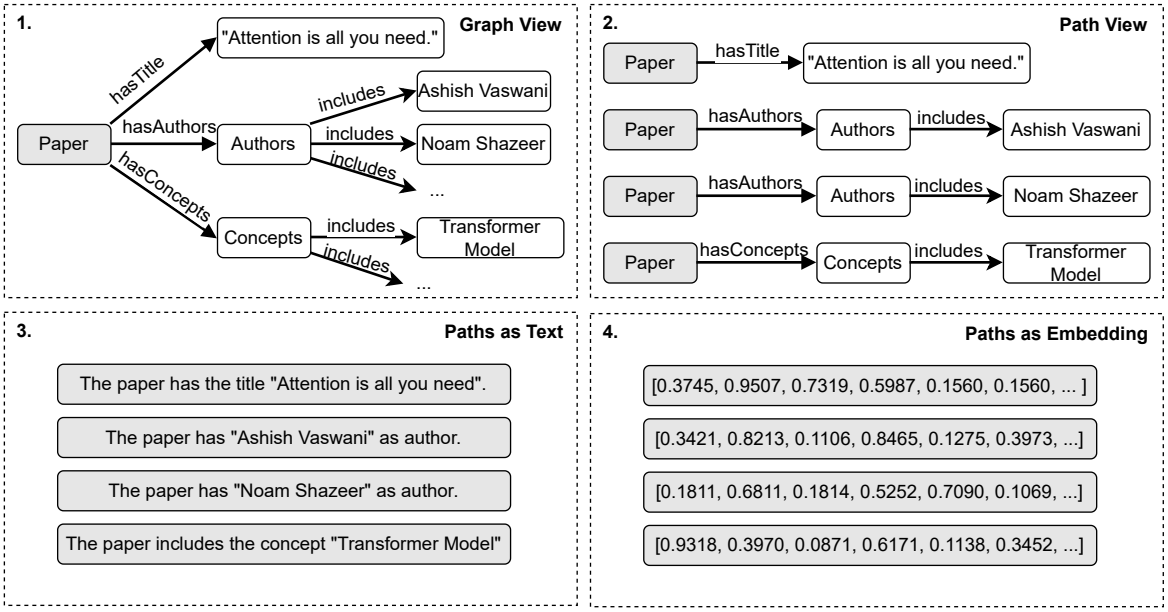

Figure 4.1.: The process of building Hubs in the HubLink Retriever.

2. The next step is to find the individual paths of the hub. These paths are called *HubPaths* and start from the root entity of the hub until either the end of the graph or another root entity of the hub.
3. After each individual path is identified, each HubPath is converted to a textual representation using a LLM. This representation is a short description including each triple of the path. It is important that each information piece from the path is included in the description to avoid losing information.
4. The final step is the conversion of the textual description of each HubPath to a vector representation using a pre-training embedding model. This representation is then later used to quickly find related paths by calculating a distance metric towards the question being asked.

4.2.4.1. Building Hubs

The detailed process of building the Hubs is described in the Algorithm 3. It shows the function `buildHubs` which is called with a list of root entities E_h. The function assumes that each entity in E_h is already confirmed to be a root entity of a Hub. The function then iterates over each entity $e \leftarrow E_h$ to subsequently build the Hub for each entity e.

For this building process, two function are called. First, the function `findPathsInHub` is called on e which finds the individual paths as shown in the ① box of Figure 4.1. In addition, this function also returns a list of entities that are used for deeper indexing. These entities are either hub entities for hubs in a deeper level or the entities at the end of a path when the maximum path length has been reached. The subsequent phase in the construction

of the hub involves transforming the paths into text and then converting this text into embeddings, as shown in the boxes ② and ③ of Figure 4.1, respectively. This is done in the function `buildHubPaths` which is invoked with the paths found in the previous call of `findPathsInHub`. In addition, `buildHubPaths` also stores these embeddings and all the HubPath information in the vector store.

