Corpus Annotation Graph Builder (CAG): An Architectural Framework to Create and Annotate a Multi-source Graph

Roxanne El Baff Tobias Hecking Andreas Hamm Jasper W. Korte Sabine Bartsch*
German Aerospace Center (DLR), Germany, <first>.<last>@dlr.de
* Technical University of Darmstadt, sabine.bartsch@tu-darmstadt.de

Abstract

Graphs are a natural representation of complex data as their structure allows users to discover (often implicit) relations among the nodes intuitively. Applications build graphs in an ad-hoc fashion, usually tailored to specific use cases, limiting their reusability. To account for this, we present the Corpus Annotation Graph (CAG) architectural framework based on a *create-and-annotate* pattern that enables users to build uniformly structured graphs from diverse data sources and extend them with automatically extracted annotations (e.g., named entities, topics). The resulting graphs can be used for further analyses across multiple downstream tasks (e.g., node classification). Code and resources are publicly available on GitHub[1, 2] and downloadable via PyPi[3] with the command `pip install cag`.

1 Introduction

In many areas of knowledge, facts are spread across a multitude of documents in various forms and modalities (e.g., texts, images, sound recordings, videos, and program code). It is of great interest to exploit explicit links between these documents and - even more so - to detect hidden (or implicit) connections — for knowledge extraction through searching, classifying, comparing, and analyzing.

The most explicit and efficient data structure for working with such interlinked document collections (corpora) is that of a graph whose nodes represent documents, entities, and annotations with edges representing relations between them. Graph databases can naturally manage these for efficient knowledge querying and storage. In particular, property graph databases like Neo4j[4] and ArangoDB[5] consider nodes as objects with attributes and are, therefore, appropriate for holding documents and metadata or more general objects of interest (OOI). OOIs can vary in scale and can be linked via containment relationships: e.g., a book series contains books, a book contains chapters, a chapter contains text, pictures, etc.

One central idea behind our framework is that graphs are not only the object of analysis but also containers for analysis results. More precisely, to unveil hidden connections, we suggest deriving a second type of nodes, which we call *annotation nodes* associated with the OOI nodes. They represent features extracted from the OOIs by methods like named entity recognition, topic discovery, sentiment analysis, image captioning, etc. Implicit links of OOIs can then be established via shared annotation nodes (e.g., common topics) that can be used for further analysis. In this way, with every analysis passed, the graph can get richer and possibly more connected. We call the resulting graph a *Corpus Annotation Graph*.

In this demo paper, we present an architectural framework that facilitates the application of the *create-and-annotate* pattern for creating such a graph: the **Corpus Annotation Graph Builder** (CAG). CAG is built on top of ArangoDB and its Python drivers. The *create-and-annotate* pattern consists of two phases (see Figure 1) that can be repeated multiple times: (1) OOI data can be collected from different sources (e.g., publication databases, online encyclopedias, news feeds, web portals, electronic libraries, repositories, media platforms) and pre-processed to build the core nodes. The component responsible for this phase is the **Graph-Creator**. (2) Annotations are extracted from the nodes, and corresponding annotation nodes are created and attached to the graph. The component dealing with this phase is the **Graph-Annotator**. At any time, new OOI nodes or new annotations can be added, introducing new information to enrich the graph with subsequent

[1] https://github.com/DLR-SC/corpus-annotation-graph-builder
[2] The README includes a link for the documentation.
[3] https://pypi.org/project/cag/
[4] https://neo4j.com
[5] https://www.arangodb.com

