Indexing Structure for Discovering Relationships in RDF Graph Recursively Applying Tree Transformation

Stanislav Bartoň
Faculty of Informatics
Masaryk university
Brno, Czech republic
xbarton@fi.muni.cz

ABSTRACT

Discovering the complex relationships between entities is one way of benefitting from the Semantic Web. This paper discusses new approaches to implementing ρ-operators into RDF querying engines which will enable discovering such relationships viable. The cornerstone of such implementation is creating an index which describes the original RDF graph. The index is created by recursive application of a transformation of graph to forest of trees. At each step, the RDF graph is transformed into forest of trees and then to each tree its extended signature is created. The signatures are accompanied by the additional information about transformed problematic nodes breaking the tree structure. The components described by the signatures are assumed as a single node in the following step. The transitions between the signatures represent edges.

1. INTRODUCTION

One form of retrieving information from the Semantic Web is to search for relations among entities. The simple relations such are the *is-a* or *is-part-of* relations can be found easily. For example using RQL [4] one can find direct relationship among entities. This means that we are able to retrieve all the descending classes of one class, even on a different level. For example the user can ask for all instances of a class 'artist' as it is shown in Figure 1. The answer to such query would be all instances of both its subclasses in the knowledge base, all painters and sculptors. But in the Semantic Web there can be observed more complex relationships among entities [9] than those simple ones.

Such complex relationship can be represented by a path between two entities consisting of other entities and their properties. To discover such complex relationships ρ-operators [1] have been developed. In this paper, the complex relationships are discussed and are referred to as Semantic Associations [9]. The ρ-operators are precisely the tools for discovering such Semantic Associations. This class contains ρ *path*, ρ *connect* and ρ *iso* operators.

ρ **path** - This operator returns all paths between two entities in the graph. An example of such relation can be seen in Figure 1 between resources `&r1` and `&r4`. Such association represents an information that a painter called Pablo Picasso had painted a painting which is exhibited in Reina Sofia Museum.

ρ **connect** - This one returns all pairs of paths that intersect in one common node and which initial nodes are the two entities for which we are searching the association. An example of such ρ connection between the resources `&r6` and `&r9` in the Figure 1 is represented by the paths from resource `&r6` to `&r8` and from resource `&r9` to `&r8` where the resource `&r8` is the common node. This association represents a fact that two artists had their artifacts (in one case it was a painting and in the other a sculpture) exhibited in the same museum.

ρ **iso** - This operator implies a similarity of nodes and edges along two paths. The similarity of the the paths, one starting in resource `&r1` and ending in `&r4` and the second one going from `&r6` to `&r8`. The two paths are ρ isomorphic since they both represent an artist creating artifact, that is exhibited in a museum.

The possible usage of searching such complex associations can be found in the field of national security. For example the system could be used on airports to help to identify suspicious passengers by looking for available connections between them.

In this paper we mainly focus on the former two operators which are the ρ path and ρ connect. We introduce a design of a indexing structure for the RDF graph that will make the discovery of the relationships described by these ρ operators effective.

Section 3 discusses the related work to the topic of indexing RDF graphs. Section 2 contains a brief introduction into the RDF and the RDF Schema. In Section 4 we present our contribution to the issue by introducing the transformation of the RDF graph into forest of trees and after-wards the application of tree signatures to those trees. Section 5