Algorithm 3 Building Hubs

Input:
E_h: `List` **of** `Entity` ▷ The root entities of the Hubs

Output:
A list of `Hub` objects that have been built

```
1:  function buildHubs
2:      Initialize hubs ← ∅                          ▷ The Hubs that have been built
3:      Initialize next_entities ← ∅                 ▷ Entities for deeper indexing
4:      for all e ∈ E_h do
5:          (p⃗, e⃗) ← findPathsInHub(e)
6:          next_entities.insert(e⃗)
7:          hub_paths ← buildHubPaths(p⃗, e)
8:          current_hub ← Hub(e, hub_paths)
9:          hubs.insert(current_hub)
10:     end for
11:     return hubs, next_entities
12: end function
```

4.2.4.2. Finding the Paths of a Hub

To find the paths within a hub, the function `findPathsInHub` is used, which is described in Algorithm 4. This function is called with the root node R of the hub. Initially, three lists are initialized: `visited`, `candidates`, and `paths`. The `visited` list keeps track of nodes that have already been processed to avoid cycles. The `candidates` list stores the root entities of other hubs that can be processed on subsequent function calls. The `paths` list stores all discovered paths within the hub.

The main loop of the function is implemented using a queue that is initially populated with the root node R and an empty list. In each loop iteration, a node e and its corresponding path $\vec{p}$ are extracted from the queue. First, the algorithm checks if e has already been visited and whether the entity is not the root node. If both conditions are true, it proceeds to the next iteration. Next, it verifies whether the current path $\vec{p}$ has reached the maximum length allowed α. If so, the path is added to the `paths` list and the algorithm continues with the next iteration.

The next step of the algorithm is to examine all outgoing edges of node e. If there are no outgoing edges, it is considered an end node, and the current path is added to the `paths`

Algorithm 4 Finding the Paths of a Hub

Input:

G: `KnowledgeGraph` ▷ The graph to search in
R: `Entity` ▷ The root entity of the hub
α: `Integer` ▷ The maximum length of a HubPath

Output:

A list of paths for the hub rooted at R

```
1:  function findPathsInHub
2:      Initialize visited ← ∅                          ▷ Set of entities visited
3:      Initialize candidates ← ∅                       ▷ Candidates for next level
4:      Initialize paths ← ∅                            ▷ Paths of triples from the graph
5:      Initialize queue ← Queue((R, ∅))                ▷ Queue with root entity and empty path
6:      while queue is not empty do
7:          e, p⃗ ← queue.pop()                          ▷ Current entity and path
8:          if e ∈ visited and e ≠ R then continue      ▷ Avoid circles
9:          visited.insert(e)
10:         if |p⃗| ≥ α then                             ▷ Path is longer than the max length
11:             paths.insert(p⃗)
12:             continue
13:         end if
14:         r⃗ ← getOutboundTriples(e, G)                ▷ Retrieve neighbors of e
15:         if r⃗ = ∅ then
16:             paths.insert(p⃗)                         ▷ End node reached
17:             continue
18:         end if
19:         for all r ∈ r⃗ do
20:             e′ ← getObject(r)   ▷ Returns the object in (subject, predicate, object)
21:             if e′ = R then continue                 ▷ Skip if going back to root
22:             if isHub(e′) then
23:                 candidates.insert(e′)
24:                 paths.insert(p⃗)
25:             else
26:                 p⃗′.insert(r)
27:                 queue.insert((e′, p⃗′))              ▷ Add neighbor to queue
28:             end if
29:         end for
30:     end while
31:     return paths
32: end function
```

list. Otherwise, each edge is evaluated to determine whether its corresponding neighboring node is a hub. If this is the case, the path is not followed further and the neighboring node is added to the `candidates` list. If none of these conditions are met, the neighboring node is appended to the path and then added to the queue for further traversal. This process continues until the queue is empty. In that case, all discovered paths within the Hub are returned.

Parameter α: Maximum Length of a HubPath

The parameter α specifies the maximum length a `HubPath` can have. It is used to limit the length of the paths to prevent the algorithm from traversing deeper into the graph than necessary. It is important for an `HubPath` to be relevant to the information in the hub. Therefore, the parameter α can help avoid paths becoming too long and losing relevance. Therefore, the choice of α depends on the structure of the graph.

4.2.4.3. Building the Paths of a Hub

The function `buildHubPaths`, described in Algorithm 5, is responsible for processing the paths of a hub as defined in ② and ③ of Figure 4.1. The function receives a list of paths $\vec{P}$ as input and iterates over it. For each path, a hash value δ is generated to uniquely identify the path. This hash value serves as a key in the Vector Store $\hat{V}$ for storing the path, the textual description of the path, and the path itself. Using δ, the algorithm performs a check on $\hat{V}$ to determine if the path is already stored. If the path is already stored, the corresponding `HubPath` object is retrieved from $\hat{V}$ and added to the list of processed HubPaths.

