SAHYADRI International Journal of Research, Vol. 3, Issue 2, 2017
ISSN: 2456-186X, Published Online December, 2017 (http://www.sijr.in/)

Research Paper

Information Extraction for Graph Data Using Domain Specific Ontology

N. Balaji[1]* and N. Megha[2]
[1] ISE Dept., Sahyadri College of Engineering & Management, Adyar, Mangaluru - 575007
[2] ECE Dept., Sahyadri College of Engineering & Management, Adyar, Mangaluru - 575007
*Email: balaji.hiriyur@gmail.com

Abstract

Ontology provides a structured way of describing knowledge. Ontology behaves like a Knowledge Base (KB) with respect to any specific domain. If we have a document repository with respect to any domain the same information can be converted as an ontology by using concepts and its relationships between them. RDF and RDFS have recently become very popular frameworks for representing data and meta-data in the form of the domain description. RDF/RDFS data can also be thought of as graph data. Here, our main focus is on keyword based querying on RDF/RDFS data. The studied and implemented approach adopts a reduced exploration mechanism, where we try to find out closely related vertices using sub graph construction phase, then removing unwanted vertices/edges and joining by using suitable join vertices. We explore the type of *(type/subclassof)* relationship during the construction of output from the given query.

Keywords: Graph, RDF, RDFS, Ontology, WordNet, DBLP, YAGO2, SAX Parser.

1 Introduction

Information Extraction *(IE)* deals with the representation, storage, organization of data and access to information items. The representation and organization of the information or data should provide the user with easy access to the information in which the user is interested. Data retrieval in the context of an Information Extraction *(IE)* system consists mainly of determining which documents of a collection contain the keyword(s) in the user query.

An *IE* system is mainly concerned with retrieving information from a particular set of domain documents [1]. In many approaches, even the database is modeled as a graph $G\ (V, E)$, where tuples are taken as vertices and key relationships are taken as the edges. The documents which may be either in the form of text documents or graph documents are represented as Resource Description Framework (RDF) triples. XML data and relational sources can be modeled by using the graph $G\ (V, E)$ [2].

RDF/RDFS is based on graph-oriented data schema and it can be represented in the form of triples and visualized as a directed graph structure. Each triple is an edge from a subject to an object with the predicate as the label of an edge [2]. A huge collection of repositories of data are modeled using the RDF framework. The significant examples are Personal Information system where emails, documents and photos merged into a graph and biological databases. There is a large amount of RDF data available on the web stored in the form of triples. These triples are represented as graphs so that an *IE* system can use the graphs and extract the relevant information. There are many techniques that extract information from the graph. One of them is *Steiner-Tree* method which extracts the relevant graphs corresponding to a particular query [4, 5]. A major disadvantage of this method is that it extracts sub-graph containing loops and cycles which further can lead to redundancy in the results.

An interesting type of query which is related to graph data is *relatedness* query [3], that explores a series of relationships between graph elements. In this type of query, the keywords correspond to *anchor* points in the graph and we need to extract a sub-graph connecting these *anchor* points. RDF query languages like $SPARQL$ [2], through which any kind of complex requirements can be expressed, requires the background and precise knowledge of schema underlying data.

Enormous data or information about specific domain are available on the web in the form of text and tables as well. Since, the data in graph structure is not readily available, query processing on this text data may be a challenging task. The construction of queries has to be done by using an ontology or one needs the knowledge of query language called **SPARQL**. To overcome this problem, we are introducing a different approach named *Graph Exploration* [4, 5] that will extract a sub-graph corresponding to a query comprising of a set of keywords, from the graph constructed using the triples. These sub-

graph contain vertices as query keywords and the edges denoting the *(type/subclassof)* relationships between the vertices.

After extraction of sub-graph for every keyword, the reduction step removes the vertices and edges which are not relevant to the query. After reduction, the merging process will merge the sub-graph based on two criteria *least-cardinality* and *similar vertices*. Two vertices are said to be similar for the constructed sub-graph, if they have same type and correspond to the same query keyword. *Cardinality* of sub-graph denotes the number of vertices present in it. In the merging process, the least *cardinality* sub-graph is merged to the other sub-graph containing the similar vertex [5, 6].

2 Motivation

Keyword search on relational data uses different methods for extracting information from the input query, the most significant one being tree structured data and graph structured data. The use of *Steiner-tree* problem extracts the output but this may contain loops and cycles and hence, this may lead to wrong output. We need to identify a different approach to explore correct output. To this end we used the approach called *Graph exploration* [3] process. The adopted approach for the construction of sub-graph or answer graph exploit semantic relationship *(type/SubClassOf)* as the keyword might represent the parent *type* and the instance data might represent the $SubClassOf$.

In order to limit the number of steps in *Graph Exploration* step, we fix the distance d to restrict the exploration for finding the vertices related to the query. By using this, d-neighborhood [3] of entities extracts possible sub-graph which connects the entities computed. We have two major issues in the *Graph Exploration* process. First, if calculated d is small then the graph becomes disconnected and then we cannot extract the links. Second, for the given set of keywords, some keywords will appear to be closer and some will be farther; due to this issue, calculation of d is not exploited [3].

Existing strategies for keyword queries on RDF data have the following limitations:

- Incorrect retrieval of sub-graph,
- Inefficient path exploration,
- Exploitation of irrelevant semantic characteristics of RDF graph and
- Pose restrictions on choosing neighborhood vertex.

A precise representation of RDF graph is the one in which types and classes denote vertices, properties denote edges, keyword queries can be answered by exploiting semantic properties [2].

RDF data set can be considered as an ontology which is used for solving the semantic dispute when separate data sources are integrated. It provides a common interface to *IR* and it can also hold information about the concepts and relations of different data sources in order to extract the information from the ontology. For this we need to understand about a query language such as **SPARQL** [2], this **SPARQL** query is explicitly used whenever information needs to be extracted.

There are three main phases in ontology based *IR* system [2] namely, ontology creation, ontology mapping, and query service. Extraction of triples from the query has an important role, because the user query is not machine understandable, so we need to translate to the ontology understandable language **SPARQL**. It has a graph-based structure and can be constructed by using triple patterns from the user query. It is difficult to recognize words as subjects or objects, because the user query is in unstructured format [2].

There are different kinds of triple pattern extraction algorithms, these implement the triple extraction based on the parser, where a *Parse-tree* [3] is constructed. Other algorithms use machine understandable and readable dictionary called *WordNet* [12], because it consumes considerably less amount of time.

A common strategy used for the implementation of keyword searches on RDF graph is the representation of a member resident graph using JENA or SESAME in which graph stores can be applied for the purpose of exploration of relationship chains, path algorithms are used [6]. The major problem with this is that it needs huge amount of memory for large or dense data graphs. This consumes more time for the exploration of relationship chains.

Two major issues have been encountered while extracting the sub-graph [4, 5]. First, if d is small then graph becomes disconnected and we cannot extract the links. Second, for the given set of keywords, some keywords will appear to be closer and some will be farther, due to this issue calculation of d is not exploited. These two issues can be rectified by suggesting a modified *reduction* strategy while keeping other two strategies same followed from the previous work [6].

