Kernel - Level Semantic Search with Knowledge Graphs

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

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

Prajwal Bhosale
Department of Computer Science
COEP Technological University
Pune, Maharashtra, India
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.

A. Abbreviations and Acronyms

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

I. Introduction

KGs are advanced data structures that represent information in a network of interconnected entities and their relationships. They consist 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.

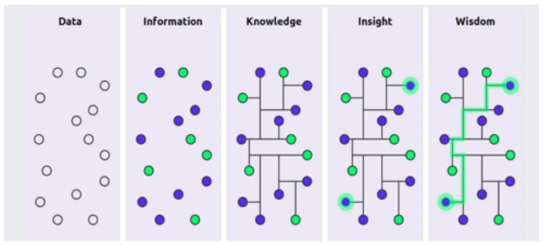


Fig. 1. Figure shows example of KG construction.

An example of KG construction is shown in the figure below

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 :

- **E-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 KG that powers financial data analytics, including sentiment analysis for companies based on current news reports and tweets.

This paper focuses on the structural implementation of a KG. It takes input from users and serves as a critical source of knowledge across various domains.
The primary contributions of this study are summarized as follows:

- It introduces a structural framework for constructing a KG at the kernel level without relying on external libraries.
- The developed KG can manage diverse types of queries, providing precise and relevant responses.

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

A primary reference for our study was the paper "Knowledge Graphs" [8] by Aidan Hogan et al. The paper features a rich bibliography with over 400 references, providing detailed explanations on KG concepts such as graph modeling, ontologies, deductive knowledge, graph embeddings and more.

The paper "The Potential of the PID Graph" [15] presented and highlighted the concept and importance of "identity" in the context of data parsing and extraction(as discussed in section..).

The paper "Context-Aware Temporal Knowledge Graph Embedding" [16] examined the concept of temporal context (as discussed in section ..).

With reference to [8], we studied various graph models, including heterogeneous graphs [17], property graphs [11], and directed-edge labelled graphs [8]. This paper delves into the implementation of directed-edge labelled weighted graphs with further additions and improvements.

"Introduction to Algorithms" [13] provided the knowledge needed to select the appropriate data structures and efficient algorithms. "The C Programming Language" [9] was used to implement these algorithms in C.

These insights from the literature review, inspired the authors to present the structural implementation and significance of KGs

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 KG should be able to handle and recognize the following characteristics of data:

- **Context of entities.** [8]
  For example, if the word "bank" is considered, 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.
- **Weight of the connection**
  The connection with highest weight should be displayed first.
  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 "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 constraints** [8], [16]
  For example, the statement "XYZ is the Prime Minister of India" will be valid only for a certain period of time(say five years) and then it may change. After expiration, the relation should be redundant.
- **Identity** [8], [15]
  For example, "Arjun plays piano" and "Arjun plays guitar", although the name "Arjun" has been repeated, but in the real world it may refer to different entities. The analyzer should assign unique identifiers to different entities.
- **Misspelled Words Queries**
  The query analyzer 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 designed input data set contains the following fields: front-weight, inference, truth-bit, noun1, noun1_id , verb , verb descriptor, noun2, noun2_id, back-weight, definition, end-time.

IV. Architecture

A. Data Structures Used

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-centric knowledge representations and their interconnections via verbs.

1) AVL Tree: AVL trees [13] are among the most efficient data structures for searching and inserting nodes(data) due to their self-balancing nature. The predictable performance of AVL trees makes them a powerful and widely used choice in computer science. They are particularly used in scenarios requiring fast and reliable search operations such as database indexing.

The insertion and search operations in an AVL tree have a time complexity of $O(\lg n)$, where "n" represents the number of nodes already present in the AVL tree.

2) Maximum Heap: Max_heaps [13] are highly efficient data structures for managing and retrieving the maximum weighted element . They maintain a structure similar to the complete binary tree structure, ensuring that the most weighted element is always at the root, which allows for constant time retrieval of the maximum value. It enables $O(1)$ time complexity for retrieval.

Inserting and deleting elements in a max heap is efficient, with both operations having $O(\lg n)$ time complexity.

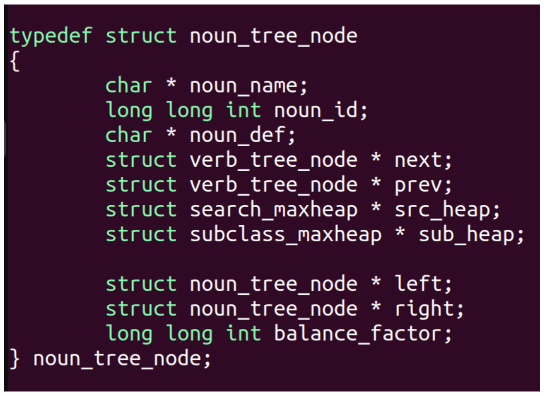
All these make Max_Heap quite useful in applications like scheduling algorithms,resource management and real-time simulations.

