Kernel-Level Semantic Search with Knowledge Graphs

Arjun Deodhar*, Arnav Prasad†, Prajwal Bhosale‡, Pratiksha Deshmukh§
Department of CSE, COEP Technological University
Pune, Maharashtra, India
*arjundeodhar@gmail.com, deodhark22.comp@coeptech.ac.in
†arnav.prasad.ap@gmail.com, arnavp22.comp@coeptech.ac.in
‡pjlbhosale@gmail.com, bhosalepp22.comp@coeptech.ac.in
§dpr.comp@coeptech.ac.in

***Abstract*—Various types of knowledge exist 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 where each node represents real-world entities and edges logically depict the relationships between the nodes. The graph can be directed or un-directed, depending on the organization's needs. 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 Graph, kernel-level implementation, AVL, heap

A. Abbreviations and Acronyms

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

I. INTRODUCTION

KGs are advanced data structures. They are designed to facilitate and improve information retrieval, question-answering, and semantic search capabilities. The concept of the KG gained significant recognition in 2012 when Google [6] publicly credited their search solution to the use of KG.

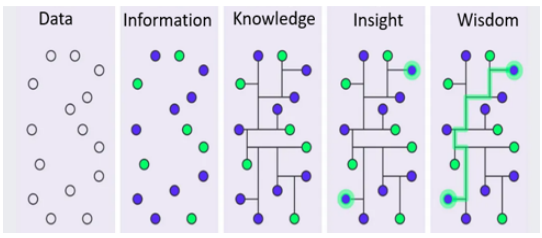

Fig. 1: Transforming Raw Data [1] into Knowledge

The relationships in KG are often labeled to provide context and meaning, forming a rich, structured representation of knowledge. They support 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.

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 algorithm presented in this paper has been evaluated and analyzed using a substantial fabricated dataset of inputs. For querying on a KG (dataset size: 917,568 lines of data), each query operation takes 0.174399 seconds.

After reviewing various research papers, many of them included the abstract concept of KG, but they lacked detailed structural implementation at the kernel level. Consequently, there arose a need to construct a fabricated input data set [10], which is provided in the GitHub link.
The paper aims to provide a road-map for a thorough exploration of the fundamental implementation aspects of KGs.

The paper is structured as follows:

Section 2 Introduces the relevant work and study on KG in the literature. Section 3 Describes the real-world data that should be handled by KG. Section 4 Explains the proposed memory architecture of KG and its data structures. Section 5 Demonstrates the proposed data flow in KG. Section 6 Elaborates on analyzing the algorithms used in KG. Section 7 Presents the pseudo-code of the algorithms. Section 8 Includes the results and performance analysis of KG. Finally, Section 9 Concludes the study.

II. LITERATURE SURVEY

Aidan Hogan et al [7] suggested concepts of KG such as graph modeling, ontologies, deductive knowledge, graph

embeddings, and more. The paper features a rich bibliography with over 400 references. With reference to [7], we studied various graph models, including heterogeneous graphs [14], property graphs [11], and directed-edge labeled graphs [7]. This paper delves into the implementation of directed-edge labeled weighted graphs with further additions and improvements.

Ergeta Muca [6] outlined the types of data sources commonly used for KG as structured inputs. The paper emphasized the advantages of KGs over traditional databases.It explained the concept of KGs, their origin, and highlighted their key applications.

Samadrita Ghosh [1] puts forward the concept of KGs, highlighting the key differences between a "graph" and a "Knowledge Graph." It also discusses the relationship between KGs and artificial intelligence.

Helena Cousijn, Ricarda Braukmann, et al. [12] presented and highlighted the concept and importance of "identity" in the context of data parsing and extraction.

Liu, Yu, and Hua [13] summarized the concept of temporal context.