Considering all vertices from the cluster and identifying the vertex which cannot be used to construct the resultant graph for the query are removed. If the keyword does not have a class name in common it implies that keywords are not too close. Then *reduction* strategy adds chaining of class nodes for processing, each cluster becomes larger based on the distance between the keywords or vertices. We have a new set of clusters, to find the suitable hook elements for joining and try to explore the required graph for the keyword query [5].

Triples are used in the representation of RDF data, where we have distinguished parts, namely, *subject, predicate, object* [2]. Every vertex corresponds to *subject, object* and an every edge corresponds to *Predicate*. These

triples are used for keeping and processing a large amount of RDF data. RDF is extended to the language RDFS for the purpose of expressing the general information about a data set [2].

2.1 Limitations of Existing Method

The methods discussed above have the following limitations which lead to incorrect results:

Exploration of possible paths: As the input RDF graph becomes larger, the exploration of possible paths implemented in reference [8] degrades in performance. An alternate strategy has been adopted to explore the possible paths starting with closely related vertices or edges to vertex mapped to every keyword and then increase the vertex set [6].

Exploration of type/subClassOf, subPropertyOf: Some semantic relationships available in RDF graph such as *subClassOf* or *subPropertyOf* are not explored while searching for the keywords. It leads to the construction of irrelevant/incomplete sub-graphs [4].

Also, the use of an Inverted index is difficult during the exploration of structural relationships. Indexing is used only to explore the set of vertices containing keywords [6].

We used the distance metric d to find out the graph which contains all elements related to the keyword query. This kind of structure called the d-neighborhood [4] is used for further processing. The Keyword may present far apart but might have a strong relationship, but d may produce a disconnected sub-graph as a resultant graph.

In reference [6], the approach uses summarization strategy where entity relations of well-defined types summarized. But this will pack all the entities of the same type in a single vertex and this leads to discharge of much information.

2.2 Overview and Problem Statement

Answering a keyword query in RDF/RDFS data is an important problem and it is resolved by using many different techniques. Present strategies used for this kind of search have various limitations and disadvantages. For the given keyword query in RDF/RDFS data, construct a sub-graph as output for each keyword along with some of their important properties and features. This can be constructed by using some of the elegant techniques in graph theory and its applications.

Construction of graph by using given keyword based queries on RDF data is represented by using graph $G\ (V, E)$. We studied and adopted the implementation techniques in reference [6]. Sub-graphs are constructed using closely related vertices, reduced and joined by using suitable vertices [5].

Many techniques have been proposed on keyword searches on relational data, tree-structured data, and graph structured data. In these methods, trees are identified by using an approximation of $Steiner - tree$ problem. In the proposed method, we label to map not only the vertices but also the edges, which include loops and cycles. To address this issue we adopt a different algorithmic approach called *Graph Exploration* [4]. Here we have an algorithm for the construction of answer graph to keyword queries on RDF data represented as graph $G\ (V, E)$.

We have to explore the associated concepts and neighboring concepts of each vertex or edge of the keyword to create a graph cluster. By using *reduction* strategy, we choose all vertices from the cluster and identify the vertex which cannot be used to construct the resultant graph for the query are removed. We have a new set of clusters in our hand, used to find suitable *hook or join* elements for connecting and then build the resultant graph for the keyword query [**?**].

2.3 Background and Related Work

In this section, a brief introduction about WordNet is given. For experimental verification, DBLP and YAGO data sets are considered, a brief description of which is given in subsequent section [7-11].

2.3.1 WordNet

WordNet is an enormous lexical database for English. It was originally developed at Princeton University [12]. As the name WordNet indicates, words are the units in WordNet, as the name indicates, though it contains idiomatic phrases, compounds, and phrasal verbs. The main purpose of WordNet was to make huge amounts of lexical knowledge available and also well-defined structure or platform that could benefit research in linguistics better than traditional dictionaries.

2.3.2 DBLP

DBLP provides an open bibliographic information on major computer science journals and proceedings. DBLP is a joint service of the University of Trier and LZI Scholss Dagstuhl. It started from a small collection of HTML files and became an organization hosting a database and logic programming bibliography site. It contains more than 3.66 million journal articles, conference papers and other publications on computer science. We can track all important journals on computer science, proceedings papers of many conferences. DBLP has been taken to stand for Digital Bibliography and Library Project and it is now simply named as The Data Base Systems Logic Programming Computer Science Bibliography [13].

2.3.3 YAGO (Yet Another Great Ontology)

YAGO is a joint project of the *MaxPlanck Institute for Informatics* and *Tcom Peris Tech* University and it is automatically extracted from Wikipedia, WordNet and GeoNames and it has a huge semantic knowledge base, derived from these resources.

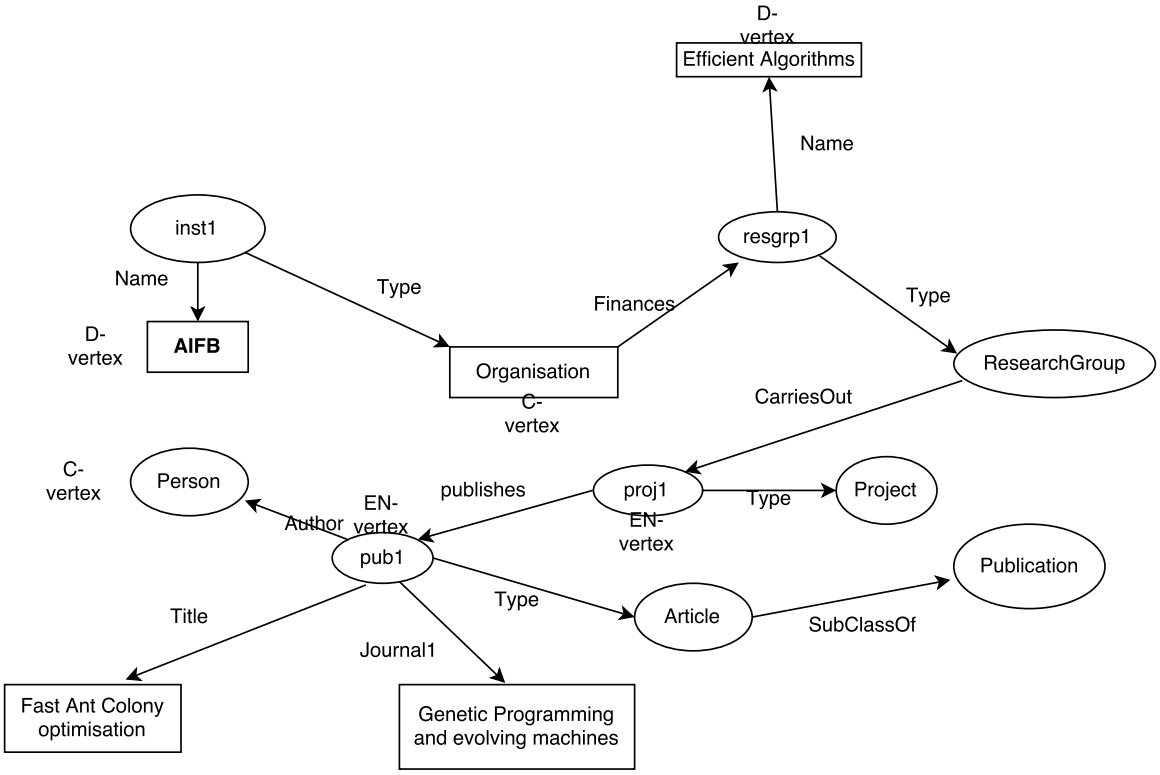

Figure 1: Example RDF Graph.