If the path is not yet in $\hat{V}$ or needs updating, it is converted into a textual representation using an LLM. This textual representation is then transformed into a vector using an embedding model. With this information, a new `HubPath` object is created which is then stored in $\hat{V}$ and added to the list of processed HubPaths. After all paths have been processed, the list of processed HubPaths is returned.

Path Hash δ

Hashing the path $\vec{p}$ to a unique hash value δ is used to identify the path in the Vector Store. Using a hash value as a key parameter has the advantage that if the path is changed, the hash value will also change. This allows to update the paths of the Hub during indexing.

4.2.5. Retrieval Strategies

In the following section we will describe the retrieval process of the HubLink Retriever. The retrieval process is responsible for finding relevant hubs and paths within the graph that can be used to generate an answer to a given question. The retrieval process is divided into two different strategies depending on whether or not a Topic Entity is provided.

Algorithm 5 Building Hub Paths

Input:
$\hat{V}$: `VectorStore` ▷ A vector store to store HubPaths
$\vec{P}$: `List` **of** `List` **of** `Triple` ▷ The paths to process

Output:
A list of `HubPath` objects representing processed paths

```
1:  function buildHubPaths
2:      Initialize hub_paths ← ∅                          ▷ Stores the processed HubPaths
3:      for all p⃗ ∈ P⃗ do                                   ▷ Iterate over all paths
4:          δ ← pathToHash(p⃗)                             ▷ Convert path to a hash value
5:          if δ ∈ V̂ then
6:              ĥ ← V̂.getHubPath(δ)                       ▷ Retrieve HubPath from vector store
7:              hub_paths.insert(ĥ)                       ▷ Append to processed list
8:          else
9:              t ← pathToText(p⃗)                         ▷ Convert path to text representation using LLM
10:             μ⃗ ← getEmbedding(t)                       ▷ Get embedding for text representation
11:             ĥ ← HubPath(t, μ⃗, δ, p⃗)                   ▷ Initialize the HubPath
12:             V̂.storeHubPath(ĥ)                         ▷ Store HubPath in vector store
13:             hub_paths.insert(ĥ)                       ▷ Append to processed list
14:         end if
15:     end for
16:     return hub_paths                                  ▷ Returns the processed HubPaths
17: end function

18: function pathToHash(p⃗)
19:     δ ← apply hash function on p⃗
20:     return δ
21: end function

22: function pathToText(p⃗)
23:     return Convert p⃗ to text using an LLM
24: end function

25: function getEmbedding(t)
26:     μ⃗ ← apply embedding model on t
27:     return μ⃗
28: end function
```

4.2.5.1. Direct Retrieval Strategy

Algorithm 6 Retrieval Strategy without Topic Entity

Input:
G: `KnowledgeGraph` ▷ The graph to search in
Q: `String` ▷ The question being asked

Output:
The final answer derived from the candidate hubs

```
1:  function RETRIEVE
2:      candidate_scorings ← FINDCANDIDATEHUBS(Q)        ▷ Find candidate hubs for Q
3:      Initialize partial_answers ← ∅                   ▷ The partial answers for each hub
4:      for all scoring ∈ candidate_scorings do
5:          partial_answer ← GETPARTIALANSWERFORHUB(scoring, Q)
6:          partial_answers.insert(partial_answer)
7:      end for
8:      if partial_answers ≠ ∅ then
9:          return GETFINALANSWER(partial_answers, Q)    ▷ Compute the final answer
10:     end if
11: end function
```

If the retriever only receives a question without a topic entity, the algorithm depicted in Algorithm 6 is used. The algorithm starts by calling the function `findCandidateHubs` with the question Q which returns a list of `HubScoringSummary` objects containing the hubs that are most likely relevant to the question. For each of this hub candidates the function `getPartialAnswerForHub` is called with the summary. This function asserts whether enough information is available in the hub to generate a partial answer that is relevant to the question. If at least one partial answer is generated, the function `getFinalAnswer` is called to combine these partial answers into a final answer, which is then returned.

The main logic of the retrieval strategy is implemented in the `findCandidateHubs` function shown in Algorithm 7. The function is called with the question Q and returns a list of `HubScoringSummary` objects that are relevant to the question. The function starts by initializing an empty dictionary *candidates* to store the candidate hubs and then enters a loop that continues until the desired number of candidate hubs γ has been found or there are no more results to process.