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

III. METHODOLOGY

Data can be of variable length, context, grammar, etc.
A KG is not just a collection of data [7] but also contains hierarchical structures, contexts for data, as well as unique identifiers for real word entities.
The constructed KG should be able to handle and recognize the following characteristics of data:

- **Context of entities.** [7]
  For example, if the word "bank" is considered, the context can be a river-side bank or bank that involves financial transactions, or specifically Bank of Maharashtra, or may have some other context.
- **Weight of the connection**
  The connection with the highest weight should be displayed first. The text analyzer should be able to assign front and back weights accordingly. 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).
- **Temporal constraints** [7], [13]
  For example, the statement "XYZ is the Prime Minister of India" will be valid only for a certain period (say five years) and after that, it may change. After expiration, the relation should be redundant.
- **Identity** [7], [12]
  For example, "Arjun plays piano" and "Arjun plays guitar", although the name "Arjun" has been repeated, 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. PROPOSED ARCHITECTURE

A. *Data Structures Used*

The implementation in C programming language [8], employs structures and pointers for AVL trees and arrays for heaps, providing a robust framework for managing non-centric knowledge representations and their interconnections via verbs.

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

*2) **Maximum Heap**:* Max_heaps are highly efficient data structures for managing and retrieving the maximum weighted element. The most weighted element is always at the root enabling $O(1)$ time complexity for retrieval

Inserting and deleting elements in a max heap has $O(\lg n)$ 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, 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:

- *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.

V. WORKFLOW

This section will illustrate how an entry is made in the KG as well as how queries are processed and the graph is traversed.

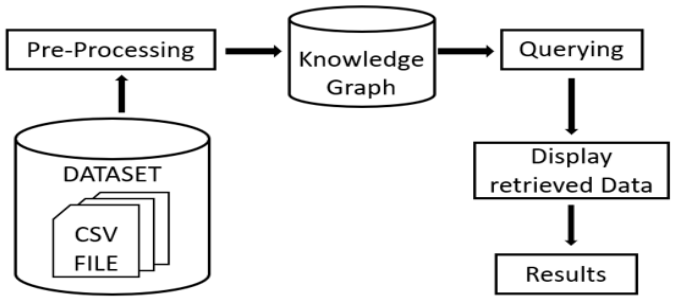

Fig. 2: The figure illustrates the workflow

A. *Example of Entry Creation in the KG*

Consider the input line: "Computer Science includes a wide range of Data Structures.", which is stored in the CSV file

Pre-processing involves the work of a parser and the insertion of the data points into the KG

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 identified as "verb" and "wide range of" will be classified as "verb-descriptors".

Now there comes a need to store this sentence in the memory where our structural concept of KG comes into use. noun1 and noun2 will be inserted into the central AVL noun tree.

A new noun, $noun3 = noun1 + "\_" + noun2$ gets formed i.e., "Computer Science_Data Structures" and will be inserted in the central AVL tree.

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 the verb descriptor "wide range of" are similarly stored in the Verb Tree and Verb descriptor Tree.

Suppose the front weight of the connection is 500 and the back weight is 300.

The memory representation for the given example is given in the figure below.

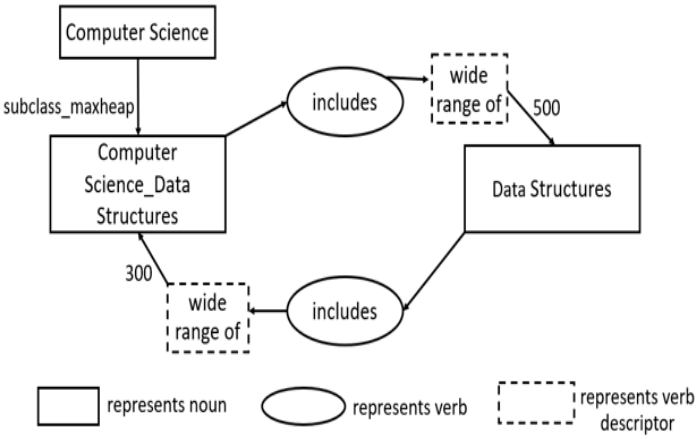

Fig. 3: Figure depicts the memory representation of KG

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

noun3 ⟶ verb ⟶ verb_descriptor ⟶ noun2

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.