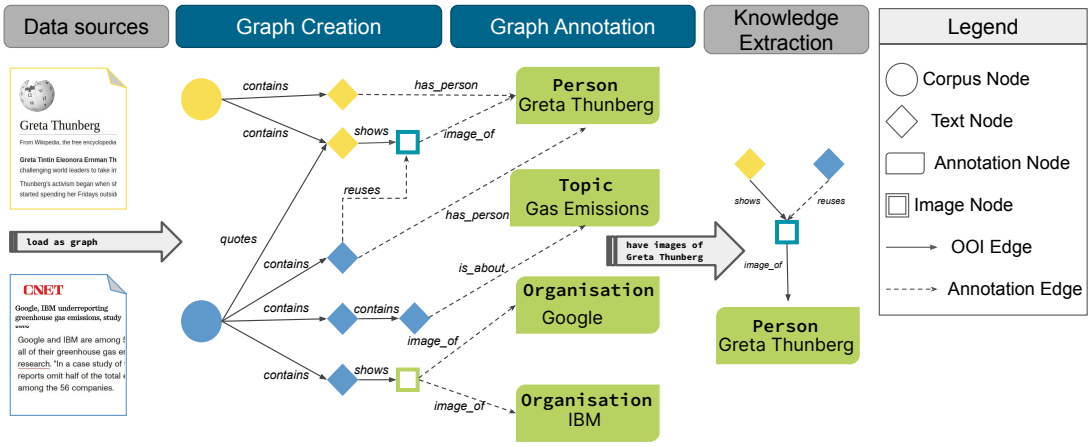

Figure 1: A simplified illustration of the framework: (1a) The data is collected from different data sources (e.g., scientific articles, Wikipedia. (1b) Then, the data is pre-processed and loaded into one common graph with core nodes and edges referred to as objects of interest (OOI). And (2) Annotations are added to the core nodes, which creates additional levels of abstraction and allows core components to be linked across different corpora. An example of a query is *Get all documents that contain images with the Person Of Interest; Greta Thunberg*. This results in getting *text a* from a datasource and *text b* from another datasource.

runs further.

The CAG architectural framework aims to offer researchers a flexible but unified way of organizing and maintaining their interlinked document collections in a reproducible way.

In the following sections, we go over related work (Section 2), then we present the architecture of CAG by describing the building blocks of our framework (Section 3). Last but not least, Section 4 exemplifies two applications, one as the backbone technology for searching scientific work and another one describing a knowledge graph construction using CAG.

2 Related Work

Existing research and tools related to graph data structures cover various topics, from database management systems for creating and maintaining graphs, over a formal framework for graph annotations, to task-oriented (e.g., information retrieval, node classification) approaches.

As we mentioned in the Introduction (Section 1), as the foundation of the architectural framework (CAG), we use a pattern of creating and annotating graphs. Currently, there exist powerful graph databases, such as Neo4j and ArangoDB.They offer a mature engine to build and persist property graphs. We base our architectural framework on ArangoDB because it allows saving attributes not only in the nodes but also in the edges of a graph. It is a valuable option for incorporating more information if needed.

Throughout the years, graphs are highly used to solve several downstream tasks, such as developing extended similarity metrics for documents embedded in graphs (Minkov et al., 2006), Named Entity Recognition (Yu et al., 2008), measuring the semantic distance between texts (Tsang and Stevenson, 2010), using a graph-based ranking algorithm (Demir et al., 2010), frame semantic parsing (Zheng et al., 2022) among many others ((Chen et al., 2022), (Colas et al., 2022), etc.). All these works demonstrate the importance of graphs and their multipurpose usage, making it essential to develop an architectural framework to create and annotate graphs so they can be (re)used later on for similar or different downstream tasks.

Regarding graph annotations, lately, Bikaun et al. (2022) introduced QuickGraph, a rapid annotation tool that allows users, via its web interface, to upload corpora and add annotations to it by selecting the annotation type from a predefined list (e.g., named entities). As mentioned above, CAG is not a tool but rather a more general programming framework that employs an architectural pattern for the goal of code centralization and reproducibility, giving users the flexibility to select their datasources and annotation types. CAG has predefined nodes (e.g., TextNode) and edges to encourage the unification of information from different sources, and it allows the addition of any annotation type to its predefined annotations.