Currently, *YAGO* has knowledge of more than 16 million entities like person, organization, cities etc., and contains more than 120 million facts about these entities [14]. It was the first academic project to build *KB* from Wikipedia, closely followed by the DBpedia project.

YAGO2 is provided in TSV format i.e., Tab-Separated-Values (TSV) where file is a simple text format for storing data in a tabular structure. The TSV format allows user to easily import the facts into a database, or to handle the data programmatically. It is a part of the Linked Open Data cloud.

3 Preliminaries

In this section, we need some of the definitions considered for the further working on remaining modules. In this section we consider the RDF/RDFS data set containing triples formed by *subject-predicate-object* [3], now we have to query using extracted keywords.

An RDF/RDFS Graph G (V, E), where V is a set of finite vertices and E is a set of finite edges, we have a tuple *(V, E, L)* where V denotes the set of vertices which is disjoint in nature and it is defined below by using equation 1,

$$V = (Classes \cup Entities \cup Data) \tag{1}$$

E denotes the set of edges which connects the vertices v_1, v_2 where $(v_1, v_2) \in V$. The various kinds of edges we considered here are, *IE*-Edges (Inter-Entity Edges), *EA*-Edges (Entity-Attribute Edges) and *Class/Subclass* edges [4, 5]. In the above figure we have represented the different types of vertices and edges by using the example RDF graph fragment.

A labeling function l is used to label every edge in the graph G. The function l is given by the equation-2 [4, 5],

$$L = L(IE) \cup L(EA) \cup \{Class, SubClassOf\} \tag{2}$$

where *class* and *SubClassOf* are the two different predefined types of edge which captures the class membership of an entity and class hierarchy [6].

Some of the restrictions listed below on function l, [4-6],

- $l \in$ L(IE) if and only if $(v_1, v_2) \in EN$,
- $l \in$ L(EA) if and only if $v_1 \in EN$, $v_2 \in D$-vertex,
- $l =$ SubClass if and only if $(v_1, v_2) \in C$-vertex and
- $l =$ Class if and only if $v_1 \in EN$-vertex and $v_2 \in C$-vertex.

Note: We have one more type of vertex called as a *CR*-vertex, which is used to represent the inter-relationship with the C-vertex [4, 5, 6].

3.1 Connectivity

3.1.1 Relationship Connectivity - *RC*

Relationship Connectivity(RC) is a finite sequence of relation such as $\{RC_1, RC_2, \cdots, RC_n\}$ where the label in the graph associated with *IE/EA/type/SubClassOf*

edges. In Figure 1, *CarriesOut, publishes, title* is *Relationship connectivity* for the RDF graph fragment. The length of a relationship connectivity is equal to the number of relations in the connectivity. Two relationship connectivity RC_i and RC_j is lying across if

$$\left[CC(RC_i) \cap CC(RC_j)\right] \neq 0 \tag{3}$$

where CC be the Class Connectivity which is defined in the next sub section.

If two relationship connectivity $RC_1 = \{S_1, S_2, \cdots, S_n\}$ and $RC_2 = \{T_1, T_2, \cdots, T_n\}$ are known as similar connectivity, if $\forall$ i, $(1 \leq i \leq n)$, $S_i = T_i$ or S_i is *SubClassOf* T_i or vice versa [6].

3.1.2 Class Connectivity - *CC*

Class Connectivity (CC) is a finite sequence of corresponding relationship connectivity $\{CC_1, CC_2 \cdots CC_n\}$ where,

- CC_i $(1 \leq i \leq n)$ is a C-vertex,
- CC_n is a D-vertex if RC_n is a IE-edge and
- CC_n is a C-vertex if RC_n is a EA-edge.

3.2 System Design Architecture

In this section, RDF data is given as input and analyzed by the analysis phase that removes RDF tags and unwanted information in the RDF data. Later using the ontology we are extracting the triples in the form of graphs. Figure 2 gives detailed system design and its overview.

After analysis phase, we have a set of *token* stream in our hand. These token streams are given as an input to the ontology. The next major and important phase is the extraction of graph patterns from the domain ontology.

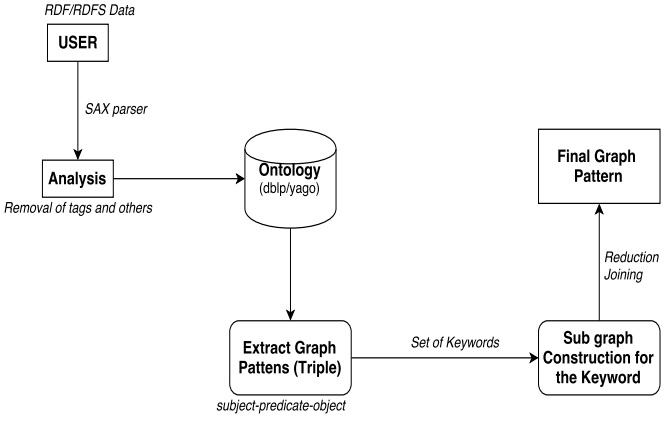

Figure 2: An Overview of Proposed System Design

In the analysis phase, the user gives RDF data and then analysed, that is the following tasks carried out in this phase. The following tasks are carried out during this phase.

- Spellcheck,
- White space removal,
- Lemmatization,
- Parts Of Speech Tagging ($POST$),
- Remove stopwords other than preposition in ontology and
- Select *Noun*, *AdjectiveNoun* as query terms.

3.3 Graph Pattern Extraction

We have the following approach to extract the graph pattern from the domain ontology [3].

3.3.1 Triple Pattern Extraction

Every RDF data is completely described using graph $G\ (V, E)$ consisting of a set of finite vertices and edges. Every vertex denotes either a subject or an object and every edge denotes a predicate described by using an RDF triple $subject - predicate - object$. Triple extraction has following three different steps or parts [3].

- Concept Identification,
- Subject Group Recognition and
- Object Group Recognition.