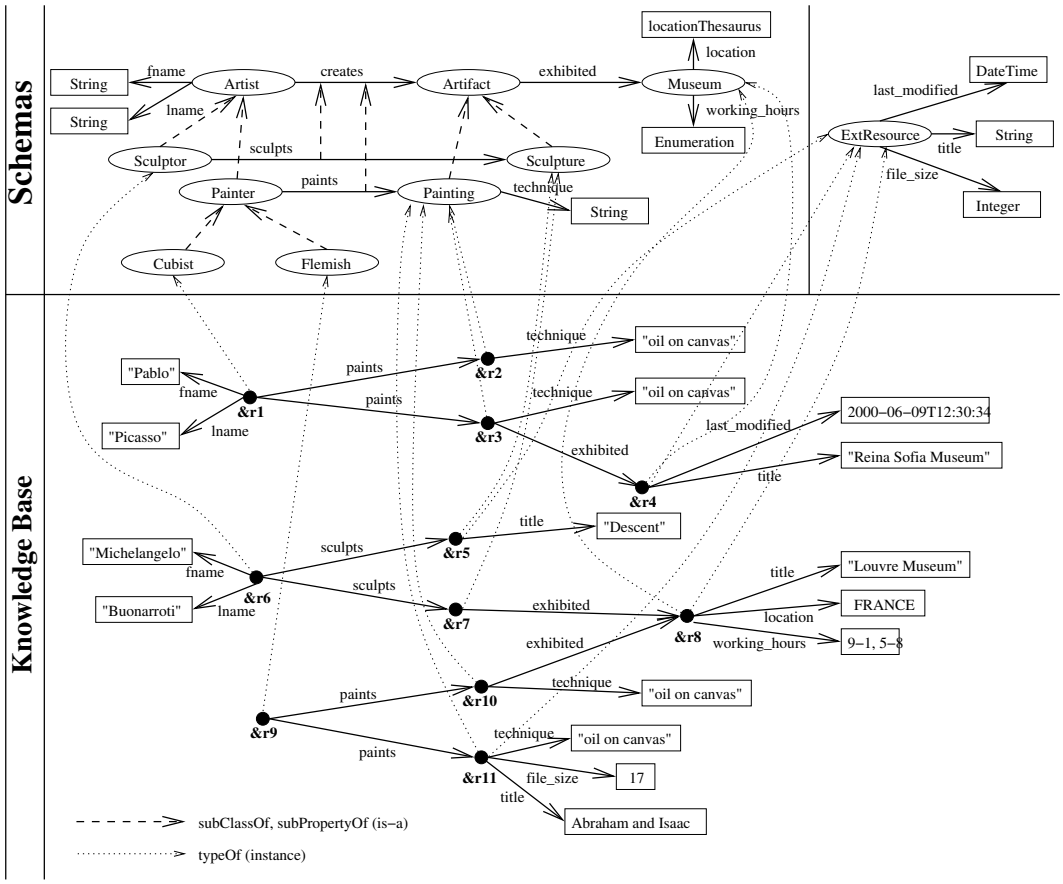


Figure 1: An example of RDF graph

demonstrates the new approach of recursive application of the graph transformation, the grouping of components and an example of the creation of the index and demonstrates the implementation of the ρ path and connect operators. Section 6 outlines possible improvements to the indexing structure that is designed in this paper. Finally Section 7 concludes the whole paper.

2. PRELIMINARIES

The RDF graph depicted in Figure 1 is visualization of an RDF and RDF Schema notation. These two languages are used to state the meta information about resources. The following subsections briefly describe this technology. In the scope of this paper the RDF is used to create the knowledge base and the RDF schema to build the schema parts of the RDF graph.

2.1 RDF

The abbreviation RDF stands for Resource Description Framework and according to [5] is supposed to be a foundation for processing metadata. It basically provides a data model for describing machine-processable semantics of data. The RDF statement is a triple (S, P, O) whose parts stand for Subject, Property and Object. Subject is usually identified by URI. It is basically a resource. The object can be either an explicit value or a resource also. Since this triple itself can be considered as a resource it can appear in an RDF statement as well. This means that the data model can be envisioned as a labeled hypergraph (each node can be an entire graph) where an edge between two nodes represents the property between a subject and an object.

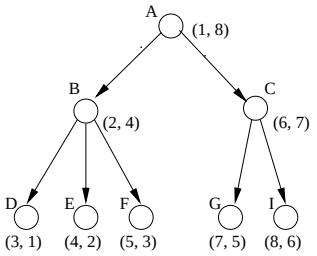
2.2 RDF Schema

Because the modeling primitives of RDF are so basic, there is no way to define the class-subclass relation. Therefore an externally specified semantics to some resources was provided. Such enriched RDF is called RDF Schema [3]. Those specific resources are for example rdfs:class and rdfs:subclass.

In such enriched environment we are able to define a simple model of classes and their relations. This can be used to define simple ontologies in the web space. The RDF Schema statements are expressed using XML together with its specific namespace. Even RDF statements can be expressed using XML with its specific namespace.

3. RELATED WORK

To make the best of the ρ operators, they should be implemented into an RDF querying system. One of such implementation is presented in [6]. The effort described there demonstrates an implementation of ρ path operator above the RDF Suite [4]. The implementation cornerstones are



Sig T = (A, 1, 8), (B, 2, 4), (D, 3, 1), (E, 4, 2), (F, 5, 3), (C, 6, 7), (G, 7, 5), (I, 8, 6)

Figure 2: An example of a tree signature for a tree *T*.

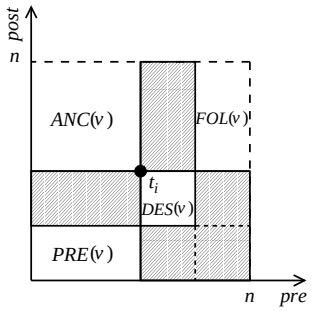


Figure 3: Properties of the preorder and postorder ranks.

two indices, Path index and Schema index. The former one is a two-dimensional array of paths - it carries the information about all paths between Class i and Class j in the schema part of the RDF graph. The latter one is used to search for a path between classes in different schemas. The Path index is very memory intensive when the data grows to large amounts. Therefore, this paper discusses a different approach to index the data for the purpose of discovering Semantic Associations.

It has been showed that the problem of searching relationship in Semantic Web is equivalent to searching paths of certain properties in directed graphs. Therefore, known conclusions and results got from the graph theory can be used to implement the ρ operators. A work described in [7] contains a solid base for such work. Unfortunately, there is not any published work discussing the use of such graph algorithms to implement the ρ operators.