Formerly, Bird and Liberman (1999, 2001) defined 'Linguistic annotation' as a descriptive or

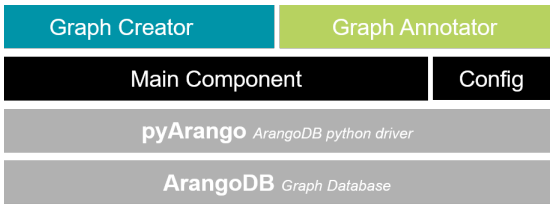

Figure 2: The building blocks of the CAG framework. The framework uses ArangoDB as a graph database with the Python driver *PyArango*. On top of that, CAG has a *Config* block responsible for establishing the connection to the database. Moreover, the *Main component* is the base component responsible for creating/updating the graph ontology, nodes, and edges. The top building blocks are *Graph-Creator* and *Graph-Annotator*.

analytical notation applied to raw language data of any form (e.g., textual, voice). The added notations may include annotations of all sorts (from phonetic features to discourse structures) (Bird and Liberman, 2001), which provides a formal framework for constructing, maintaining, and searching linguistic annotations while remaining consistent with many alternative data structures and formats. They focus on linguistic annotations regardless of the raw data from which they exploit the commonalities between them. Our work here adapts the same concept for *annotations*. Our annotation framework extends an existing graph with new node types holding a notation derived from any raw data. Maeda et al. (2002, 2006) present the Annotation Graph Toolkit, a formal framework to support the development of annotation tools of time series data based on Bird and Liberman (2001) framework; these tools allow users to annotate a data point manually. The main difference is that CAG is an architectural framework, not a toolkit. It allows the incorporation not only of time series data but any data. Most importantly, our framework supports automated bulk annotations instead of manual ones.

3 The Framework

In this section, we describe the technical framework of the Corpus Annotation Graph builder (CAG). As mentioned in Section 1, we create an architectural framework that employs the *create-and-annotate* pattern, which requires two main building blocks in CAG: Graph-Creator and Graph-Annotator. As shown in Figure 2, these two blocks are built on top of several other blocks. From the bottom up, CAG uses ArangoDB as a graph database to store graphs, which is accessed via the python library, PyArango[6]. After that comes the **Main Component** block, which creates and updates the graph ontology by manipulating the graph elements. It uses the **Config** block to establish the connection to the database. Table 1 summarizes both components' methods, each a Python class.

Function Name	Description
Main Component `cag.framework.component.py`	
Constructor	Establishes the connection to the database using the *Config* class, creates the graph ontology if it does not exist, or updates it.
get_document	Gets a node or an edge based on a key or set of attributes.
upsert_node	Updates or creates a new node instance. It uses the get_document function to fetch existing nodes.
get_edge_attributes	Gets the edge attributes based on Edge type and from and to nodes.
upsert_edge	Update or create a new edge instance.
Config `cag.utils.config.py`	
Constructor	Establishes the connection to the database using the database attributes: database URL, username, password, and port. It retries in case of connection failure.
configuration	Re-establishes a new connection and rewrites a previous one.

Table 1: Summary of the functions in the `Main Component` and the *Config*. These functions are used by CAG's Graph-Creator and -Annotator.

The sections below delve deeper into the two main building blocks on top, Graph-Creator, and Graph-Annotator, where we explain the relation between them and between the lower blocks.

3.1 Graph-Creator

As mentioned in Section 1, a Graph-Creator (GC) creates objects of interest (OOI) from a datasource as nodes and edges where these OOIs are directly extracted without further analysis (e.g., text content, image). GC offers a unified layout, structuring the creation process of a graph from one or many datasources. **`GraphCreatorBase`**, CAG's primary GC class, allows the management of a datasource

[6] https://pypi.org/project/pyArango/

Function Name	Description
Constructor	Sets the path to the source data, sets the data configuration from *Config*, and calls the *initialization* method that should be implemented.
init_graph	An `abstract` method to initialize the nodes and edges by using pre-existing node/edge specific methods (e.g., `create_corpus_node`, `create_text_node`, `create_image_node`, `create_author_node`) or by using the *upsert_node* or `upsert_edge` from the **Component** class.
update_graph	An `abstract` method to update the nodes and edges by using pre-existing methods from the **Component** methods.