Each iteration starts by retrieving a number of paths δ from the vector store $\hat{V}$. δ is calculated by multiplying the number of paths that are still needed to reach γ with the number of paths α that are considered for each hub. Using this calculation ensures that enough paths are retrieved with a high probability of seeing the desired number of candidate hubs while ensuring that not too many paths need to be retrieved. The retrieval from $\hat{V}$ is done using the function `similaritySearchExcludingHubs`. This function queries $\hat{V}$ using the question Q, a list of entities $\vec{e}$ that are excluded from the search, and the number of paths to retrieve δ. It returns a list of `HubPath` objects that have the highest similarity score towards Q clustered

Algorithm 7 Find Candidate Hubs

Input:

G: `KnowledgeGraph` ▷ The graph to search in
$\hat{V}$: `VectorStore` ▷ A vector store to store HubPaths
α: `Integer` ▷ Number of paths considered for each Hub
γ: `Integer` ▷ Number of Hubs to compare
Q: `String` ▷ The question

Output:

A list of `HubScoringSummary` objects that are relevant to Q

```
1:  function findCandidateHubs
2:      Initialize candidates ← ∅                          ▷ List of candidate hubs
3:      while |candidates| < γ do
4:          δ ← (γ − |candidates|) · α                     ▷ Number of paths to search for
5:          e⃗ ← {candidate.HubRoot.Entity for candidate in candidates}
6:          r⃗ ← V̂.similaritySearchExcludingHubs(Q, e⃗, δ)
7:          if r⃗ = ∅ then Break                            ▷ No more results to process
8:          for all (hub_entity, hub_paths) ∈ r⃗ do
9:              p⃗ ← hub_paths[0 : min(α, |hub_paths|)]
10:             candidate ← HubScoringSummary(hub_entity, p⃗, 0)
11:             candidates.insert(candidate)
12:             if |candidates| ≥ γ then Break             ▷ Reached desired number of hubs
13:         end for
14:     end while
15:     for all candidate ∈ candidates do
16:         if |candidate.Paths| ≥ α then Continue
17:         p⃗ ← V̂.similaritySearchByHub(candidate.HubRoot.Entity, Q, α)
18:         candidate.Paths ← paths ∪ {p ∈ p⃗ | p ∉ candidate.Paths}
19:     end for
20:     return candidates
21: end function
```

by their hub entity. The similarity score is calculated using a distance metric between the question embedding and the path embedding using cosine similarity. Excluded from the search are the hubs that have already been found. Each hub is then processed by selecting the first α paths from the list of paths. A `HubScoringSummary` object is created for each hub containing the hub entity, the paths, and the score. For this strategy, the hub score is set to 0 because no further pruning is applied which makes the calculation of this score unnecessary. The created summary is then added the *candidates* list.

After the desired number of candidate hubs has been found or there are no more results to process, we need to ensure that each candidate hub has at least α paths. Therefore, we iterate over each candidate hub and check if the number of paths is less than α. If this is the case, we retrieve additional paths from the vector store using the function `similaritySearchByHub`.

This function queries $\hat{V}$ using the hub entity and the question Q to retrieve paths only for this hub. The paths are then added to the list of paths of the candidate hub to fill it up to α. Finally, the list of candidate hubs is returned.

Parameter α: Number of Paths for a Hub

The parameter α specifies the number of paths that are retrieved for each hub. It controls the amount of information about a hub that is used to generate a partial answer. The choice of α has a direct influence on the costs and quality of the retrieval process. A higher value of α increases the amount of information that is considered for each hub, which leads to a higher probability of finding information about a hub that is relevant to the question. However, this also increases the context window that is queried to the LLM for the generation of the partial answer, which can lead to higher costs.

Parameter γ: Number of Hubs to Compare

The parameter γ specifies the number of hubs that are compared to each other to find the most relevant hubs for the question. The choice of γ has a direct influence on the quality of the retrieval process. A higher value of γ increases the number of hubs that are considered for the generation of the final answer. This can lead to a higher quality of the final answer, but also increases the costs of the retrieval process. Another consideration are the types of questions that are asked. If the questions require information that is stored across multiple hubs, a higher value of γ might be needed to be able to fully answer the question.