B. *Output Extraction and Querying the KG*

1) Types of Queries in KG: As an end-user, one may seek the most relevant response to their query. The "Query" stage involves asking the KG various types of queries like :

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

2) Handling Spelling Errors in Queries: 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 search_noun_and_print_lines() and print_info_lines() are used, detailed in subsequent sections. These are called once the query is processed properly.

3) Keeping the Response concise: 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.

Suppose the query analyzer receives the input ”Describe Computer Science” and the user also 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 below.

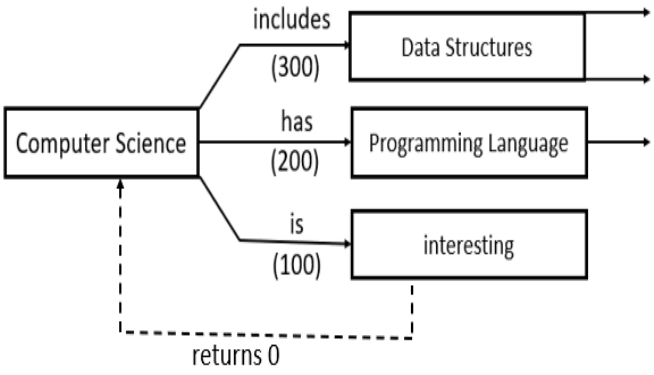

Fig. 4: Figure illustrates an example of graph traversal

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

The traversal algorithm will prioritize the connections based on their weights. It will allocate lines in proportion to the weights. Here, the proportion is 3 : 2 : 1 (i.e., Data Structures, Programming Language, Interesting).
The allocated lines will be distributed to the next nodes in these proportions and this process will continue recursively until all the requested lines have been displayed.

Now, in the above example, the 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 for itself, 4 for ”Programming Language”
- ”Computer Science is Interesting”: 1 line for itself, 2 for ”Interesting”

One line is consumed by the direct connections themselves, and further lines are given for the next-level connections. 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 reallocated 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 utilizing its sub-classes, which 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 the same connections are not printed more than once. In the end, the function returns the number of lines that were printed by it.

VI. Implementation of Querying and Traversal

The approach for traversing the KG is ”Weight Proportional Breadth-First Traversal.” Users can interactively query the graph.
search_noun_and_print_lines() takes input:

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

print_info_lines() also takes similar inputs, but instead of *input_noun* it takes a pointer to the *input_noun*

The pseudo-code for both functions are as follows:

Algorithm 1 search_noun_and_print_lines

```
procedure search_noun_and_print_lines(input_noun, total_lines)
    Search for input_noun in central_noun_tree
    matching_noun ← search_result_pointer
    if (matching_noun == NULL) then
        Perform another search
        Display array of choices to the user
        for selected_option in displayed array do
            call print_info_lines(selected_option)
        end for
    else
        call print_info_lines(matching_noun)
    end if
end procedure
```

Algorithm 2 print_info_lines

```
procedure print_info_lines(total_lines, input_noun_ptr)
    if (Too few lines allocated for printing) then
        for (Most important connections in search_max_heap) do
            print connection
        end for
        return(number of connections printed)
    else
        construct queue for traversing all connections
        for (each node in traversal queue) do
            call print_info_lines(node)
        end for
    end if
    if (total_lines not exhausted) and (traversal is in first level) then
        for (each node in subclass max_heap) do
            call print_info_lines(node)
        end for
    end if
    return(number of lines printed)
end procedure
```