Table 2: Summary of the main functions in the `Graph Creator` abstract class.

Name	Attributes and Description
Nodes	
GenericOOSNode	timestamp. The generic node class from which all python class nodes inherit.
CorpusNode	name, type, description, created_on and timestamp.
TextNode	text, timestamp
AuthorNode	name and timestamp
ImageNode	URL and timestamp
WebResource	URL and timestamp
Edges	
BelongsTo	timestamp. For example a *TextNode* BelongsTo a *CorpusNode*
HasAuthor	timestamp. For example, *TextNode* HasAuthor an *AuthorNode*
RefersTo	timestamp. For example, *TextNode* RefersTo a *WebResource* node

Table 3: A sample of the predefined nodes and edges in CAG (`cag.graph_elements`).

within a graph. It is an abstract class that inherits all the functionalities of the "Main Component" (Table 1). As shown in Table 2, `GraphCreatorBase` enforces the implementation of two functions: one for *initializing* the graph from a datasource and one for *updating* it. Additionally, GC offers a predefined set of edges and nodes with their corresponding maintenance (inserting/updating) that can be optionally used (see Table 3). A project can have as many graph creators as is needed, usually one per datasource. A sample code is available on GitHub[7].

Another trait of GC is that it allows time propagation through linked nodes since each OOI node carries a timestamp of its creation. For example, if a set of content nodes (N) linked to a parent node (P) is updated/created, the latest timestamp of these nodes can be recursively propagated to P.

The CG creates a graph by loading raw data to it. This graph is enriched by using the Graph-Annotator.

3.2 Graph-Annotator

The Graph-Annotator (GA) enriches the graph by analyzing object of interest (OOI) nodes and linking them to newly created annotation nodes. For example, as shown in Figure 1, a textual node can be linked with a *has_person* edge to a named entity node of type PERSON and attribute "Greta Thunberg". Annotations can be applied on different levels: (1) a collection of nodes such as corpus level to extract corpus statistics, topics, etc., (2) a single node such as text nodes (e.g., keyphrases), image nodes (e.g., generated captions), etc.

Figure 3 shows the workflow of GA. A set of nodes, predefined by the user, is fed to a customizable **pipeline** where each **pipe** has three responsibilities:(1) accessing the node(s) and extracting a corresponding feature (e.g., named entities), (2) processing the features (e.g., count the number of times a named entity, e.g., ORGANIZATION: "Google", occurred), (3) updating the graph by saving the annotations corresponding to this pipe. For example, Figure 3 shows that all the text nodes were selected to be annotated. They are, then, fed to a pipeline that has three pipes: a *sentence divider*, a *named entity pipe*, and an *emotion* pipe. After running the pipeline, the graph is updated with the corresponding annotation nodes.

Pipe We define a CAG pipe as a set of objects that encapsulates several functionalities to deal with the technical part of annotation, the post-processing, and the persistency of the annotation nodes in the graph. A pipe is mainly defined in two steps: (1) defining its attributes in CAG's `registered pipes dictionary` and (2) and defining a `PipeOrchestrator`.

`registered pipes` is a Python dictionary that has one entry for each pipe, holding all at-

[7] https://github.com/DLR-SC/corpus-annotation-graph-builder/blob/main/examples/1_create_graph.ipynb

