Kernel - Level Semantic Search with Knowledge Graphs

Arjun Deodhar
Dept. of Computer Science
COEP Technological University
Pune, Maharashtra
deodhark22.comp@coeptech.ac.in

Arnav Prasad
Dept. of Computer Science
COEP Technological University
Pune, Maharashtra
arnavp22.comp@coeptech.ac.in

Prajwal Bhosale
Dept. of Computer Science
COEP Technological University
Pune, Maharashtra
bhosalepp22.comp@coeptech.ac.in

***Abstract*—Various types of knowledge exists across individuals, processes and tools. The chief objective of Knowledge Graph (KG) is to aggregate the data into graph format, ensuring that it remains manageable, non-corrupted, scalable and easily discoverable.**

At its core, a KG is a structure were each node represents real-world entities and edges logically depict the relationships between the nodes.The graph can either be directed or undirected, depending on the organization's need. Our approach involves a directed graph that also includes backward edges!

The end objective of KG is to operationalize Knowledge (a piece of information) at the kernel level and make it available to users when they feed specific queries to the graph. The output should be the most relevant and concise response available, neither too lengthy nor too brief.

***Index Terms*—Knowledge Graphs, AVL, heap**

A. Abbreviations and Acronyms

- **KGs**: Knowledge Graphs
- **AVL**: Adelson-Velsky and Landis
- **max_heap**: maximum heap
- **NLP**: Natural Language Processing

I. Introduction

KGs are advanced data structures that represent information in a network of interconnected entities and their relationships. It consists of nodes, representing entities (such as people, places, or concepts), and edges, representing the relationships between these entities. They are designed to enable machines to understand and process complex data, facilitating improved information retrieval, question answering, and semantic search capabilities. The concept of the KG gained significant recognition in 2012 when Google [7] publicly credited their search solution to the use of KG. The relationships in KG are often labeled to provide context and meaning, forming a rich, structured representation of knowledge. One of the key advantages of KGs is their ability to integrate and organize data from various sources, providing a unified view that can be easily queried and analyzed. They support both direct and inferred relationships, enabling more sophisticated reasoning and inference.

KGs have various applications :

- **Commerce**: Widely used by platforms like Amazon [2], [3] and eBay [4] to describe and categorize products for sale.
- **Social Networking**: Utilized by platforms such as LinkedIn [5] to manage and connect users, jobs, skills, and more.
- **Finance**: Bloomberg [6] has developed a knowledge graph that powers financial data analytics, including sentiment analysis for companies based on current news reports and tweets.

This paper is focused on implementation of a KG which takes input from user and further acts a source of knowledge that's significantly needed in wide range of areas. We have stated the core structure and workflow of data that's needed to implement large models like LLM, NLP, etc without using any external library or 3rd party components. The aim of this paper is to provide a road-map for a thorough exploration of the fundamental implementation aspects of KGs.

II. Literature Survey

An extensive body of literature, including surveys, books, and academic publications, has been dedicated to exploring the intricacies of KGs. They provide the aspects and key topics related to KG.

We also came across one of the languages / vocabulary in which a KG may be designed, which is a data-modelling language called RDF (Resource Distribution Framework) [1].

Our goal is to put forward the structural implementation and significance of KG.

III. Methodology

Data includes a wide range of scope and context. It can be of variable length, contexts, grammars, etc. A KG is not just a collection of data, but also contains hierarchical structures, contexts for data, as well as unique identifiers for real word [8]

The constructed Graph should be able to handle and recognize the following characteristics of data:

- *Context* [8] of entities. For example, if I am speaking regarding "bank", the context can be river-side bank or the bank which involves financial transactions or specifically Bank of Maharashtra or may have some other context. KGs should be able to manage all these varying contexts and should have scope for addition of new contexts in it.

- *Weight* [8] of the data(relation). The output should be in accordance to decreasing order of these weights to achieve the most desired response. This includes recognition of ”front-weight” and ”back-weight” of data. For example, consider the sentence : ”Arjun plays piano”. This may be an important attribute for ”Arjun” but not that relevant for node (entity) ”piano”(i.e. piano is being played by ”Arjun” and piano will have more important(weighted) characteristics). The text analyzer should be able to assign front and back weights accordingly.
- *Temporal* [8] constraints. For example, the statement ”XYZ is the Prime Minister of India” will be valid only for a certain period of time(say 5 years) and then it may change. After expiration, the relation should be redundant.
- *Identity* [8], for example, ”Arjun plays piano” and ”Arjun plays guitar” here both ”Arjun” have the same name, but in the real world they may be different entities. The analyzer should assign unique identifiers to different entities.
- *Misspelled Words Queries* : The constructed graph should be able to detect spelling errors in input queries and deliver the correct output by prompting the matching entities.