4. INDEXING RDF GRAPHS

The idea of indexing RDF graph demonstrated in this paper is based on a transformation of the graph into tree or forest of trees in which the searching for relationship between particular nodes will be much easier than in general directed graph. Considering ρ-path and ρ-connect operators, the objective is to find certain paths that represent the associations among particular nodes. Therefore a convenient indexing structure to each tree is deployed to make such searching as efficient as possible. Thus the signature [10] to each tree is to be created. This approach solves the problem of getting the relationship between each pair of nodes in a tree by an atomic operation. Such relationship between two nodes in a tree is represented by their mutual position in such tree (i.e. ancestor, descendant, preceding or following node). The tree signatures are further described in the following subsection.

4.1 Tree signatures

The idea of the tree signature is to maintain a small but sufficient representation of the tree structures. The preorder and postorder ranks[1] are used as suggested in [8] to linearize the tree structure.

The basic tree signature is a list of pairs. Each pair contains a tree node name along with the corresponding postorder rank. The list is ordered according to the preorder rank of nodes. An example of a simple tree T described by its signature *Sig T* can be found in Figure 2. In the example, the preorder rank of each node is included for illustration.

Given a node v with particular preorder and postorder ranks, their properties can be summarized in a two-dimensional diagram, as illustrated in Figure 3, where *ancestors ANC(v)*, *descendants DES(v)*, *preceding PRE(v)* and *following FOL(v)* nodes of v in the particular tree are clearly separated in their proper regions. Due to these properties the mutual position of two nodes within one signature is clear immediately after reading a record of either of them in the particular signature.

According to the signature structure the basic tree signature can be further extended. To each entry a pair of preorder numbers is added. Those numbers represent pointers to the *first following*, and the *first ancestor* nodes of a given node. If no terminal node exists, the value of the first ancestor is zero and the value of the first following node is n+1, where n is the number of nodes in a tree. Such an enriched signature is called *extended signature*. Later on when we refer to signature we will mean the extended one.

4.2 Transforming the graph into forest of trees

The structure of the RDF Schema and the knowledge base can be envisioned as a directed graph with arcs provided with labels, example is shown in Figure 1. The inconvenience of this structure lies in the problem of searching path between nodes. Such searching algorithms work with great time computational complexity.

Because the structure depicted above is not really a general directed graph, we can get the benefit of the schema part of the structure since it carries useful information about the knowledge base. The schema part has the same function as a schema in the relational database. Then if we could reduce the problem of searching in the whole graph to the problem of searching in the schema, which is considerably smaller, we could use the same algorithms with better time complexity results. But since the graph can contain several schema definitions and the resources can be derived from more than one schema, the desired paths can only be found using the real data because they would not be included in the schema definition.

[1] How the preorder and postorder ranks are obtained please refer to [10].

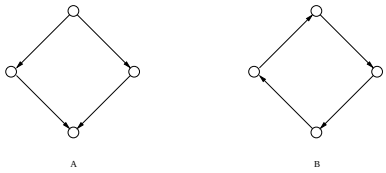


Figure 4: Directed graphs that are not trees.

4.2.1 Knowledge base transformation

A tree can be defined as a directed graph in which is true that (1) each node has zero or one incoming edge and (2) it does not contain a cycle. Directed graphs marked as A and B depicted in Figure 4 break those rules respectively. The transformation of the directed graph into forest of trees lies in the removal of such problematic cases.

If we consider the problem marked as (A) in Figure 4, part (1) in Figure 5 shows a transformation to achieve structure conforming to the rule marked as (1). The black node in the phase 1 in Figure 5, means that the node will be 'divided' into two nodes in the following phase. The next phase has two alternatives, phase 2a demonstrates the division of a node with a duplication of all descendants to all divided nodes. Phase 2b shows the division without duplication. The right way to handle such situation is to use the latter method since it prevents the uncontrollable growth of the structure. This assures that the structure will grow in linear space instead of possible exponential growth. The descending nodes should be cut off into stand alone component to avoid 'short cuts' within one component. This becomes important in the moment of finding paths between nodes.

Thus the whole graph is traversed and all the nodes that have more than one incoming edge are divided into exact amount of nodes that is the number of that node's incoming edges. This transformation can lead to breaking the graph[2] into several components. These components are either trees or directed graphs containing a cycle. To identify which components are trees a rule that a graph is a tree only if it has exactly $n+1$ edges, where n represents the number of nodes in a particular component. The non-tree components are then transformed as follows.

The transformation of the directed graph containing a cycle is depicted in the part marked as (2) in the Figure 5. The spanning tree of such component is found and the nodes, which edges are not contained in the spanning tree are divided. The transformation works in the way that it divides the particular node into two, that the first one contains all the edges that have the original node as the terminal one, and the extra node has all the edges that had the original as a initial one.