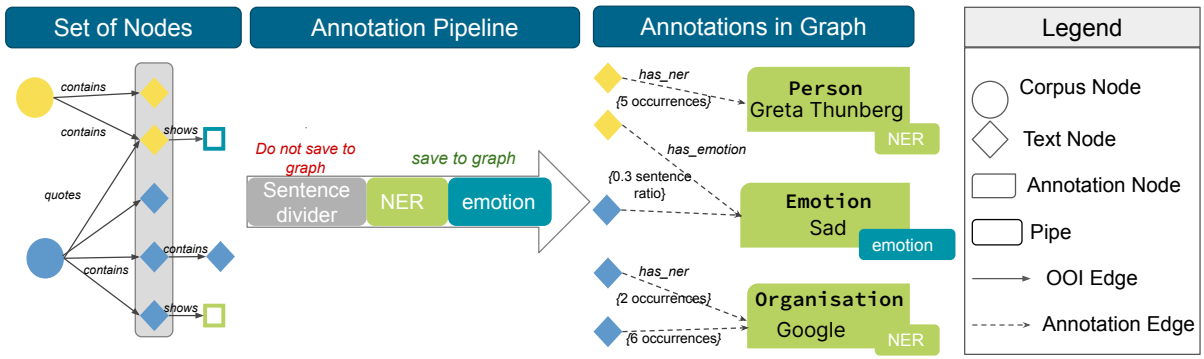

Figure 3: A simplified illustration of the Graph-Annotator. (1) The text node set is selected and fed to a pipeline. (2) The pipeline consists of three pipes: sentence divider (preprocessing), NER, and Emotion pipes. After annotating, (3) the annotation nodes are saved in the graph.

tributes (Table 4) needed for a pipe to tackle its responsibilities and distinguished by a unique key.
.

Attribute	Description
orchestrator_class	The path to the pipe's orchestrator class.
pipe_id_or_func	A *Spacy* predefined id (e.g., ner) or a customized function id that is implemented in the *pipe_path*.
pipe_path	The path to the customized function or empty in case of Spacy.
level	Annotation level whether it is on the level of a single node or a set of node.
data_type	Whether the annotated OOI is a text, image or URL.
annotated_node_name	the name of the OOI node being annotated.
node_class	The Path to the class of the annotation node.
edge_class	The Path to the class of the edge node.

Table 4: The attributes of a registered pipes. CAG's predefined pipes are under `cag.framework.annotator.registered_pipes.pipes`

The `PipeOrchestrator` is an abstract python class that loads the pipe components based on the attributes provided in the `registered pipes`. The orchestrator validates the attributes in Table 4 (e.g., ensures correct paths) and loads the required python modules, making them accessible later on for the **pipeline**. It also creates new annotation node/edge types by updating the graph ontology. Additionally, the PipeOrchestrator is an abstract class that enforces the implementation of three methods: `create_node`, `create_edge` and `save_annotations`. The latter method should loop over the annotations and use the other two methods to save the annotation nodes and edges.

After defining all the pipes, the **pipeline** manages them.

Pipeline GA offers a feature to unify the pipeline's definition. The **`pipeline`** manages pipes by using the pipe's information accessed from the `PipeOrchestrator`. It deals with enforcing the flow of the pipes (e.g., extract sentences before classifying them), executing the pipeline (using *pipe_id_or_func* in Table 4), which outputs the annotations, and saving to the graph (by calling the pipe's corresponding `save_annotations` mentioned previously). The Pipeline class supports the execution of customized pipes (a function call), Spacy (Honnibal et al., 2020) customized components, or Spacy native components (e.g., SentenceRecognizer for sentence segmentation). The pipeline only executes these functions based on information provided by the pipe, and it is not responsible for the logic within these pipes. A pipe is added as follows to a pipeline (sample on GitHub[8]):

```
add_annotation_pipe(
    name="MyPipeOrchestrator",
    save_output=True,
    is_spacy=True
    )
```

It is important to note that for provenance track-

[8] https://github.com/DLR-SC/corpus-annotation-graph-builder/blob/main/examples/2_annotate_graph.ipynb

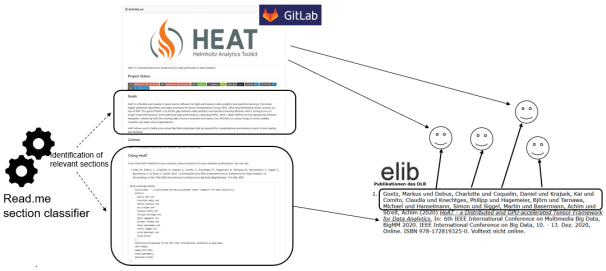

Figure 4: Linking Git repositories and publication entries through readme files and metadata.