B. Structural Implementation

1) ***Central Noun Tree****:* The core of our KG 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.

2) ***Prev and Next Verb Trees****:* As explained previously, each noun_tree node has its own next and previous verb_trees. These contain connections to edges as well as some information about the verb being stored Each verb_tree node comprises of:



```
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. 2. Structural implementation in C

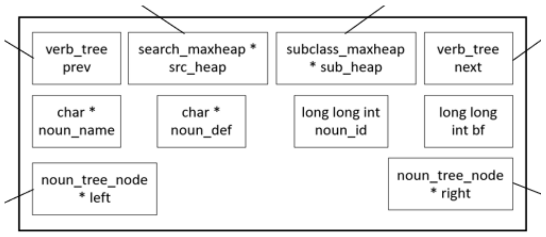


Fig. 3. Figure shows Structure diagram for Central Noun Tree node.

- *verb_name*: Pointer to a string in the database tree of verbs
- *qheap*: A max_heap that contains essential edge information
- *left, right, balance_factor*:for maintaining AVL tree.
- *next_verb_tree*: An AVL tree containing verbs that form front links to other nouns.

*3) **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 references to ”plays” will eventually point to the same memory location. Nodes in these trees are not interconnected, focusing solely on memory optimization.

*4) **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.

*5) **Edge Representation, search max_heaps**:* 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.
- *weight*: The significance of the connection.

Search_max_heap nodes point to these edges. //

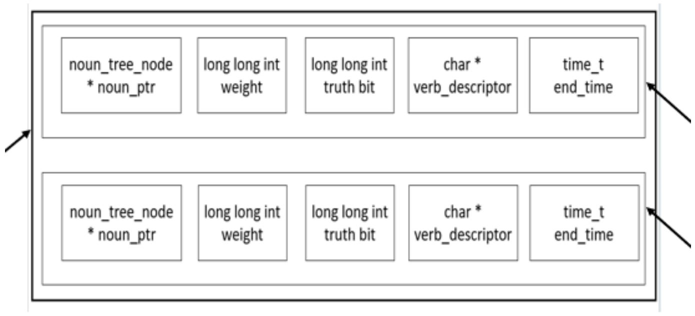


Fig. 4. Figure illustrates the Structure of Edge.

IV. WorkFlow

A. Example of Entry Creation in the KG

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” (first noun of the sentence) and ”Data Structures” as ”noun2” (second noun of the given input sentence). Phrases like ”includes” will be classified as ”verb” and ”wide range of” will be classified as ”verb-descriptors”.

Now there comes a need to store this sentence into the memory where our structural concept of KG comes into use. noun1 and noun2 (i.e., ”Computer Science” and ”Data Structures”) will be inserted into the central AVL noun tree (explained in section ....).

A new noun i.e., noun3 also gets formed: i.e., noun1 + ”_” + noun2 i.e., ”Computer Science_Data Structures” and will be inserted in the central AVL tree (reason prescribed in section ...).

Each noun should only be inserted if it does not already exist in the central AVL tree. This prevents duplicates, ensuring no redundant nouns are stored, thus optimizing memory usage.

If the noun already exists, connections will be made through the existing noun itself.

The verb ”includes” and verb descriptor ”wide range of” are also stored in a similar manner in the Verb Tree and Verb descriptor Tree (explained in section ....).

Further, irrespective of which branch of the central AVL tree the noun gets inserted, the connections among them are made, i.e.:

$$\text{noun3} \longrightarrow \text{verb} \longrightarrow \text{verb\_descriptor} \longrightarrow \text{noun2}$$

(explained in section ....)

In instances where the input line ”Computer Science includes a wide range of Data Structures.” is repeated, instead of creating new connections, the weights associated with the existing connections are incremented. This ensures that no redundant data and connections are stored.

This structure helps maintain the relationships within the KG, ensuring that each entry is efficiently organized and connected.

B. Output Extraction and Querying the KG

As an end-user, one may seek the most relevant response to their query. Users might ask questions such as:

- ”Computer Science includes?”
- ”? includes Data Structures”
- ”Computer Science includes Data Structures: True or False?”
- Or broader queries like: ”Describe Computer Science” or ”Computer Science?”

Queries may also contain spelling errors, requiring the system to provide the most relevant output and potentially prompt the user with a ”Did you mean?” suggestion.
For example, if a user inputs: ”Comptr Science includes?”
the system should prompt:
”Did you mean: Computer Science includes?”
and proceed based on the user’s response.

As developers, it is crucial to handle all these types of queries efficiently. Therefore, functions like `display_info_lines` and `print_info_lines` are used, which are detailed in subsequent sections.

Responding to questions like ”Describe Computer Science” could overwhelm the user with an extensive amount of data, as the KG may contain thousands of entries related to Computer Science.
To manage this, the algorithm prompts the user with a follow-up question: ”How many lines of information do you need?” If the user specifies 10 lines, the system will display the 10 most important and weighted connections.
There is also an option for the system to display all connections in decreasing order of their weights.

V. Querying and Traversal

The approach for traversing the KG is ”Weight Proportional Breadth-First Traversal.”