Concept Identification: Current phase used to find whether the given keyword has a noun or not with the help of $WordNet$ [12]. Later it defines the *concept* for each noun in the given keyword with the help of domain ontology. Suppose, if the word is similar to class type, then the word represented as **class**. If the word is similar to data type then the word represented as **predicate** and finally, if the word is not suitable with the class or data type or object type then the word represented as **constraint**. These three different concepts explored in this phase [3]. Here, we used the *Edit-distance* and the $WordNet$ to calculate the similarity between the words and *concepts* obtained from the domain ontology.

$$Sim(W_1, W_2) = \sum_{i=1}^{4} \beta_i \prod_{j=1}^{i} Sim_j(W_1, W_2) \tag{4}$$

where, β is an adjustable parameter; $Sim(W_1, W_2)$ be semantic similarity. We have taken the threshold value for semantic similarity is 0.5.

By using this, every *Concept* is divided into original *Concept atoms* and every original *Concept atom* again divided into the following [3]: Primary, Relation, Symbol Part and Other.

Then figure out the complete semantic similarity of *concepts* is equal to the weighted average of above mentioned four different parts of *Concept atom*.

Subject Group Recognition: In this phase, we need to extract the subject and predicates. All triples

(*subject*, *predicate*, *object*) extracted using the above-mentioned phases., And these used to build the $SPARQL$ query for browsing an ontology [3].

Object Group Recognition: This phase describes the extraction of objects corresponding to the set of keywords from the input. Predicate, which denotes the relationship between subject and object, obtained using constraints from the previous phase [3].

Algorithm 1 Triple Pattern Extraction

```
Input: Query Sentence.
Output: Subject, Predicate and Object.
Function: Triple Extraction(Query Sentence).
List ← Noun Detection                                  ▷ using WordNet
for (w ∈ List) do
    conc ← concept(w)                                  ▷ concepts
    concept_list.append(conc)                          ▷ list of concepts
end for
Object ← Object_extract(List,conceptList)              ▷ extraction of objects
Subject ← SP_extract(List,conceptList)                 ▷ extraction of subject and predicate
Result ← (Object ∪ Subject)                            ▷ collection of all triples as a result
return Result.
```

Algorithm-1 detects the noun from the dictionary $WordNet$ and generates subject, predicate and object and a value for unstructured query [3].

Algorithm-1 explains the extraction of triplets from the input sentence and recognizes the nouns from the the dictionary *WordNet*, Objects, the Relations from the Concept list. This will extract the Subject Predicate in one function and Object and its corresponding values in another function. The results of these two will be combined in the in this section of the Algorithm.

4 Algorithm Description

4.1 Problem Statement

The given query Q consists of list of keywords like $\{k_1, k_2.....k_n\}$, and these list of keywords constructs the graph G' to the query Q with minimum possible sub graphs like G', G''..... such that,

- each keyword is contained at least one vertex/edge in a graph G.
- the graph G contains only keyword vertex and class vertex which are connected Inter-Entity and Entity-Attribute edges.
- Graph G' is minimum that means no sub-graph of G' can be *graph(G')* to Q.

ROLE/GOAL: Construction of graph G' for the keyword query Q in RDF data [2]. The construction doesn't use the distance neighborhood method, and it will extract both the trees and cycles in it.

4.2 Illustration of Adopted Approach

Keyword Mapping: Choose each keyword k_i in the vertex set and map it using String Search strategy to one of the vertices in the graph. Taking one mapped vertex corresponding to each keyword, the list of mapped keywords (KS) constructed and it acts as a response to the algorithm. The graph G' is constructed in three different steps as depicted in [6].

- Subgraph-Construction
- Subgraph-Reduction
- Subgraph-Joining

Algorithm 2 Construct_Resultant_Graph

```
Input: K[n]
Output: Sub_Graph (G' [ ])                             ▷ list of sub graphs
G' [ ] ← ϕ
for (keyword k ∈ K) do
    G' ← G' ∪ {Sub_Graph(k,G)}
    SGC ← SubGraph_Construction                        ▷ construction of all sub graphs
end for
Resultant_Graph (RG) = ϕ                               ▷ set of resultant graphs
for (graph G' ∈ SGC) do
    R ← Graph_Reduce (G') ▷ reduction of unnecessary vertices and edges in the sub graph
    RG ← RG ∪ {Graph_Join (R)}
end for
Return RG
Sub_Graph(k, G')
g' ← ϕ                                                 ▷ set of sub graphs
for (vertex v ∈ vertex-set(k, G)) do
    g' ← g' ∪ {construct-subgraph (v, G)}
end for
```

Algorithm-2 gives the detailed description to interpret resultant graph as output. We have a set of keywords in our hand as a response to our method. For each keyword, $k_i \in K[n]$, construct a sub-graph along with its adjacent vertices and their edges. Our approach has two different methods viz, Graph_Reduce method, used to remove all unnecessary vertices and its edges in the sub-graph and Graph_Join method, used to merge all sub-graph constructed after the Graph_Reduce method [6].

Subgraph-Construction: From the vertex set, v_i is taken and the type of vertex identified. If the vertex v_i is similar to C-vertex then we are adding all the C-vertices and CR-vertices along with their edges. If the vertex v_i is similar to D-vertex then we are considering C-vertex for the class to which entity of this D-vertex is connected and the CR-vertex for that particular C-vertex along with its edges added. CR-vertices are considered only when they are related to D-vertex through their EN- vertex [4, 5].

And finally if the vertex v_i is similar to IE-edge then we are adding the C-vertex and EA-edges. As stated above the sub-graph construction step carried out for all the vertices in the keyword set KS and outputs the sub-graph corresponding to every keyword. This sub-graph acts as a response to the reduction step [5]. Formally, sub-graph formed by using C-vertices and its associated edges in the original graph G.

Homomorphic Structures: Homomorphic Structures defined as similar kind of graph structures obtained during the Subgraph-construction step. These homomorphic or similar graphs merged based on the type of vertex (D- vertex) and edge type (EA- edge). If the two graph structures g' and g'' are different for different D- vertex and EA- edge only and if C-vertex(g') and C-vertex(g'') are same i.e., C, CR- vertices are same, then these graphs are homomorphic in nature [6].

Algorithm 3 SubGraph_Construction(sg[n])

```
Input: K[n]                                              ▷ set of keywords
Output: sg[n]                                            ▷ list of sub graphs constructed
sg[n] ← ϕ                                                ▷ set of sub graphs is empty
if (K ∈ C − vertex) then                                 ▷ Class vertex
    sg = sg ∪ {CR- Vertex (C- Vertex)}                   ▷ Add all C-vertices
else
    if (K ∈ D − vertex) then                             ▷ Data vertex
        sg = sg ∪ {type(D- Vertex)}                      ▷ type of D-vertex
        sg = sg ∪ {CR- vertex(C- vertex)}                ▷ add Data vertex with their Class vertices
    else
        if (K ∈ EA- edge) then                           ▷ entity attribute edge
            sg = sg ∪ {type(EN- Vertex(EA- edge))}       ▷ add all entity attribute edge with their entity vertex
            sg = sg ∪ dummy − vertex
        else
            if (K ∈ IE- edge) then                       ▷ Inter entity edge
                sg = sg ∪ {type(EN- vertex(IE- vertex))} ▷ add all inter entity edges with their C vertex and CR vertex
                sg = sg ∪ {CR- vertex(C- vertex)}
            end if
        end if
    end if
end if
return sg[n]                                             ▷ returns list of sub graphs
```

Algorithm-3 returns the list of sub-graph from the set of keywords. For each keyword, k_i in the set detects the type of vertex whether the keyword k_i belongs to C-vertex. If yes, then add it to the list sg. The Later algorithm detects whether it belongs to D-vertex, if the keyword k_i belongs to D-vertex then add it to the list sg along with its CR-vertex.