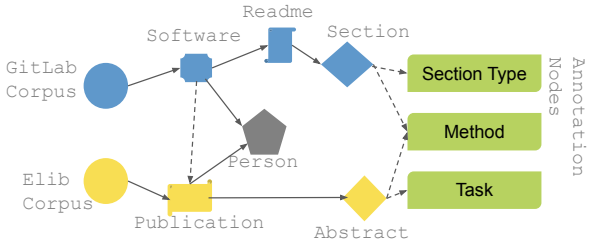

Figure 5: CAG structure for linking GitLab (blue) projects and publications (yellow). Graph creators create blue, yellow, and grey (common node) nodes and solid edges, and graph annotators add green nodes and dashed edges by analyzing the OOI nodes.

ing of annotations, an annotation node can be equipped with additional metadata about its creation timestamp, the component that created the annotation, and additional parameters needed for the analysis reproduction. An example is listed below:

```
{
  analysis_component: 'keyphrases,'
  parameters:
            {
              algorithm: text_rank,
              relevance_threshold: 0.75
            }
}
```

4 Use Cases

4.1 Linking Publications and Software

In many fields of science, it is common to use Git repositories hosting platforms such as GitHub and GitLab. Those repositories often have a "readme" file introducing the purpose of tools, dataset descriptions, or usage instructions. Thus, these files agglomerate valuable science and software knowledge (El Baff et al., 2021) that is worth exploring. Often one can also find references to associated traditional publications. We use the CAG to integrate search over fragmented information in publication outlets and Git repositories. More precisely, we build a graph that indirectly links repositories and papers via intermediate keywords and persons and directly by parsing paper references in repositories' "readme" files (Figure 4).

The German Aerospace Center (DLR), with a constantly growing corpus of publications and software tools, is interested in efficiently retrieving digital scientific information. Thus, as a backbone of an internal search application, the CAG framework is used to build an organizational knowledge graph linking publications in the DLR publication database elib[9] and repositories in its self-hosted GitLab instance[10].

Graph-Creator. The graph creator processes a dump of GitLab "readme" files and the associated project metadata (e.g., contributors, title, URL) retrieved through the GitLab API[11], as well as dumps of publication abstracts from the elib publication database along with titles, authors, institutes into the hierarchical structure of OOI nodes depicted in Figure 5 with colors blue for projects, yellow for publications and grey for common nodes between the datasources, linked via solid edges.

Graph Annotators. We implement a pipeline containing several pipes to annotate the graph by extracting information from the text nodes and saving the annotation nodes, as shown in Figure 5.

Readme section classifier: We use the Prana et al. (2019) readme classifier to extract sections labeled as a description (What/Why sections), reference, and acknowledgment. These labels are linked to the readme text nodes.

Reference parser: We developed a tool for parsing paper references in "readme" from raw form (e.g., APA, BibTeX) to items such as author, title, and venue. This annotator uses the extracted information to establish links to publication nodes where possible.

Concept extraction: For linking software repositories and publications, even if there is no direct reference or overlap in persons, we use the SPERT (Eberts and Ulges, 2020) information extraction model trained on scientific corpora SciERC (Luan et al., 2018) to extract concepts where we focus on "Method" and "Task" since these constitute the most relevant links between papers and software projects. Such concept annotations are attached

[9] https://elib.dlr.de/

[10] https://gitlab.dlr.de/

[11] https://docs.gitlab.com/ee/api/

to the text nodes of publication abstracts and the "readme" sections tagged as "description".