4.2.5.2. Graph Traversal Retrieval Strategy

The retrieval strategy with a topic entity is described in Algorithm 8. The strategy is used when a topic entity T is provided, which represents an entity in the graph and serves as the starting point for the retrieval process. The retrieval algorithm starts by initializing a list *roots* containing the topic entity T with both search directions for the graph. After this initialization, a loop runs until the maximum level β is reached or an answer has been found. Additionally, in each iteration, the algorithm also checks if there are still entities remaining in the *roots* list. If not, the algorithm ends.

In each loop iteration, the function `getHubPathsForEntities` is called, which returns a list *hub_scorings*. This list contains a summary of the HubPaths and scores for each hub, calculated through similarity computations between the path embeddings and the question embedding. `getHubPathsForEntities` also returns a list *next_entities* that contains candidates for traversal in the next iteration. After the summaries have been collected, the function `pruneHubs` is called, which selects the top γ most relevant hubs from the *hub_scorings* list. This is done by first calculating the average score for each hub based on the scores of its HubPaths, then sorting the hubs by this score, and selecting the top γ hubs. After the relevant hubs have been selected, the function `getPartialAnswerForHub` is called

Algorithm 8 HubLink Retriever with Topic Entity

Input: G : KnowledgeGraph ▷ The Graph to search in
Input: Q : String ▷ The question that is asked
Input: T : Entity ▷ The entry node to start the retrieval from
Input: β : Integer ▷ The maximum level to traverse
Input: γ : Integer ▷ The number of Hubs to compare

function RETRIEVEWITHTOPICENTITY
  Initialize $roots \leftarrow \emptyset$ ▷ The entities to search from
  $roots$.INSERT($EntityWithDirection(T, True, [])$) ▷ To traverse to the left
  $roots$.INSERT($EntityWithDirection(T, False, [])$) ▷ To traverse to the right
  Initialize $level \leftarrow 1$ ▷ The current processing level
  $\vec{\varepsilon} \leftarrow$ GETEMBEDDING(Q) ▷ The embedding of the question
  while $level \leq \beta$ **do**
    if $roots = \emptyset$ **then return** ▷ No more entities to traverse
    $hub\_scorings, next\_entities \leftarrow$ GETHUBPATHSFORENTITIES($\vec{\varepsilon}, roots$)
    $roots \leftarrow next\_entities$ ▷ The next entities to traverse
    if $hub\_scorings = \emptyset$ **then Continue** ▷ No HubPaths at the level
    $relevant\_scorings \leftarrow$ PRUNEHUBS($hub\_scorings, \gamma$)
    $partial\_answers \leftarrow []$ ▷ The partial answers for each Hub
    for $scoring \leftarrow relevant\_scorings$ **do**
      $p_a \leftarrow$ GETPARTIALANSWERFORHUB($scoring, Q$)
      $partial\_answers.insert(p_a)$
    end for
    if $partial\_answers \neq \emptyset$ **then**
      return GETFINALANSWER($partial\_answers, Q$)
    end if
    $l++$
  end while
  return ▷ No answer has been found
end function

function PRUNEHUBS($hub\_scorings, \gamma$)
  for $hub\_scoring \leftarrow hub\_scorings$ **do**
    $hub\_scoring.score \leftarrow$ AVERAGEPATHSCORE($hub\_scoring.Paths$)
  end for
  $hub\_scoring \leftarrow$ `Sort` $hub\_scorings$ `by score`
  return `Top` *gamma* `entries of` $hub\_scorings$
end function

for each hub to generate a partial answer. If at least one partial answer is generated, the function `getFinalAnswer` is called to combine the partial answers into a final answer, which is then returned.

Parameter β: Maximum Level to Traverse

The parameter β specifies the maximum level that the retrieval process should traverse in the graph and as such limits the number of hops that are considered for the retrieval process. The choice of β can influence whether the retrieval process can find the relevant information to answer the question. A higher value of β increases the number of hops that are considered, which can lead to a higher probability of finding relevant information. However, this also increases the costs of the retrieval process.