VII. 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}

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

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

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

In the worst case, searching in the AVL tree will require :

$$T_{\text{noun\_searching}}(k, p) = O(k \lg p) = O(n \lg n) \quad \ldots(2)$$

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

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:

$$T_{\text{searching\_heaps}}(n) = O(n) \quad \ldots(3)$$

Insertion into various heaps must be done if the edge does not exist.

$$T_{\text{heap\_insertions}}(n) = O(\lg n) \quad \ldots(4)$$

B. *Querying*

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) ***search_noun_and_print_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 $O(p \lg n)$ 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**:
  If k lines are to be printed and the noun has m connections in its search max heap, ($k \leq m$) then in the worst case, it will cost:

$$T(k, m) = O(k \lg m) \quad \ldots(5)$$

- **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:

$$T(k) = O(k \lg k) + O(k) \quad \ldots(6)$$

  This is overall $O(k \lg k)$
  Once the traversal queue has been constructed, print_info_lines() will be called for each node in the queue.
  In the worst case, the starting noun of traversal can be connected with $O(n)$ nodes. Hence, the worst-case time complexity of print_info_lines() is

$$T(n) = O(n \lg n) \quad \ldots(7)$$

search_noun_and_print_lines() is also worst case $O(n \lg n)$, since its time complexity is asymptotically the same.

Hence, the conclusion is as follows:

1) **Insertion** From equations (1) ... (4)

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

2) **Querying** From equations (5) ... (7)

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

VIII. RESULTS

The graph constructed with the input data consistently produces accurate outputs.

The analysis utilized an 11th Gen Intel(R) Core(TM) i5-1135G7 processor, featuring 8 cores and VT-x virtualization support. The insertion algorithm segregates the contexts from the data, supports direct and transitive inference, and can give an accurate KG.

The algorithms presented in Sections 5 and 6 underwent thorough analysis across various parameters, including different numbers of input lines. The analysis focused on the time required to create and query a KG. The table below puts forward the time analysis of KG, showcasing its remarkably fast response.

Number of input lines	Time taken for KG creation (seconds)	Time taken for querying(seconds)	Total lines displayed
11328	0.3888892	0.008026	6491
33984	1.167377	0.013361	11111
101952	9.375197	0.021687	27347
305856	212.202868	0.059075	82043
917568	2795.292009	0.174399	246131

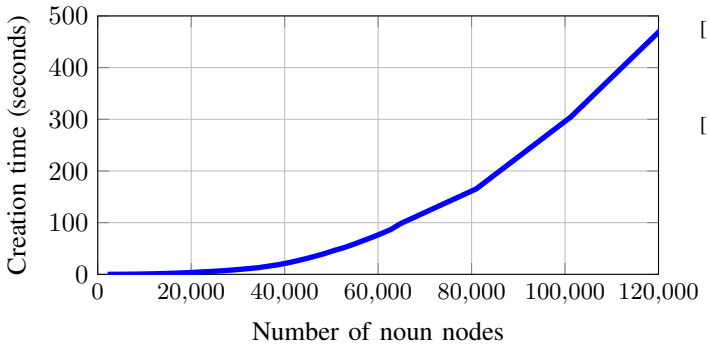

Fig. 5: Graph explains the performance analysis of creating the graph

As depicted in Figure 5, the graph conforms to the mathematical analysis of the algorithm presented in this paper.

IX. CONCLUSION

This study puts forward a fundamental kernel-level implementation of a KG. The proposed system demonstrates that querying a KG, which stores millions of data points, is remarkably fast. The structure of the KG is designed to ensure that accurate relationships are provided for given queries, achieving 100% accuracy.

The present work can be improved by integrating KG with NLP, which would enable the construction of KG from unstructured data.

REFERENCES

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

[2] Da Xu, Chuanwei Ruan et al, "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 et al, "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] 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

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

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

[9] 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

[10] Fabricated Data Set available at : https://github.com/Arnav2Prasad/C_Knowledge_Graph_project

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

[12] 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

[13] 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

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