This use case is similar to the portal https://paperswithcode.com/, but our solution is automatized. The CAG framework eases processing large volumes of publications and repository corpora into a well-defined graph format.

4.2 Wikipedia Page Revisions

We present here a simple use case depicting the usage of the Graph-Creator and -Annotator from a widely known source, Wikipedia.

As a community-driven encyclopedia, Wikipedia has a lens on societal discourse. Since the entire revision history of each article is publicly available, Wikipedia constitutes a research dataset that allows tracing the evolution of themes over time. Using the CAG framework, we demonstrate how to create a graph from Wikipedia revisions for the two categories, *climate change* and *artificial intelligence*. We extract the revisions for two periods, October 2012 and -2022. The data is downloaded using our tool, `wikipedia-periodic-revisions`[12] which downloads Wikipedia revisions for a specific category and period. It, then, saves each Wikipedia Page as a file. The code is available on GitHub[13].

Graph-Creator. Figure 6 shows the nodes and edges predefined by CAG (in blue) and the ones the newly defined ones (in yellow). The Wikipedia graph creator loops over the pages to load the data into the graph. On top, it creates a Wikipedia corpus node as the most general OOI node, referenced by Wikipedia articles. Each article comprises of revision nodes, each linked to a text node. Additionally, the graph creator already establishes cross-article connections by parsing images and external references. Figure 6 shows the general scheme of our use case.

Graph-Annotators. We create an annotation pipeline utilizing the Spacy named entity recognition module[14] to extract named entities from revision texts. Text nodes containing common entities (e.g., referring to the same Organization) will be indirectly linked together through these entities. In this way, article relationships are also established

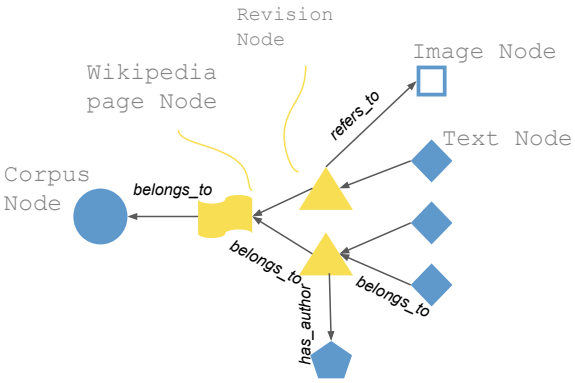

Figure 6: Wikipedia graph structure. CAG's predefined nodes are in blue, and newly defined nodes are in yellow.

based on common referencing (either explicit by containing, for example, common images (an OOI) or implicit by just mentioning common entities).

The resulting CAG structure (see Figure 6) allows interesting graph queries to analyze the co-evolution of Wikipedia articles through time. Examples are: "In which context a certain organization (named entity) is mentioned on Wikipedia?" or "Which articles reciprocally reference each other?".

5 Conclusion and Future Work

We presented CAG, a publicly available architectural framework aiming to employ unified and reproducible patterns in graph creation and extension via annotations. CAG allows users to concentrate on graph ontologies and pipelines while the framework takes the burden of handling repetitive and cumbersome tasks. We further aim to extend our framework to have an analysis component incorporating dynamic data analysis through time and space, exploiting the 'create-and-annotate' graph results.

6 Acknowledgment

This project is funded by the German Federal Ministry of Education (BMBF) under the InsightsNet[15] project. We want to thank Shahbaz Syed[16] for his valuable input, especially regarding the Figures presented in the paper. We also would like to thank the reviewers for their helpful comments.

[12] https://github.com/DLR-SC/wikipedia-periodic-revisions, also downloadable via PyPi https://pypi.org/project/wikipedia_tools/

[13] https://github.com/roxanneelbaff/cag_wikipedia_usecase

[14] https://spacy.io/api/entityrecognizer

[15] http://insightsnet.org/

[16] https://scholar.google.com/citations?hl=en&user=eGe86TEAAAAJ

References

Tyler Bikaun, Michael Stewart, and Wei Liu. 2022. QuickGraph: A rapid annotation tool for knowledge graph extraction from technical text. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 270–278, Dublin, Ireland. Association for Computational Linguistics.