Obviously, after transforming all the non-tree components, we get a forest of trees representing the original graph. Of course we have to store the information about the divided nodes to assure that no information contained in the original graph will be lost in the new structure. Such information is stored in two inverted files where the first one is used to get all the multiple nodes[3] in the particular signature, and the second table stores to each multiple node all signatures it appears in. Those two inverted files connect the components back into the original graph.

The time computation complexity of the transformation of a general directed graph into forest of trees is estimated to $\mathcal{O}(4 * card(E))$ in the worst case. The algorithm traverses the graph to identify the nodes with more then one input edge and divides such node, this can be done at most the total number of edges in the graph. Thus the complexity depends rather on the number of edges than the number of nodes.

4.3 Motivation for the recursion

Once we have obtained the desired forest of trees we create a signature for each component (tree) of the transformed graph which together with the additional information about multiple nodes will represent the index to the original RDF graph. The time computational complexity of such operation is equal to $\mathcal{O}(n)$ since the algorithm used traverses each node in each component once. The additional information connecting signatures together is built along and deploys only atomic operations. Such information about the multiple nodes is represented by two inverted files. One has in each row a name of a multiple node together with a particular signature or signatures it appears in. And the other one has a row for each signature with a list of multiple nodes contained in it.

Above such index algorithms implementing the ρ path and ρ connect operators have been designed. The outline of those algorithms is demonstrated in the following sections. The more detailed insight into those algorithms can be found in [2].

4.3.1 Algorithm for discovering paths

This algorithm returns an answer whether there exists a path between two nodes. The algorithm traverses the forest of trees only in one direction, so to tell whether the path between two nodes exist we have to switch the start and end node and deploy the algorithm again if the search has not been successful for the first time. As a by-product it also creates a list of multiple nodes that lie on the path between the two nodes. The exact path is not computed

[2]We consider that at the beginning the graph consists from only one component.

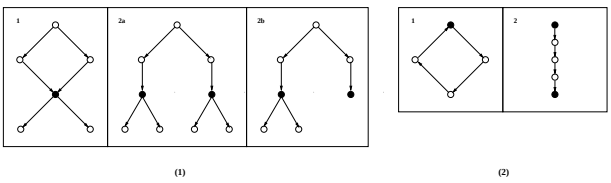


Figure 5: Transformation of a graph to conform with rules (1) and (2) respectively.

[3]A node which was represented as a one in the original graph, but is represented by several nodes in the new structure.

at this point. Another function to which this list is passed takes care of the exact path computation. To make the most from the tree structure of this index, the path is computed from the bottom to the top, the first ancestor pointer from the signature is used to traverse the path.

Therefore the algorithm traverses the index structure in only one direction, from bottom to top, it has to be deployed twice unless the path has not been found in the first deployment. Thus to check whether there is not a path between two nodes we have to execute the algorithm twice with both nodes used as a starting point respectively. This implies that the time computational complexity of finding a path between two nodes mainly depends on existence of such path and in the worst case is $\mathcal{O}(n)$. The problem of dual execution could be solved if we could tell the mutual position of the two nodes in the indexing structure. Then we could deploy the algorithm exactly once with the correctly chosen starting node.

4.3.2 Algorithm for discovering connections

As for the ρ connect operator, the nature of the designed index structure implies that the connection, the intersecting node, can only be a multiple node. Therefore the problem of finding two paths that intersect is reduced to finding a multiple node, to which exists a path from either node. So this searches the index structure in a direction that the edges have. Its starting nodes are the two nodes to which it is looking for connection.

Throughout the algorithm a set of multiple nodes, nodes which lie below the particular starting node and are possible intersection, a set of checked nodes, nodes through which the algorithm already switched to different signatures and got all usable multiples in it, and a set of to do multiple nodes, nodes that have to be still checked, are built to each starting node. In each cycle iteration those sets are updated for each starting node separately, each starting node gets one turn to check one multiple node. At the end of each iteration, the algorithm checks whether there is a non-empty intersection of possible intersecting nodes and if such intersection exists, it checks whether there exist paths from this node to both starting nodes.

The above outlined algorithm for finding path intersection also very intensively depends on the existence of such intersection. So far we can not stop the algorithm without searching the entire index that is reachable from the two starting points. It obviously also suffers from the impossibility of telling the mutual position of two nodes in the indexing structure. Therefore the time computational complexity is unacceptably high when looking for a connection that apparently does not exist in a very large graph.

4.3.3 Summarization

As is discussed at each of above algorithms, they both suffer from the ignorance of mutual position of the signatures in the index. Therefore in a cyclic graph, the algorithms have to search throughout the whole graph, in the means of the indexing structure, to check almost all signatures, to find all paths between two nodes. Though the indexing structure contains considerably less signatures than the original graph contained nodes. After all, the ignorance of the mutual position of two nodes in the indexing structure can be seen clearly at the path algorithm, it can not decide which node should be the starting one.