In the second part, algorithm-5 detects the type of edge such as EA (Entity-Attribute)-edge. If it is an EA-edge then add it to the list sg with their vertices (EN-vertices) incident on EA-edge. If the edge belongs to IE (Inter-Entity)-edge then add IE-edge incident on the EN-vertex and along with their C-vertex or CR-vertex. Finally, the algorithm returns a list of sub-graphs $sg[n]$ as advice for the next step.

Subgraph-Reduction:

In this step, algorithm remove unwanted vertices for the constructed sub-graph from the original graph G. We have pairwise sub-graph, the similar vertices identified. This computation by intersection of the vertex list pair. If two vertices are similar, then any one of the following property should obey,

- they are the same vertex in the G.
- the vertex $v_i \in V$ related to some subclass relationship.
- there exist a connection between the two vertices, through some intermediate vertices (CR-vertex).

Second property is used only if the similar vertex is empty, it indicates that the vertices corresponding to the keywords are not close to each other (neighbors), and we extend the cluster by C-vertex chain with the help of third property. The union of all similar vertices is extracted by considering all the pairs, that gives the list of vertices which is used for the further process. If there is more than one chain count, then the chain with the minimum number of C-vertices are considered [6].

The pruned vertices along with their incident edges are removed from the Sub-graph Construction. The updated list of sub-graph will be obtained and this list will be given as an input to the joining step [6].

Algorithm 4 Graph_Reduce(sg[n])

```
Input: sg[n]                                             ▷ list of sub graphs
Output: sg[n]                                            ▷ updated sub graphs list
C_list ← union of all C- vertices in the sg              ▷ list of all class vertices
for (pairwise {g,h} ∈ sg) do
    if (g = h OR h = g) then                             ▷ g and h are two different sub graphs
        Clist = Intersection of (g,h)                    ▷ intersection of all C-vertices
    end if
    if (Clist == isempty) then
        for (pair of C- vertex ∈ {g,h}) do
            update RC based on the shortest relation connectivity chain
            update CC based on the class connectivity of RC
        end for
        Clist ← union of all Clist
        ReduceClist ← (C_list - Clist)
        ReduceClist(sg[n])
    end if
end for
return (sg[n], Clist)                                   ▷ returns list of all C-vertices and sub graphs after reduction
```

Algorithm-4 returns the sub-graph after removing the unnecessary vertices and edges. In the first stage of our algorithm, we find out the complete C-vertex set by using the operation union of all similar vertices. For each pairwise sub-graph, the algorithm finds out the C-vertices which give the $Clist$.

By using complete C-vertex set and *Clist* set, figure out the vertices which will be reduced from the sub-graph. These vertices along with associated edges removed from the corresponding sub-graph. The new list of sub-graph returned from the algorithm, which is considered as an input for the next Joining or merging step [5, 6].

Sub-graph Joining

In this step, the property of Similar Vertices is used which is defined earlier in the previous step. Finding the sub-graph for C/CR-vertices of minimum *cardinality* are chosen. Once you find the vertices to be joined, then the corresponding clusters are glued or joined together. Duplicate vertices are removed and a new joint sub graph component is used for further hooking operation [4, 5].

The above process is carried out until the vertex set becomes null. Then find the loosely hanging vertices in the sub-graph G' if any, and try to remove these vertices from G' and keeps the remaining G' with vertices and edges which form the resultant sub-graph G'.

Algorithm 5 Graph_Join (sg[n])

```
1: procedure GRAPH_JOIN(sg[n])
2:     Input: sg[n]                          ▷ list of sub graphs after reduction
3:     Output: sg[n]                         ▷ updated sub graphs after joining
4:     graphjoin ← ReduceClist(sg[n])
5:     while (cardinality(∃ i such that sg[i] is unprocessed) do
6:         g ← ReduceClist with less vertex cardinality
7:         h ← ReduceClist with min relationship connectivity chain
8:         ReduceClist ← merge(g, h) ▷ update the ReduceClist after merging two sub graphs
9:         graphjoin ← merge(g, h)
10:        graphjoin ← (graphjoin ∪ ReduceClist)
11:    end while
12:    remove all hanging vertices of graphjoin if it is present
13:    return sg[n]                          ▷ updated list of sub graphs after joining
14: end procedure
```

Algorithm-5 considers the pairwise sub-graphs, choose a C-vertex or CR-vertex with less *cardinality* and those sub-graphs are joined or merged together. The duplicate vertices are removed from the sub-graph. This updated sub-graph and the new sub-graph from the list are chosen and again the same procedure is applied followed for all sub-graphs in the input sub-graph list.

Finally we came up with the sub-graph with some of the vertices loosely hanging. So these vertices should be removed. If a vertex is loosely hanging, then it is not a keyword and it has only one edge associated with it.

4.3 Indexing

The following Indices are maintained in our project for the optimal construction of sub-graphs and it will improve the performance of the each and every step of the algorithm.

- *Class/type* ← For each EN- vertex the associated *Class/type* is maintained as a list. It is used to find out the Class-Connectivity in the reduction step.
- *SubClassOf* ← For each C- vertex is a *SubClassOf* C- vertex is maintained as a list.
- *Relationship* ← For each EN- vertex there is an IE-edge is maintained in a bidirectional way.

4.4 Ranking

Since we have constructed multiple sub-graphs through our approach and we need to score each graph G' precisely with their corresponding meaning and identify the top one. We are using the approach of structural compactness as one of the major criteria to rank/score and also we used two more approaches viz relationship relevance and vertex type relevance in our ranking module [6]. We used some other scoring/ranking strategies which are used in our work and will be explained below in detail [9].

The basic idea of the ranking method existing in [7] is to first assign each r-radius graph, a score using standard IR-ranking formula or its variants and then combine the individual score using a score aggregation function (SUM) to obtain the final score. It cannot evaluate the structural relevancy among input keywords, which captures the phrase-based relevancy between input keywords.

Compactness Relevance:

Compact answers should be preferred that means closer connections between the mapped vertices are preferred rather than farther connections. The keywords are taken into consideration which has a closer connection between the mapped keywords or vertices. This leads to the precise meaning called structural compactness with respect to the graphs. This has been determined in two different ways: First, the **mapped vertex compactness** and the second, **vertex relevancy** which it is mapped [6, 7].

Here, if the length of C-vertex chain between the mapped vertices is greater, then the compactness between them is lesser [6].