To accomplish all these features, the input data set that we designed contain the following fields: front-weight, inference, truth-bit, noun1, noun1_id , verb , verb descriptor, noun2, noun2_id, back-weight, definition, end-time.

Once a query is fired, the constructed KG should be able to give concise outputs. For example, if the user requests only 10 lines of output, the response should adhere to this limit and prioritize the main points.

IV. Architecture

The implementation in C programming language [9], employs structures and pointers for AVL trees and arrays for heaps, providing a robust framework for managing noun-centeric knowledge representations and their interconnections via verbs.

```c
typedef struct noun_tree_node
{
        char * noun_name;
        long long int noun_id;
        char * noun_def;
        struct verb_tree_node * next;
        struct verb_tree_node * prev;
        struct search_maxheap * src_heap;
        struct subclass_maxheap * sub_heap;

        struct noun_tree_node * left;
        struct noun_tree_node * right;
        long long int balance_factor;
} noun_tree_node;
```

Fig. 1. Structural implementation in C

AVL trees are used for their self-balancing property which ensures worst-case $O(\lg n)$ time complexity for search and insert operations. Heaps, on the other hand, excel in extracting elements based on their priority with worst-case $O(\lg n)$ time complexity for insertion and retrieval.

A. Structure

1) Central Noun Tree: The core of our knowledge graph is an AVL tree where each node represents a noun. Each noun node comprises the following elements:

- *noun_name*: Pointer to a string representing the noun.
- *noun_id*: A unique identifier for the noun.
- *noun_definition*: Pointer to a string containing the definition of the noun.
- *next_verb_tree*: An AVL tree containing verbs that form front links to other nouns.
- *prev_verb_tree*: Pointer to an AVL tree containing verbs that form back-links to other nouns.
- *search_max-heap*: A max-heap for extracting the highest priority connections.
- *subclass_max-heap*: A max-heap for accessing the highest priority sub-classes of the noun.
- *left, right, balance_factor* : for maintaining AVL tree.

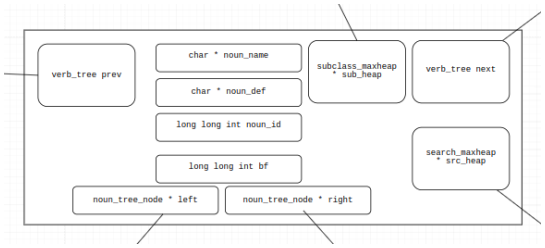

Fig. 2. Figure shows Structure diagram for Central Noun Tree node.

2) Verb Database Trees: Two additional AVL trees are maintained to store:

- verbs.
- verb descriptors.

These database trees serve as storage repositories to efficiently reuse allocated string memory. For example, if the verb ”plays” occurs in multiple input lines, then memory for the string ”plays” will be allocated only once and all the input lines will eventually point to the same memory (”plays”). Nodes in these trees are not interconnected, focusing solely on memory optimization.

3) Sub-class Max_heap: This includes the sub-classes of the noun. For example, animal_python,animal_human and animal_lion will be sub-classes of animal

- A pointer to the target noun node.
- *weight*: The importance of the subclass.

4) Edge Representation: Edges encapsulate the relationships between noun nodes. They are stored in the *query_max_heap* of the verb nodes. Each edge contains:

- *noun_ptr* : A pointer to the target noun node.
- *truth_bit*: A boolean indicating the truthfulness of the connection.
- *end_time*: The validity period of the connection.

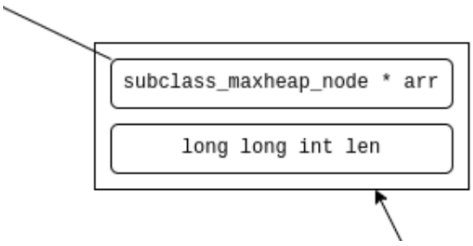

Fig. 3. Figure illustrates the Structure of Sub-class Max_heap.

- *weight*: The significance of the connection.

Search_maxheap nodes point to these edges. //

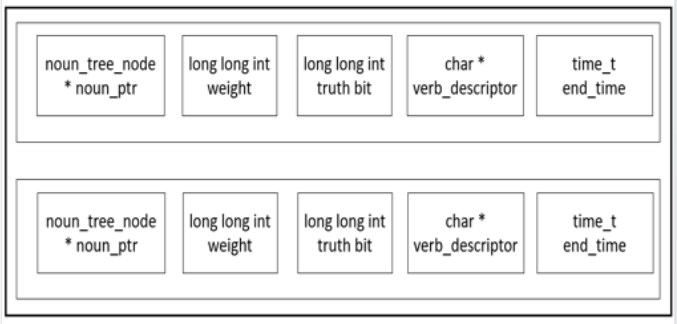

Fig. 4. Figure illustrates the Structure of Edge.

WorkFlow

This section will illustrate how an entry is made in the KG. For the input line : Computer Science includes wide range of Data Structures.