Another drawback presented by the above algorithms caused by the ignorance of the mutual position is that the output of the algorithms is some path or a connection, not all paths and connection as it would be desired.

Hence, it is logical to improve this indexing structure by another level that would ease the problem of telling the mutual position of nodes in the graph and that would also make possible to instantly query for all paths of desired properties. The notion of the second level is the use of the same idea of transforming the graph into forest of trees and that is exactly the aim of the approach discussed in the following section.

5. RECURSIVE APPLICATION OF GRAPH TO TREE TRANSFORMATION

The tree signatures together with its inverted files fully represent the original RDF graph. In the first sight, this information can be used to create a undirected graph, where individual signatures represent vertices and divided multiple nodes represent edges between the particular vertices. Since we would like to apply the graph to forest of trees again, the undirected graph is not desired. But under closer investigation, directions to the edges in the newly built graph can be added. This can be done by taking into account the fashion in which the multiple node has been divided and a direction of edges pointing to and from it. Basically, the node that has been divided represents a set of new nodes, those can be divided into two groups, one that contains nodes that have out-coming edges and the other group containing those that have only incoming edge. Then an edge is created for each signature containing a node from the latter group and the signature from the former group with this direction. This idea represents the direction of a path in the RDF graph and is represented in Figure 6.

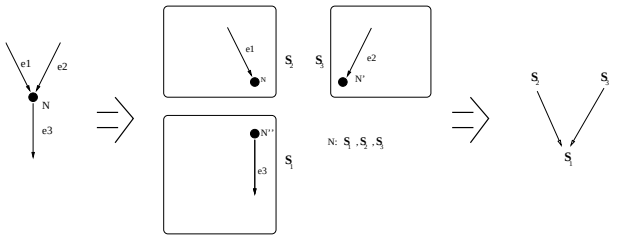


Figure 6: Giving the edges between signatures its directions.

From the graph theory we can represent directed graph by its incidence matrix. Since each matrix also represents some relation a transitive closure of such relation can be built to get immediate information about connectivity of each pair of vertices. Such transitive closure is $t(M) = \sum_{n=1}^{\infty} M^n$. Where M is our incidence matrix. According to GT the number of powers computed is at most the size of the matrix. According to graph theory, each step in the computation represents paths of length l between two vertices where l is the current power. Then each number in the $t(M)$ is

Step	# of nodes in original graph	# of nodes after transformation	# of components	Avg # of nodes in component	Min nodes in component	Max nodes in component
1	169,271	273,140	17,453	15	2	2578
2	15,214	77,526	11048	7	2	924
3	10,109	62,756	9,194	6	2	822
4	8,834	59,360	8,574	6	2	789
5	8,402	58,312	8,324	7	2	782
6	8,283	58,033	8,256	7	2	779

Table 1: Summarization of individual steps of graph to forest of trees transformations.

equal to the amount of paths between the two vertices. If we accompany this number by a set of the particular paths as they were built during the computation of the transitive closure, we could immediately tell whether there is a path between any two nodes and further more, we could tell through which vertices it goes.

Such an approach could be used directly on the original graph, but the problem is the memory intensiveness of such a solution. The size of the matrix would limit this solution to relatively small numbers of nodes in the original graph. The mentioned small numbers represent thousands of nodes but the real data can comprise of hundreds of thousands of nodes.

As for the matrix, if the transformed graph comprises of larger amount of nodes than is our limit for creating incidence matrix the whole procedure can be applied again. The matrix does not have to be built since we are transforming the graph into trees in which we can tell the mutual position of each pair of nodes.

5.1 Overhead in the result

The data that is being used to investigate possibilities of the designed structure is a part of the Open Directory Project[4] that is in RDF format and represent a graph comprising of about 170,000 nodes.

If we apply the graph to forest of trees transformation to this RDF graph we get a new graph comprising of 273,000 nodes and 17,000 components. Then the average amount of nodes in a component is 15 nodes. The maximum and minimum number of nodes in a component is 2500 and 2, respectively. This unbalance rises from a fact that we took only a part of the whole Open Directory by extracting the first 100,000,000 lines of the RDF dump of the Open Directory Project. This concludes that the graph on the input contains more then one component. The indexing structure comprising of the signatures of the individual components and the inverted files is then created. As the newly created graph comprises of 17,000 nodes against the 170,000 of nodes of the original graph, it is still too much to create the incidence matrix and to compute its transitive closure. Therefore we apply the same transformation again. Table 1 summarizes the information about the parameters of the transformed graphs.

The results depicted in Table 1 demonstrate that the recursive application of the graph to tree transformation considerably diminishes the amount of the nodes in the transformed graph. It also shows that this progress converges to some limit, in this case the limit is around 8 thousand. The observation also is that the difference between the third and fourth application of the transformation is so negligible that makes it questionable if the transformation is worth the effort since the overhead of the added nodes is considerably great.