The structural compactness for the graph G' is given below:

$$SC(k_i, k_j|G') = \left[\frac{1}{(C\_len + 1)^2}\right] \tag{5}$$

if vertices are same i.e., $k_i = k_j$.

$$SC(k_i, k_j|G') = \left[\frac{1}{2(C\_len + 1)^2}\right] \tag{6}$$

if vertices are different i.e., $k_i \neq k_j$.

Where, SC be the Structural Compactness and C_len is number C-vertices having the minimum length + 1.

Term Relevance: For each keyword in the set, a term relevance matching score will be calculated by using standard IR strategy and we are combining the compactness relevance too.

$$RANK(k_i, k_j|G') = \Big[SC(k_i, k_j|G') * \Big((TR(k_i|G') + (TR(k_j|G')\Big)\Big] \quad (7)$$

$$RR(G') = \left[\frac{|C\_vertices| \in G'}{|C\_vertices|}\right] - \left[\frac{|C\_vertices|addedinG'}{|C\_vertices|}\right] \quad (8)$$

Equation 7 is used for the above-mentioned computation of term relevance [5].

Here, $RANK$ be the rank value for the answer graph constructed from algorithm and TR be the term relevance for the keyword in the resultant graph G'.

Vertex Relevance: The vertex/edge has prominent role for ranking strategy. In the resultant graph G' having the *Class*-vertex should rank higher for the specified keyword. C-vertices or IE-edges will have the greater value followed by EA-edges followed by D-vertices. The vertex relevance(VR) value for all the mapped vertices are computed here [5].

Relationship Relevance: In this method, we find the missing relationship or interconnected vertices. This information is necessary to find the neighbors of C-vertices which will be used for the construction of the required graph G' [5]. It is computed using the equation 8.

The complete rank value is calculated by using equation 9.

$$RANK(G') = \sum_{1 \le i \le j \le n} Rank(k_i, k_j|G') + VR(G') + RR(G') \quad (9)$$

5 Design and Implementation

JENA [15] or SESAME [16] which are RDF graph stores can be used to find the path and its relationship between keyword queries. The major drawback in this technique is that it is not scalable for large graphs. It requires high memory and takes a considerable amount of time to explore the relationship between keywords. To overcome this, we used existing RDF data store technology Allegro RDF suite [6]. This technology used to find the triples *subject-predicate-object* for the given keyword query. But the above mentioned RDF suite also consume more time to extract triples. Instead, we demonstrated the exploration of triples for a given RDF query in our thesis. In this section we discussed about SAX parser and detailed implementation of our algorithm with different data sets like *DBLP* and *YAGO*.

5.1 SAX Parser

SAX (Simple API for XML) is an event-driven parser for XML documents. SAX provides a mechanism for reading data from an XML document that is an alternative to that provided by the *Document Object Model*(DOM), where DOM operate on the document as a whole, SAX parser operates on each piece of the XML document sequentially.

SAX is a streaming interface for XML, which means that applications using SAX receive event notifications about the XML document being processed an element, and attribute, at a time in a sequential order starting at the top of the document and ending with the closing of ROOT element.

We should use a SAX parser because,

- we can process the XML document in a linear fashion.
- we are processing a very large XML document where DOM tree consumes too much memory.
- Data is available as soon as seen by the parser, so SAX works well for an XML document that arrives over a stream.

6 Illustration of the Algorithm With Example Queries

We used the *DBLP* [13] dataset for illustration. It contains major computer science journals and proceedings like *article, inproceedings, proceedings, book, incollection, phdthesis, masterthesis and www.* Each entity, in turn, has some attributes such as *author, editor, title, booktitle, pages, year, journal, publisher, volume* and so on. Before subgraph construction, entire DBLP data set needs to be parsed using SAX Parser. SAX Parser parses the data set and stores it in memory as a graph structure. When each keyword given as input to our method, the entire adjacency list is parsed, vertices with their relationships are retrieved.

For our convenience we considered *inproceedings* and *proceedings* entities from the *DBLP* data set. Inturn in *inproceedings* we are extracting the attributes such as *author, cite, conference, title, year* and in *proceedings*, *confname* and *confdetail* are extracted [13].

The complete algorithmic framework for keyword search on RDF data is implemented using Triple-extraction algorithm. The strategy has been implemented using Java. For experimental verification, *DBLP* and *YAGO2* data set has been used. The system has been tested on these data sets for different keyword queries.

Each keyword is mapped to vertices and edges of different types or classes. In our adopted approach, keyword gets mapped to different vertices and edges of the RDF graph data. The constructed sub-graph in the sub-graph construction algorithm may have a similar type of keyword vertices like D-vertices, the relationship, and the subclass. For this kind of sub-structures, the complete

CR-vertices are extracted in memory. These vertices are used in the sub-graph reduction and sub-graph joining process in the later stages.

Some keywords do not exploit the *SubClassOf* property, where some keywords mapped directly to an edge of an RDF graph data. This results in an empty output graph using our approach. Thus the different class of keyword queries can be recognized.

We have listed some of the keyword queries for the *DBLP* and *YAGO*2 data set in Table 1. The following keyword queries have been tested in our system and answer graph has been extracted. Each keyword query consumes a finite amount of time to extract answer graph. *DBLP* data set has been parsed using SAX Parser to store entire RDF data as a graph in the memory as a pre-processing step.

Table 1: List of queries chosen from DBLP and YAGO2.

Sl No.	Query (DBLP and YAGO2)
1	C.Thomas Wilkas, Frank manola, Information, 1993
2	Rajesh Parekh, Vasant Honavar, Database, 1999
3	Stan Liao, Kurt Keutzer, IR, 1998
4	Edmund M A Ronald, Dominique, Data Mining, 1999
5	Steven R Newcomb, XML, relational, 1994
6	Arvind Hulgeri, S. Sudarshan, Keyword Search, 2009
7	Alan W Biermann, machine Learning, 2001
8	Dominique Bolignano, Jonathan K Millen, Cryptographic
9	Eli Upfal, Gopal Pandurangan, 2001
10	semantic, mathematical model
11	Hash Functions, 1998
12	probability, 1994, ac, nagomi
13	Srinivasan, 2003, ontology
14	Frank manola
15	Sreenivasa kumar
16	Sreenivasa kumar, vinu, bhaskar
17	created
18	Alfred_Hitchcock
19	Apple_Inc
20	influences
21	Abraham_Lincoln
22	Murray_Rothbard

Table 2 gives the detailed information about the time taken to construct answer graph for the different data sets like *DBLP* and *YAGO2*.

Table 2: Time taken by the DBLP and YAGO2 data-sets.

Data set	Size (GB/MB)	No. of vertices	Parsing time (sec)
DBLP	2 GB	3332013	5522 (95 min 13 sec)
YAGO2	40.8 MB	834750	20

6.1 Example-1

Query: Frank manola

First our algorithm parses the *DBLP* data-set and stores in the memory. The keyword **Frank manola** searched by using string matching algorithm and graph is traversed, vertices adjacent to the keyword vertex **Frank manola** along with it's relationships are extracted. For example, our algorithm extracts the relationships such as *Title, year, publ, cite, author* and their instances for the keyword **Frank manloa**. The constructed sub-graph is stored in the adjacency list for the further processing.