User can shoot their desired queries in an interactive manner.

Suppose the KG receives the input ”Computer Science includes ?” and the user requests: ”I want 15 lines of data.” Although the KG may contain thousands of connections, for simplicity, let’s assume it has three direct connections as shown in the figure

The dotted line ”returns 0” is not a connection, and is rather a function call’s return value

Given the limited number of lines to be displayed, the KG will prioritize the connections based on their weights using a max heap structure and will allocate ratios to the nouns of the next connections. In this case, the proportion is 3 : 2 : 1 because the weights of the connections are in the ratio 3 : 2 : 1 (i.e., Data Structures, Programming Language, Interesting). The allowed lines will be distributed to the next nodes in

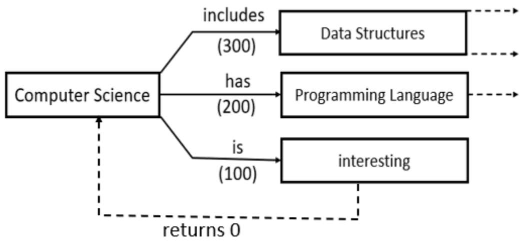


Fig. 5. Figure illustrates an example for graph traversal

these proportions, and this process will continue recursively until all the requested lines have been displayed.

Now, in the above example, *total_lines* allocated are 15. Hence, the following distribution will occur for each connection

- ”Computer Science includes Data Structures” : 1 line (for the connection itself), 6 for ”Data Structures”
- ”Computer Science has Programming Language” : 1 line, 4 for ”Programming Language”
- ”Computer Science is Interesting” : 1 line, 2 for ”Interesting”

One line is consumed by the direct connections themselves, and further lines are given for the next level connections. A traversal queue will get constructed and recursive calls of `print_info_lines` will be made for each of ”Data Structures”, ”Programming Language” and ”Interesting”.

Assuming that ”Data Structures” had only 2 direct connections, then 2 lines will get printed (out of the 6 that were allocated to it) and the remaining lines will get re-allocated to ”Programming Language”.

This means that the recursive call of ”Programming Language” will have 8 lines (4 allocated before, 4 re-allocated from ”Data Structures”)

Here ”Programming Language” has 1 line to print, hence the remaining 7 lines get re-allocated to ”Interesting”.

”Interesting” has no lines to print, and hence returns the remaining lines (9 lines) to the noun ”Computer Science”.

Now, ”Computer Science” constructs a traversal queue by utilising its sub-classes, that are not shown in the figure, and similar recursive calls occur for the subclass nouns.

An important thing to note here is that sub-classes are only traversed for the first level, and further levels do not access subclass heaps.

This feature of the algorithm ensures that connections are not printed more than once, and the same nodes are never traversed. In the end, the function returns the number of lines that got printed by it

The functions used for traversal are given below

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.

`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.

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

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. For the example, refer (...)

The user can then choose to select one or multiple options from the provided suggestions and get the desired output.

B. `print_info_lines`

There are two cases in this function:

- **Too Few Lines Allocated for Printing:**
  Here, the input noun has fewer relations than the lines allocated for it. Therefore, it will print all its relations, and the function will return.
  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 allocated lines has not been exhausted, then the algorithm enters 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.
  - The number of allocated lines to print is exhausted.

  The KG is designed to ensure that the same relation is not traversed or printed more than once.

VI. 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
- Length of the string to be searched = 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 $O(\lg k)$.

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

B. Querying

Querying involves two functions, `display_info_lines` and `print_info_lines`, as mentioned in the previous section. The function `display_info_lines` is just a wrapper on `print_info_lines`

For worst-case analysis, we consider *total_lines*, the input to these querying functions to be arbitrarily large, and in the context of the C Programming Language, we have considered *total_lines* to be INT_MAX.

1) `display_info_lines:` :

Searching for *input_noun* takes overall $O(k \lg n)$ time where
k = size of the input string
n = number of nouns (noun nodes) in the noun tree.
If a perfect match isn't found, the second search will also take worst case $O(pn \lg n)$ where
p = string in the noun tree that has maximum length
After that, the array of choices is created and displayed. If the user wants to traverse all the k choices, then the recursive function print info lines is called for each choice noun.