Steven Bird and Mark Liberman. 1999. Annotation graphs as a framework for multidimensional linguistic data analysis. In *Towards Standards and Tools for Discourse Tagging*.

Steven Bird and Mark Liberman. 2001. A formal framework for linguistic annotation. *Speech Communication*, 33(1):23–60. Speech Annotation and Corpus Tools.

Chen Chen, Yufei Wang, Bing Li, and Kwok-Yan Lam. 2022. Knowledge is flat: A Seq2Seq generative framework for various knowledge graph completion. In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 4005–4017, Gyeongju, Republic of Korea. International Committee on Computational Linguistics.

Anthony Colas, Mehrdad Alvandipour, and Daisy Zhe Wang. 2022. GAP: A graph-aware language model framework for knowledge graph-to-text generation. In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 5755–5769, Gyeongju, Republic of Korea. International Committee on Computational Linguistics.

Seniz Demir, Sandra Carberry, and Kathleen F. McCoy. 2010. A discourse-aware graph-based content-selection framework. In *Proceedings of the 6th International Natural Language Generation Conference*. Association for Computational Linguistics.

Markus Eberts and Adrian Ulges. 2020. Span-based joint entity and relation extraction with transformer pre-training. In *Proceedings of the 2020 European Conference on Artificial Intelligence*, pages 2006–2013. IOS Press.

Roxanne El Baff, Sivasurya Santhanam, and Tobias Hecking. 2021. Quantifying synergy between software projects using readme files only. In *Proceedings of the 33rd International Conference on Software Engineering and Knowledge Engineering*, volume 33. KSI Research Inc. and Knowledge Systems Institute Graduate School.

Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. 2020. spaCy: Industrial-strength Natural Language Processing in Python.

Yi Luan, Luheng He, Mari Ostendorf, and Hannaneh Hajishirzi. 2018. Multi-task identification of entities, relations, and coreference for scientific knowledge graph construction. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3219–3232, Brussels, Belgium. Association for Computational Linguistics.

Kazauki Maeda, Steven Bird, Xiaoyi Ma, and Haejoong Lee. 2002. Creating annotation tools with the annotation graph toolkit. In *Proceedings of the Third International Conference on Language Resources and Evaluation (LREC'02)*, Las Palmas, Canary Islands - Spain. European Language Resources Association (ELRA).

Kazuaki Maeda, Haejoong Lee, Julie Medero, and Stephanie Strassel. 2006. A new phase in annotation tool development at the Linguistic Data Consortium: The evolution of the annotation graph toolkit. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC'06)*, Genoa, Italy. European Language Resources Association (ELRA).

Einat Minkov, William Cohen, and Andrew Ng. 2006. A graphical framework for contextual search and name disambiguation in email. In *Proceedings of TextGraphs: the First Workshop on Graph Based Methods for Natural Language Processing*, pages 1–8, New York City. Association for Computational Linguistics.

Gede Artha Azriadi Prana, Christoph Treude, Ferdian Thung, Thushari Atapattu, and David Lo. 2019. Categorizing the content of github readme files. *Empirical Software Engineering*, 24(3):1296–1327.

Vivian Tsang and Suzanne Stevenson. 2010. A graph-theoretic framework for semantic distance. *Computational Linguistics*, 36(1):31–69.

Xiaofeng Yu, Wai Lam, and Shing-Kit Chan. 2008. A framework based on graphical models with logic for Chinese named entity recognition. In *Proceedings of the Third International Joint Conference on Natural Language Processing: Volume-I*.

Ce Zheng, Xudong Chen, Runxin Xu, and Baobao Chang. 2022. A double-graph based framework for frame semantic parsing. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4998–5011, Seattle, United States. Association for Computational Linguistics.