The overhead of newly added nodes due to the transformation in the first step is about 60%. In the following steps, the overhead is about five to seven times the amount of transformed nodes, but on average it is a 38% of the number of nodes in the original graph. But we should consider that in the each following step the resulting set of signatures must not contain the unique names of dived nodes, it can directly point to a particular signature on the lower level to which the particular divided node refers.

5.2 Putting signatures into groups

The preceding section concluded that the recursive application of the graph to forest of trees transformation reduces the amount of nodes in each step. In the first few steps the number of reduced nodes is significant. In large scale data it still could not be enough to make possible to built the incidence matrix due to the limitation of a slow convergence to a certain limit. Therefore we have to deploy other approach to diminish the number of nodes in the transformed graph. Still it has to be done in a way to preserve the properties of the designed indexing structure.

To get over the limit of the recursive application of the graph to forest of trees transformation we have to find another way to decrease the number of nodes in the transformed graph. The transformation used is putting signatures into groups.The particular group is also accompanied by the incidence matrix to maintain the information about the reachability of the particular signatures in that group. Speaking in the terms of graph theory we allocate subgraphs of a certain size which are then represented as a single node in the transformed graph. Hence we have to store the incidence matrix or its transitive closure the size of the subgraph must be considerable.

5.2.1 Multiple edges between groups

In the respect of allocating subgraphs in the graph there arises a problem of multiple edges between them. This means that there would exist at least two different edges of the same direction between two groups. This fact represents that there are at least two different paths between two groups represented by one sequence of components. A field in the matrix stores unique sequence of components representing the path, but if we had more then one edge between two groups the sequences stored at that field would not be distinct. This would lead into problems with path computation on the lower level.

So if we avoided such multiple edges between groups the path stored in the matrix would directly correspond to a sequence of nodes on the lower level.

But the problem is that the limitation of single edges be-

[4]Can be found at http://www.dmoz.org.

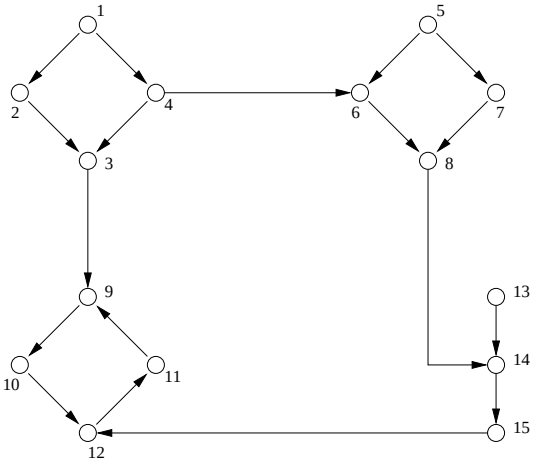


Figure 7: Phase 0: Original graph.

tween groups can lead into a hard computational problem of finding such subgraphs that would satisfy such condition.

5.3 Searching the Rho associations using the indexing structure

Let now investigate the usage of the designed indexing structure on an example of searching associations between nodes. Figure 7 represents a graph where we would like to search the associations. Firstly, the transformations applied to the original graph will be presented. Then the usage of the index that is created along the transformations of the graph will be demonstrated.

5.3.1 Phase 1

Firstly we apply the graph to forest of trees transformation. The result of such transformation is depicted in Figure 8. The groups in frames represent tree components. The black nodes indicate the nodes that had been divided. The lists on the right represent the inverted files that store the information about multiple nodes in the tree components. On the left side of the figure, the smaller graph represents the new graph considering each component as a single node and the edges of the graph represent the transitions among the tree components.

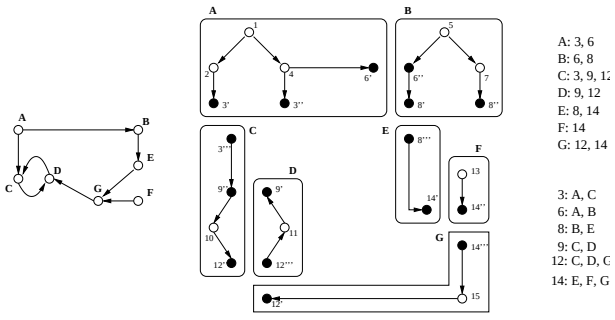


Figure 8: Phase 1: The graph to forest of trees transformation.

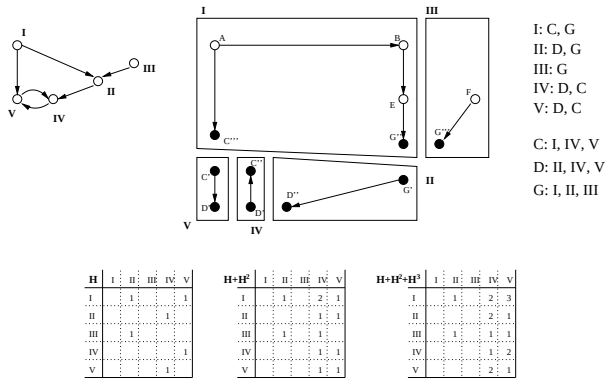


Figure 9: Phase 2: The graph to forest of trees transformation applied on the result of Phase 1.