Algorithm 9 Get HubPaths for Entities

Input: $\hat{V}$: VectorStore ▷ A vector store to store HubPaths
Input: $\vec{\varepsilon}$: List **of** Float ▷ The embedding of the question
Input: *entities* : List **of** Entity ▷ The entities to search for
Input: α : Integer ▷ Number of paths to consider for each Hub

```
1:  function getHubPathsForEntities
2:      (hub_entities, e⃗) ← findHubRoots(G, entities)
3:      next_entities ← e⃗                          ▷ The root nodes for the next level
4:      if hub_entities = ∅ then return ([], next_entities)
5:      hub_scorings ← []                          ▷ List of HubScoreSummary
6:      for e ← hub_entities do
7:          paths_r ← V̂.searchByHubId(ε⃗, e, α)
8:          hub_scorings.insert(HubScoreSummary(e, paths_r))
9:      end for
10:     return (hub_scorings, next_entities)
11: end function
```

The function `getHubPathsForEntities` is described in Algorithm 9. It returns a list of `HubScoringSummary` objects containing the relevant hubs and their paths with associated scores for the given entities. Additionally, the function returns a list of entities that will be used in the next iteration of the retrieval process for deeper graph traversal.

`getHubPathsForEntities` begins by calling `findHubRoots`, which identifies the next hub root entities starting from the topic entity while considering the direction of the graph. The root entities of the discovered hub are stored in the list *hub_entities*, and the corresponding entities for the next iteration are stored in the list *next_entities*. If no root entities are found for the hub, the function terminates. Otherwise, the root entities of the hub are processed in a subsequent loop.

For each hub root entity, a search is performed in the Vector Store $\hat{V}$ to find the top α paths that are most relevant to the question. This search only considers paths that belong

to the given hub identified by the root entity. The discovered paths are then stored in a `HubScoringSummary` object and added to the *hub_scorings* list. After processing all root entities of the hub, both *hub_scorings* and *next_entities* are returned.

4.2.6. Answer Generation

Algorithm 10 Get Partial Answer

Input: R_h : EntityWithDirection ▷ The root entity of the Hub
Input: $\vec{H}$: List **of** HubPathWithScore ▷ The HubPaths with scoring of the Hub
Input: Q : String ▷ The question that was asked
Input: $\hat{D}$: Database ▷ A database that stores additional knowledge

1: **function** GETPARTIALANSWERFORHUB
2: Initialize $\hat{k} \leftarrow \emptyset$ ▷ Additional Knowledge about the Hub
3: **if** $R_h.PathToTopic \neq \emptyset$ **then** ▷ Check if we have a path to a Topic entity
4: $\vec{p} \leftarrow R_h.PathToTopic$ ▷ The path from the topic entity
5: $t* \leftarrow$ PATHTOTEXT($\vec{p}$) ▷ Convert the path to text representation
6: $\hat{k}$.INSERT($t*$) ▷ Add the path to the additional knowledge
7: **end if**
8: $link\_knowledge \leftarrow \hat{D}$.GETLINKKNOWLEDGE($R_h.Entity$)
9: $\hat{k}$.INSERT($link\_knowledge$) ▷ Add the knowledge to the additional knowledge
10: **return** Query LLM for a partial answer for q with the paths $\vec{p}$ and knowledge $\hat{k}$
11: **end function**

The algorithm `getPartialAnswerForHub` shown in Algorithm 10 shows the generation of a partial answer from the information of a hub. To create this answer, the text representations of the HubPaths are used and enriched with additional information. The first additional piece of information is a textual description of the complete search path. This path is created by the `findHubRoots` function and stored in the `EntityWithDirection` object. It contains the path starting from the topic entity to the root entity of the hub. This information is intended to inform the LLM about how the information was found in the graph. Since this information requires traversal through the graph, it is only used in the retrieval strategy with a topic entity. The second additional information used to generate the partial answer is provided through the linking concept. This involves additional information associated with a hub that can be retrieved via the root entity of a hub. This information could include metadata or original texts related to the hub, which further enrich the structural information of the hub from the KG. Once all information about the hub has been collected, the LLM is queried with this information to generate a partial answer.

The function `getFinalAnswer` shown in Algorithm 11 is responsible for combining the partial answers generated from the hubs into a final answer. The partial answers are passed to the LLM along with the original question. The LLM then generates a final answer based on the information provided in the partial answers and the question. The final answer is then returned to the user.

Algorithm 11 Get Final Answer

Input: Q : String ▷ The question that was asked
Input: $\vec{A_p}$: List **of** String ▷ The partial answers from the Hubs

1: **function** GETFINALANSWER($Q, \vec{A_p}$)
2: **return** ask the LLM to give an answer based on the partial answers $\vec{a}$ and the question Q
3: **end function**

4.3. Discussion about Retrieval Strategies

In this section, we discuss the advantages and disadvantages of the strategies introduced. The advantage of using a topic entity is that it restricts the search space within the graph. Without such an entity, the entire graph would need to be searched, as the location of relevant information is unclear. This could lead to inaccuracies since information from irrelevant parts of the graph might be included. Therefore, the topic entity acts as a filter for the retrieval process, helping to focus the search on relevant portions of the graph.