Here, the parser will identify "Computer Science" as noun1 , "Data Structures" as noun2, "includes" as verb and "wide range of" as verb descriptor. The algorithm will create a new noun, that is noun3 (noun3 = noun1 + "_" + noun2). For the given example, noun3 is "Computer Science_Data Structures" All the 3 nouns will be inserted to the central noun tree, if they do not exist. Addition of verb and verb descriptor is also done to the verb database tree.

Now to make a connection: It will make the front connection as noun1 -¿ noun3 with weight as front-weight. If the connection is already present, it will add the front-weight to the existing weight of the relation.As a result, the priority of the existing relation will increase.

Similarly, a back connection as noun3 -¿ noun1 will be made with weight as back-weight.

Making a connection involves storing the edge data in the query max_heap of the noun's verb and a pointer to the edge in search_max_heap of the noun.

Later, a pointer to noun3 is inserted in the subclass max_heap of noun2 with weight as front-weight. The definition of noun3 is stored in noun3 as a string.

V. ANALYSIS OF ALGORITHMS

A. Insertion

The parameters that affect the time complexity of Insertion are:

- Input noun1 string size = in_{n1}
- Input noun2 string size = in_{n2}
- Input verb size = in_v
- Input verb desc size = in_{vd}
- Input definition size = in_{def}

Firstly, creation of noun3 is done, which is a concatenation operation, which is a linear time complexity operation

$$T_{\text{concat}}(in_{n1}, in_{n2}) = O(in_{n1} + in_{n2}) = O(n) \tag{1}$$

Now, the nouns, verb, and verb descriptor are searched in the trees of the KG:

- Number of nouns in the tree = p
- Maximum string length of noun in the tree = k

AVL search will need to traverse till the leaf nodes in the worst case. Each comparison is a string comparison and hence takes linear time:

$$T_{\text{noun\_insertions}}(k, p) = O(k \lg p) = O(n \lg n) \tag{2}$$

The time complexity of searching in all the trees(centralized noun AVL tree , database trees that include verb and verb descriptor tree) will be:

$$T_{\text{total}}(n) = O(n \lg n) \tag{3}$$

If the nouns or verbs are not present, then their insertion will cost $O(n \lg n)$ time complexity in the worst case.
Next, we check if the connection already exists. This involves a linear search in various max_heaps:

$$T_{\text{searching\_heaps}}(n) = O(n) \tag{4}$$

Insertion into various heaps must be done if the edge does not exist.
Consider : size of the heap = k
then, insertion is done in $O(\lg k)$.

$$T_{\text{heap\_insertions}}(n) = O(\lg n) \tag{5}$$

VI. QUERYING AND TRAVERSAL

The approach for traversing the KG is "Weight proportional breadth-first traversal". When all connections of a node are to be traversed, the algorithm allocates an amount of lines to each child of that node, which that node can print. This allocation is done in proportion to the weights of the relations. For this, the **display_info_lines** function is used.

A. *display_info_lines*

This function takes

- *input_noun* : A string whose connections are to be displayed
- *total_lines* : The maximum number of lines to be printed.

For worst-case analysis, we consider *total_lines* to be arbitrarily large, and in the context of the C Programming Language, we have considered it to be `INT_MAX`.
`display_info_lines` does:

1) Search for *input_noun* : Search for the noun in the noun tree of the KG. This search is done to find a 100% matching string. As discussed above, this takes $O(kn \lg n)$ time where
k = string in the noun tree that has maximum length
n = number of nouns (noun nodes) in the noun tree.

If a perfect match isn't found, then another search is done, which gathers all nouns that are matching more than a certain threshold, and an array of choices is generated from this traversal This entire process also takes $O(kn \lg n)$ time.

2) *Display array of choices*: This is displayed as a "did you mean : " question, to correct spelling mistakes, in which the user is prompted with matching nouns. Depending on the user's choice, the traversal is done.

If the user wants to traverse all the k options, then the recursive function `print_info_lines` is called for each options.

B. *print_info_lines*

There are two cases in this function:

- Too few lines allocated for printing:
  Here, the input noun has less relations than the lines allocated to it. Hence, it will print all its relations and the function will return.
  If k lines are to be printed and the noun has m connections in its search max_heap, then in the worst case it will cost :
  $$T(k, m) = O(k \lg m) \tag{6}$$
  Note that $k \leq m$.

- There are enough connections to traverse:
  A *traversal_queue* data structure is used for this purpose.
  All the k connections of the noun are retrieved and lines are allocated to them.
  This takes $O(k)$ time.
  Later on, constructing the queue takes
  $$T(k) = O(k \lg k) + O(k) \tag{7}$$
  This is overall $O(k \lg k)$.

Now, once the *traversal_queue* has been constructed, the function `print_info_lines` will be called for each node in the queue.