The constructed sub-graph is taken as input to our reduction step and unwanted vertices along with their relationships are removed. Because if the given keyword query belongs to the entity *inproceedings* and *proceedings* only extracted and keyword which has relationship for the keyword query are extracted, remaining vertices and their relationships are removed from the sub-graph constructed. This is known as sub-graph reduction step.

If the keyword is available in more than one sub-graph with different relationships or types are considered based on the number of vertices available in the sub-graph reduction step. The complete keyword vertex with their relationship is merged with the other sub-graph and declared as answer graph for the keyword query. This can be visualized by using adjacency list data structure. The answer graph for the keyword query **Frank manola** is shown in Figure 3.

6.2 Example-2

Query: vasant, 1999

The keyword **vasant, 1999** searched by using string matching algorithm and graph is traversed, vertices adjacent to the keyword vertex **vasant, 1999** along with it's relationships are extracted. For example, our algorithm extracts the relationships such as *Title, year, publ, cite, author* and their instances for the keyword **vasant, 1999**. The constructed sub-graph is stored in the adjacency list for the further processing.

The constructed sub-graph is taken as input to our reduction step and unwanted vertices along with their relationships are removed. Because if the given keyword query belongs to the entity *inproceedings* and *proceedings* only extracted and keyword which has relationship for the keyword query are extracted, remaining vertices and their relationships are removed from the sub-graph constructed. This is known as sub-graph reduction step.

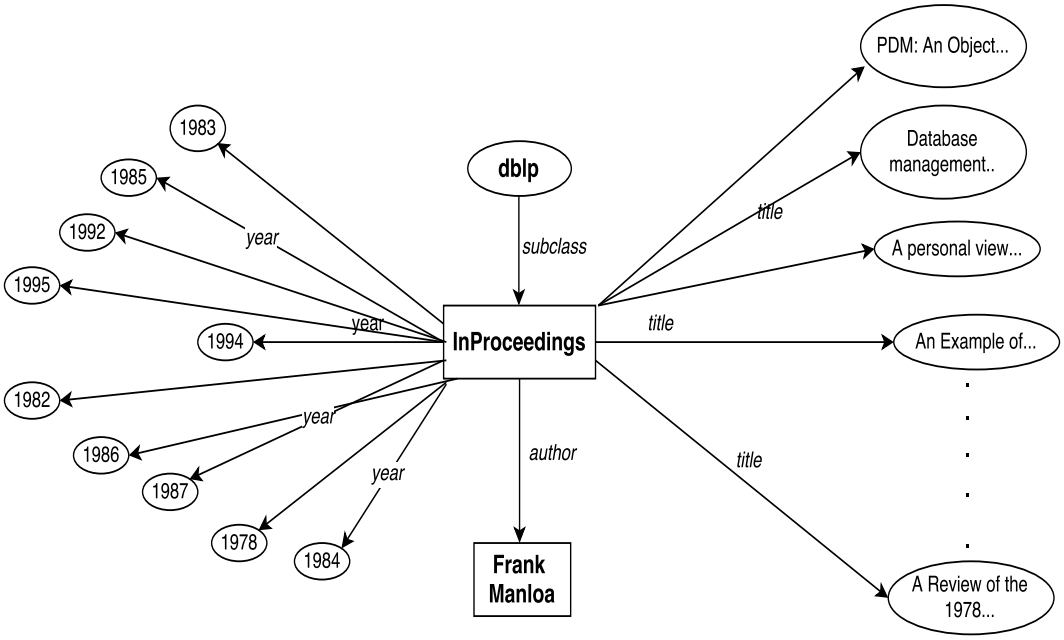

Figure 3: Answer graph for the keyword *Frank Manola*

If the keyword is available in more than one sub-graph with different relationships or types are considered based on the number of vertices available in the sub-graph reduction step. The complete keyword vertex with their relationship is merged with the other sub-graph and declared as answer graph for the keyword query. This can be visualized by using adjacency list data structure. The answer graph for the keyword query **vasant, 1999** is shown in Figure 4.

6.3 Example-3

Query: Sreenivasa kumar

The keyword **Sreenivasa kumar** searched by using string matching algorithm and traverse the entire graph from top to bottom and extract the vertices adjacent to the keyword vertex **Sreenivasa kumar** with their relationships. For example, our algorithm extracts the relationships such as *Title, year, publ, cite, author* and their instances for the keyword **Sreenivasa kumar**. The constructed sub-graph is stored in the adjacency list for the further process.

The constructed sub-graph is taken as input to our reduction step and unwanted vertices along with their relationships are removed. Because if the given keyword query belongs to the entity *inproceedings* and *proceedings* only extracted and keyword which has relationship for the keyword query are extracted, remaining vertices and their relationships are removed from the sub-graph constructed. This is known as sub-graph reduction step.