Another advantage of using a topic entity is that it allows tracking the path from the topic entity to each hub in the graph. This enables the retriever to gather additional information about the entire path to the hub and incorporate it into the answer generation process. This makes it possible to include graph information that exists outside of the hub in the answer. For example, when asking about papers from a specific publisher, where the publisher is not part of the hub, the retriever can determine through the path to the hub that a paper comes from a particular publisher. However, this requires that the publisher is included in the path to the hub.

Durch das Traversieren der Pfade des Graphen, ist es außerdem möglich das Retrieval in mehreren sogenannten Leveln durchzuführen. Das bedeutet, dass statt das Retrieval ohne Ergebnis abzubrechen, wie bei der direkten Strategie falls nicht direkt die relevanten Hubs gefunden werde, die Suche nach weiteren relevanten Hubs fortgesetzt werden kann. Dies bedeutet, dass wenn die initialen Hubs keine Ergebnisse liefern, durch weiteres Suchen entlang der Pfade des Graphen, weitere Hubs gefunden werden können, die möglicherweise Antworten auf die Frage liefern können. Wenn also Beispielhaft Publicationen und Authoren getrennte Hubs sind, und die Frage nach einem Paper eines bestimmten Autors gestellt wird, kann der Retriever über den Paper Hub traversieren und anschließend über den Author Hub die Antwort zusammenführen. In diesem Fall beinhalten die Hubs alleine noch nicht die Antwort, da allerdings der Pfad der Traversierung berücksichtigt wird, kann der Retriever die Antwort generieren.

Das traversieren der Pfade während des Retrievals bringt allerdingsa auch Nachteile mit sich. Zunächst ist es notwendig eine geeignete Topic Entität zu finden. Sie kann direkt vom Fragesteller angegeben werden, beispielsweise wenn bereits bekannt ist das sich die Frage nur auf Publicationen eines bestimmten Research Fields bezieht. Es gibt auch die Möglichkeit die Topic Entität automatisch aus der Frage zu bestimmen, beispielsweise

durch Named Entity Recognition. Dafür ist es notwendig das die Frage eine entsprechende Information enthält, die als Topic Entität verwendet werden kann. Beispielsweise die Frage "Which papers from the software architecture field, talk about the use of machine learning?" enthält die Information das es sich um das Research Field Software Architecture handelt. Dies kann als Topic Entität verwendet werden kann, wenn der Graph Research Fields als Entitäten speichert. Wenn allerdings die Frage "Which papers talk about the use of machine learning?" gestellt wird, fehlt die Information über das Research Field. In diesem Fall ist es in der Regel nicht möglich eine Topic Entität zu bestimmen und der gesamte Graph muss durchsucht werden. Wie man an diesen Beispielfragen sehen kann, ist die genaue Formulierung der Frage entscheidend für die Qualität des Retrievals.

Desweiteren ist die Traversierung des Graphen Zeit und kosten aufwendig. Dies liegt daran, dass zunächst die Hubs im Graphen ausgehend von der Topic Entität durch traversierung der Pfade gefunden werden müssen. Anschließend muss für jede gefundene Root Entität die entsprechenden HubPaths geladen werden um anschließend auszuwerten ob eine partielle Antwort generiert werden kann. Dieser Prozess ist aufwendig und kann je nach Größe des Graphen und Anzahl der Hubs einige Zeit in Anspruch nehmen. Der Zeitaufwand steigt weiter, wenn die initialen Hubs keine Antwort liefern und weitere Level traversiert werden müssen. Dadurch steigen auch die Kosten des Retrievals, da für jede Hub Auswertung LLM aufrufe getätigt werden müssen.

4.4. Limitations

1. If the information that is askes for is not indexed as part of a hub, it can't be found by the retriever. This is a limitation of the retrieval process, as the retriever can only find information that is part of a hub.

4.5. Previous concepts

4.6. Challenges

4.6.1. Prompt Engineering

4.6.2. Cost vs. Performance

4.6.3. Hub Definition

4.7. Summary