If it occurs that all connections have been processed, and the number of lines allocated has not been exhausted, then we enter the subclass max_heap of the input noun. The same procedure is followed: line allocation, queue construction, and recursive function calls. The algorithm terminates when one of the following conditions is met:

- There are no remaining relations to print.
- Number of lines allocated to print is exhausted.

The KG is designed to ensure that the same relation is not traversed or printed more than once.
In the worst-case, every node in the KG will have a connection to the starting node of the breadth-first traversal, with the node connections arranged in a "linked list" fashion. For example, if the input sentences are structured as follows:

- noun_a verb1 noun_b ... noun3 = noun_ab
- noun_ab verb2 noun_c ... noun3 = noun_abc
- noun_abc verb3 noun_d ... noun3 = noun_abcd
- . . . and so on

Here we see that if the traversal begins at noun1, then it can be connected with at the most $n/3$ nouns. Hence, the function `print_info_lines` is called $n/3$ times which is $O(n)$.
The worst-case time complexity of `print_info_lines`, T_p, is
$$T(n) = O(n \lg n) \tag{8}$$
where n is the total number of nodes (nouns) in the graph. Since `display_info_lines` calls this function and the time complexity is asymptotically the same, `display_info_lines` is also worst case $O(kn \lg n)$
Typically, not all nodes of the graph are traversed, and the time required is often less!

VII. QUERYING THE KG

Our current KG framework is built utilizing structured data. User can shoot their desired queries in an interactive manner. The questions posed to the knowledge graph can be of various types such as:

1) What is Python?
2) What does python support ?
3) Which Programming language includes many libraries?
4) Which language is better than python ?
5) Describe Knowledge Graph

and many more...

RESULT

The graph constructed with the input data consistently produces accurate outputs.
KG can infer new knowledge by applying logical reasoning over the existing data, supporting both direct and transitive inference. This capability is crucial for applications such as question answering, where the system needs to derive answers that are not explicitly stated but can be inferred from the available data.
The algorithm is able to segregate the contexts from the data, and is able to give an accurate and reliable KG.
On a 16-bit processor: For small inputs (10,000 lines), the code constructed the graph in 1 second, and querying was much more faster. The code took less than 20 minutes to construct

a graph from 1,20,000 input lines The querying took about a millisecond to fetch the desired output effectively.

CONCLUSION

The structure of the KG eventually created ensures that correct relations are given for suitable query. It yields 100% accuracy.

It is an effective implementation of the concept of KG

The further goal is to create a KG from unstructured data with the help of NLP, making it more scalable, as illustrated in the image:

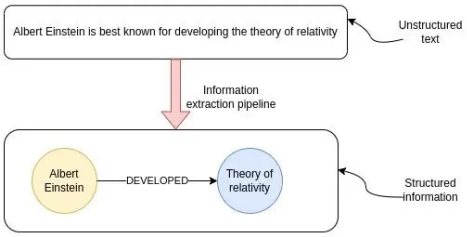

Fig. 5. Figure illustrates the working of NLP

REFERENCES

[1] RDF Schema https://www.w3.org/TR/2014/REC-rdf-schema-20140225/

[2] Knowledge Graph for Products http://dx.doi.org/10.1145/3336191.3371778

[3] Arun Krishnan. 2018. Making search easier: How Amazon's Product Graph is helping customers find products more easily. Amazon Blog. https://blog.aboutamazon.com/innovation/making-search-easier

[4] R. J. Pittman, Amit Srivastava, Sanjika Hewavitharana, Ajinkya Kale, and Saab Mansour. 2017. Cracking the Code on Conversational Commerce. eBay Blog. https://www.ebayinc.com/stories/news/cracking-the-code-on-conversational-commerce/

[5] Qi He, Bee-Chung Chen, and Deepak Agarwal. 2016. Building The LinkedIn Knowledge Graph. LinkedIn Blog. https://engineering.linkedin.com/blog/2016/10/building-the-linkedin-knowledge-graph

[6] Edgar Meij. 2019. Understanding News using the Bloomberg Knowledge Graph. Invited talk at the Big Data Innovators Gathering (TheWebConf). Slides at https://speakerdeck.com/emeij/understanding-news-using-the-bloomberg-knowledge-graph

[7] https://medium.com/analytics-vidhya/introduction-to-knowledge-graphs-and-their-applications-fb5b12da2a8b

[8] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia D'amato, Gerard De Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan Sequeda, Steffen Staab, and Antoine Zimmermann. 2022. Knowledge graphs. ACM Computing Surveys. https://arxiv.org/pdf/2003.02320

[9] The C Programming Language https://github.com/auspbro/ebook-c/blob/master/The.C.Programming.Language.2Nd.Ed%20Prentice.Hall.Brian.W.Kernighan.and.Dennis.M.Ritchie..pdf