5.3.2 Phase 2

To demonstrate the recursive application of the graph to forest of trees approach the result of the Phase 1 represented as a graph is taken and the same graph to forest of trees transformation is applied again. The result is depicted in Figure 9. The matrices in the bottom of the figure represent the computation of a transitive closure of the first incidence matrix H. Computing the third power of the incidence matrix is enough to get the whole transitive closure since in the following powers the numbers are increasing due to the cycle between node IV and V but does not compute any new path.

Again, the boxes in Figure 9 define newly transformed tree components and the inverted files on the right side transfer the information about the connectedness of the divided nodes from the lower level of the indexing structure.

5.3.3 Composition of the resulting indexing structure

The resulting composition of the designed indexing structure after Phase 2 is depicted in the Figure 10. The bottom level number 0 represents the nodes of the original graph. The level marked as level 1 corresponds to the graph after the first transformation to forest of trees. The highest level of the structure contains the transitive closure of the matrix H. The matrix H is gained observing transitions among components of the graph after the last transformation or grouping. In our simple example the matrix H represents the graph where the nodes are the components got in the Phase 2 and the edges are the transitions between those components. The transition are made through the multiple nodes created during the transformation of the graph into a forest of trees.

5.3.4 ρ path operator

As it was already mentioned in Section 1, the ρ path operator returns a set of all paths between a pair of nodes. Figure 11 demonstrates the procedure of searching paths in the indexing structure proposed in this paper. The searches are executed above the indexing structure built for the original graph depicted in Figure 7. The pair of nodes to which the ρ path is searched are "1" and "10". Since all nodes carry also the information into which components of the

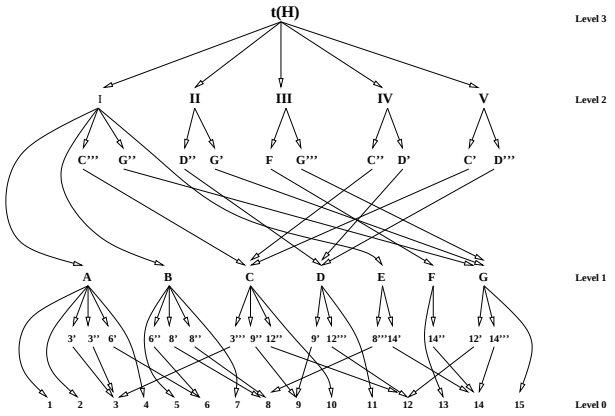


Figure 10: Visualization of the indexing structure.

transformed graphs they belong, the names of the top level components to which the nodes "1" and "10" belong are retrieved. In this case those components are "A" and "C" in the second level. On the third level it is "I" for the node "1" and "I", "IV" and "V" for the node "10". Generally, when looking for the components to which the examined nodes belong, the visualization of the indexing of the indexing structure can be used. It starts at the lowest level and goes against the direction of the edges to get to the root of the graph. Therefore, in case of the node "10", the node "C" is visited on the second level and nodes "I", "IV" and "V" are visited on the third level of the indexing structure. This idea is in the opposition to the notion of the usage of the indexing structure, since it is used in the top down fashion.

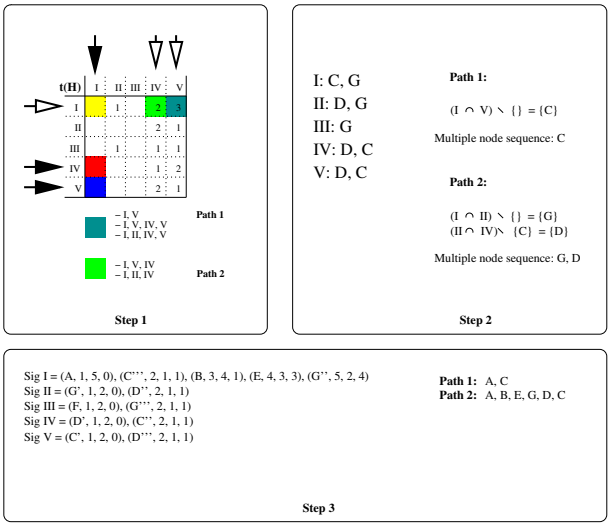


Figure 11: Searching paths using the indexing structure.