2) `print_info_lines:` :

- **Too Few lines allocated for printing** :
  Here, the input noun has fewer relations than the lines allocated for it. Therefore, 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 k connections of the noun are retrieved, and lines are allocated to them. This operation takes $O(k)$ time. Later, constructing the queue takes:

  $$T(k) = O(k \lg k) + O(k) \tag{7}$$

  This is overall $O(k \lg k)$
  Note that $k > m$ Once the traversal queue has been constructed, the function `print_info_lines` will be called for each node in the queue.
  Since `display_info_lines` calls this function and the time complexity is asymptotically the same, `display_info_lines` is also worst case $O(n \lg n)$
  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` is

  $$T(n) = O(n \lg n) \tag{8}$$

Since `display_info_lines` calls this function and the time complexity is asymptotically the same, `display_info_lines` is also worst case $O(n \lg n)$
Typically, not all nodes of the graph are traversed, and the time required is often less!

Hence, the conclusion is as follows:

1) **Insertion** From equations (1) - (5), worst case time complexity for insertion is

$$T_{insertion}(n) = O(n \lg n)$$

2) **Querying** From equations (6) - (8), worst case time complexity for querying is

$$T_{querying}(n) = O(n \lg n)$$

VII. 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 one 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.

VIII. 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

IX. REFERENCES

REFERENCES

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

[2] Da Xu, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan "Product Knowledge Graph Embedding for E-commerce" January 2020 available at : https://doi.org/10.1145/3336191.3371778

[3] Arun Krishnan "Making search easier", Amazon Blog, August 17, 2018, available at https://blog.aboutamazon.com/innovation/making-search-easie

[4] R. J. Pittman, Amit Srivastava, Sanjika Hewavitharana, Ajinkya Kale, and Saab Mansour, "Cracking the Code on Conversational Commerce", eBay Blog, April 6, 2017 , available at : https://www.ebayinc.com/stories/news/cracking-the-code-on-conversational-commerce/

[5] Qi He, Bee-Chung Chen, and Deepak Agarwal "Building The LinkedIn Knowledge Graph", LinkedIn Blog, October 6, 2016 , available at : https://engineering.linkedin.com/blog/2016/10/building-the-linkedin-knowledge-graph

[6] Edgar Meij "Understanding News using the Bloomberg Knowledge Graph" BloombergEngineering, May 14, 2019 , available at https://speakerdeck.com/emeij/understanding-news-using-the-bloomberg-knowledge-graph

[7] Ergeta Muca, "Introduction to Knowledge Graphs and their Applications" Published in Analytics Vidhya Dec 10, 2019, available at : https://medium.com/analytics-vidhya/introduction-to-knowledge-graphs-and-their-applications-fb5b12da2a8b

[8] Aidan Hogan, Eva Blomqvist, et al, "Knowledge Graphs", ACM Computing Surveys, 11 September 2021, available at : https://arxiv.org/pdf/2003.02320

[9] Brian Kernighan, Dennis Ritchie "The C Programming Language", 2nd edition, 8 April 1988

[10] Samadrita Ghosh,'Knowledge Graphs — What, Why, and How", medium , Dec 19, 2022, available at : https://samadritaghosh.medium.com/knowledge-graphs-what-why-and-how-84f920316ca5

[11] Foundations of Modern Query Languages for Graph Databases https://dl.acm.org/doi/pdf/10.1145/3104031

[12] Ramanathan V. Guha, Rob McCool, and Richard Fikes. 2004. Contexts for the Semantic Web. In The Semantic Web - ISWC 2004: Third International Semantic Web Conference, Hiroshima, Japan, November 7-11, 2004. Proceedings (Lecture Notes in Computer Science), Frank van Harmelen, Sheila McIlraith, and Dimitri Plexousakis (Eds.), Vol. 3298. https://www.researchgate.net/publication/221466471_Contexts_for_the_Semantic_Web

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. "Introduction to Algorithms" 4th edition, April 2022

[14] Félix Melchor, Santos López et al "Literature review about Neo4j graph database as a feasible alternative for replacing RDBMS" ,December 2015 ,available at : https://www.researchgate.net/publication/307180380_Literature_review_about_Neo4j_graph_database_as_a_feasible_alternative_for_replacing_RDBMS

[15] Helena Cousijn, Ricarda Braukmann et al. Connected Research: The Potential of the PID Graph, Patterns, Volume 2, Issue 1, 2021, 100180, ISSN 2666-3899, available at : https://www.sciencedirect.com/science/article/pii/S2666389920302440

[16] Liu, Yu and Hua"Context-Aware Temporal Knowledge Graph Embedding" ,October, 2019 available at : https://www.researchgate.net/publication/337245161_Context-Aware_Temporal_Knowledge_Graph_Embedding

[17] Rana Hussein, Dingqi Yang, and Philippe Cudré-Mauroux. 2018. Are Meta-Paths Necessary? https://dl.acm.org/doi/10.1145/3269206.3271777