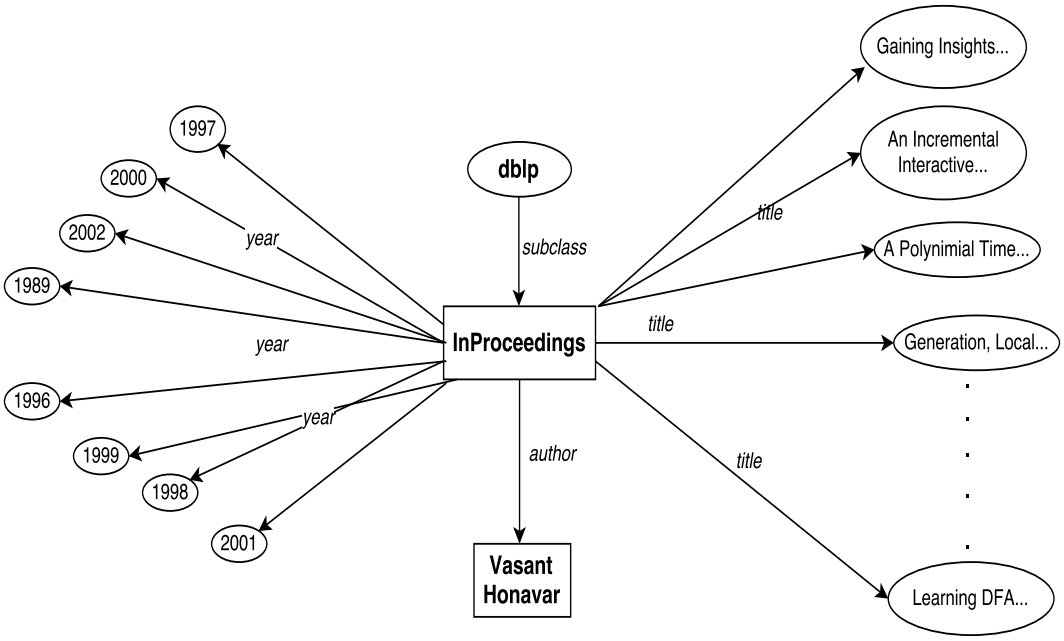

Figure 4: Answer graph for the keyword *vasant, 1999*

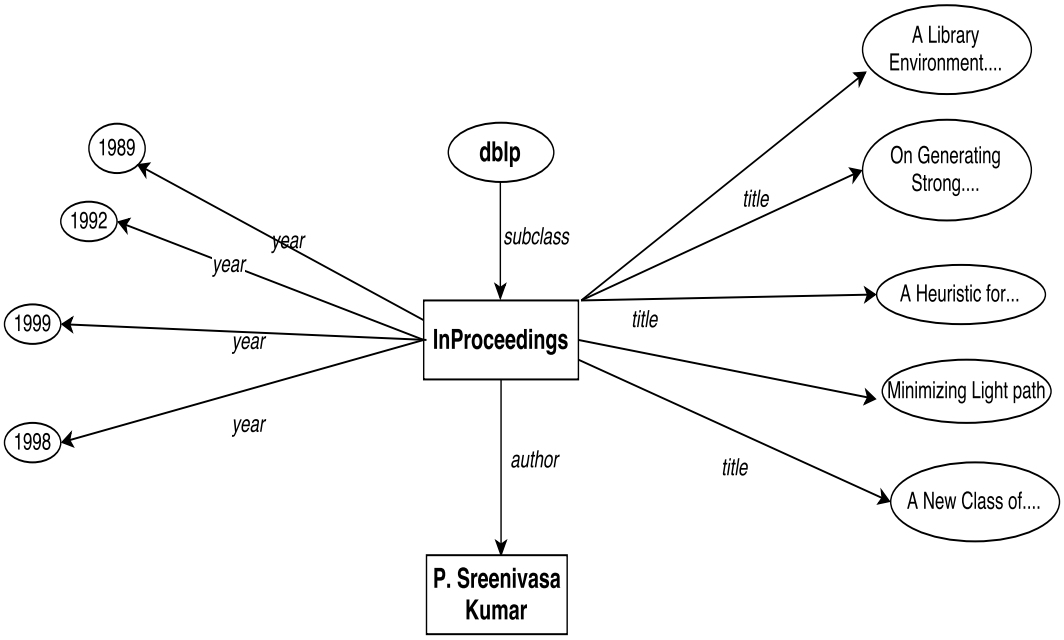

Figure 5: Answer graph for the keyword *Sreenivasa kumar*

If the keyword is available in more than one sub-graph with different relationships or types are considered based on the number of vertices available in the sub-graph reduction step. The complete keyword vertex with their relationship is merged with the other sub-graph and declared as answer graph for the keyword query. This can be visualized by using adjacency list data structure. The answer graph for the keyword query **Sreenivasa kumar** is shown in Figure 5.

7 Conclusion

A novel algorithm has been proposed to construct an output/answer graph G' by using the keywords extracted from the domain ontology and knowledge repository represented as RDF/RDFS graphs. We aim to find the structural information with respect to the set of extracting keywords from the domain ontology and RDF/RDFS data set.

The same approach has been extended to semantic search by using domain specific ontology. After extraction of the triple from the domain ontology, we have constructed the sub-graph as a resultant answer for a given keyword query. We have also exploited other useful information called relationship *(type/SubClassOf)* between the keywords or elements. Relevant scoring mechanism has been used to rank the extracted sub-graph and top-k sub-graphs scored.

Through our approach, it is very difficult to explore a precise graph structure for very large/dense graph and it can be extended to the same for the *LOD (Linked-Open-Data)* graph. Every constructed resulting graph is searched in the very big graph database of their specific domain.

References

[1] Ricardo Baeze-Yates and Berthier Ribeiro-Neto, "Modern Information Retrieval", Second Edition - 1999 by the ACM press.

[2] Pascal Hitzler, Markus Krotzsch and Sebastian Rudolph, "Foundations of Semantic Web Technologies", editor - CRC Press, edition - 2009.

[3] Zin Thu Thu Myint and Kay Khaing Win, "Triple pattern Extraction for Accessing Data on Ontology" in Proceedings of International Journal of Future Computer and Communication, China, vol.3, no. 1, pp. 40-44, 2014.

[4] Parthasarathy, P Sreenivasa Kumar and Dominic Damien, "Algorithm for answer graph construction for keyword queries on RDF data", in IW3C2, Ph.D. Symposium, India, vol. 978 no. 1 pp. 366-370, 2011.

[5] Parthasarathy, P Sreenivasa Kumar and Dominic Damien, "Ranked answer graph construction for Keyword Queries on RDF Graphs without Distance neighborhood Restriction", in IW3C2, Ph.D Symposium, India, vol. 978 no. 1 pp. 361-365, 2011.

[6] Parthasarathy, P Sreenivasa Kumar and Dominic Damien, "Semantic answer graphs for keyword queries on RDF/RDFS graphs", in SWJ, vol.978, no. 03, pp. 407-418, 2009.

[7] Thanh Tran, Haofen Wang, Sebastian Rudolph and Philipp Cimiano, "Top-k Exploration of Query Candidates for Efficient Keyword Search on Graph-Shaped (RDF) Data" in IEEE International Conference on Data Engineering, USA March-April, vol.01, no. 01, pp. 405-416, 2009.

[8] Thanh Tran, Philipp Cimiano, Sebastian Rudolph and Rudi Studer, "Ontology-based Interpretation of Keywords for Semantic Search" in ISWC 2007 + ASWC Busan, Korea, vol. 4325, no. 1, pp. 523-536, November 2007 Proceedings.

[9] Guoliang Li, Beng Chin Ooi, Jianhua Feng, Jianyong Wang and Lizhu Zhou, "EASE: An Effective 3-in-1 Keyword Search Method for Unstructured, Semi-structured and Structured data", in SIGMOD, vol. 978, no. 1, pp. 903-914, 2008.

[10] H. He, H.Wang, J.Yang and P.S.Y u.Blinks: "BLINKS:Ranked keyword searches on graphs", in SIGMOD Conference, vol. 978, no. 1, pp. 305-316, 2007.

[11] V. Karcholia, S.Pandit, S.Rudolph and R.Studer "Ontology based interpretation of keywords for semantic search", in ISWC/ASWC, vol. 4825, no. 1, pp. 523-526, 2007.

[12] WordNet 2.1 reference manual, "http://wordnet.princeton.edu/man/", "cognitive science laboratory, Princeton University", vol. 54, 2005.

[13] "http://dblp.uni-trier.de/".

[14] Thomas Rebele, Fabian Suchanek, Johannes Hoffart, Joanna Biega, Erdal Kuzey and Gerhard Weikum, "YAGO: A Multilingual Knowledge Base from Wikipedia, Wordnet and Geonames" in ISWC, vol. 9982, no. 2, pp. 177-185, 2016.

[15] "https://jena.apache.org/documentation/javadoc/jena/org/apache/jena/graph/Triple.html".

[16] "https://franz.com/agraph/allegrograph/".