The fields (I, I), (I, IV), (I, V) where the paths from "1" to "10" should be stored and the fields (IV, I) and (V, I) for paths going from "10" to "1" are retrieved from the top matrix $t(H)$. This part of the procedure is depicted in Figure 11 as a Step 1. In our example, the fields (IV, I) and (V, I) are empty, that indicates that there does not exists any

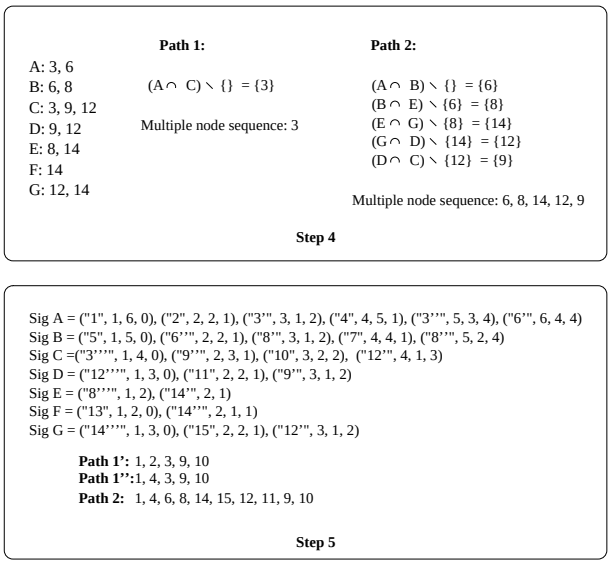


Figure 12: Searching paths using the indexing structure part 2.

path either from "IV" to "I" or "V", "I" in our original graph and therefore there does not exist any path between nodes "10" and "1". The field (I, I) is a special case that is examined directly on the lower level because it falls into a scope of one signature where the mutual position of any two nodes is clear. The other results bring the fields (I, IV) and (I, V). Those fields contain the number 2 and 3 respectively and sets of five paths from "I" to "IV" and "V". Three of those paths can be immediately omitted in the following steps since they contain some other path as a prefix. That indicates a cycle on some of the lower levels. In Figure 11, the resulting paths are marked as Path 1 and Path 2.

In the second step of the procedure, we descend in the indexing structure one level lower and retrieve a multiple node sequence for each path gained in the first step. This method is presented in Figure 11 as a Step 2. The sequence is gained from the inverted file that stores to each component the set of multiple nodes it contains. The multiple nodes are the only nodes taken into account at this moment since only those can represent a way of getting from one component to another. The sequence then represent an order of transfers between the particular components. Basically, the path was translated from the terms of one level to the terms of the lower level.

After the sequence of multiple nodes for each path is gained, the actual tree signatures are consulted to compute the exact path from the initial node to the terminal one. In this example at the second level, the initial node is "A" and the terminal node is "C". At this point the computation takes into account the information got in Step 1 that "A" and "C" lie in the same component. Their mutual position in component "I" is read from its tree signature. Since the gained path is precisely one of the already acquired paths it is not taken into account further on. The signatures describing the individual components depicted in Figure 9 are shown

in Step 3 of Figure 11. The signatures are listed at an abbreviated way, the last number representing the pointer to the first following node is left out[5], since it is not used at this point. Using the first ancestor pointer stored at each node in the signature, the sequence of multiple nodes is transformed into a path including also the nodes that are not multiple together with the initial and the terminal node.

The search continues in Step 4 where the method descend to a lower a level of the indexing structure again. The inverted file of a components gained in the Phase 1 depicted in Figure 8 is used to compute the order of transitions between components along the path using the precomputed sequence of components from the Step 3. This procedure is demonstrated in Figure 12.

After the multiple node sequence on this level is gained at the Step 4, the tree signatures are used to compute the exact path between the input nodes. Since this is the lowest level of the indexing structure, the three sequences of nodes computed in Step 5 marked as Path 1, 1' and 2 are the actual paths and are the result of the ρ path operator applied to a pair of nodes "1" and "10" in the original graph.Path 1 and 1' are derived from one multiple node sequence got from the higher level. This was caused by the fact that node "3" is in our example contained twice in the the component "A" and from the node "1" exists a path to both occurences of the node "3".

The Steps 4 and 5 represent the same operations above the indexing structure as the Steps 2 and 3. The only difference is that each pair of steps is carried out on a different level of the index. This concludes that the general method of path computation would consist of initial Step 1 and then Step 2 and 3 repeated while the lowest level of the index is not reached where the input of the following pair of steps is the output of the preceding pair of steps. If also the grouping of components that is discussed in Section 5.2 is involved in the index, the Step 1 is also repeated at the levels that represent the grouping.

5.3.5 ρ connect operator

The objective of finding all intersecting paths going from two given nodes in the original graph maps to an objective of finding a multiple nodes in the indexing structure that have those two paths as a common terminus. Those multiple nodes are the connecting nodes as are defined by the ρ connect operator. Essentially, the connecting node has to be a node in the original graph that has input degree greater than one, since it has to be a terminus of two distinct paths. But such node has to be, of course, divided during the transformation since it breaks the desired tree structure. So the connecting node can only be some of the divided nodes which are referred to as multiple ones. Hence, the objective now is to find a path to a common multiple node from both given initial nodes. The top level matrix $t(H)$ is consulted again to find such paths. The fields that lie on the row indicated by each initial node and which have positive numbers in the

[5]The format of each entry in the signature is: (name_of_the_node, preorder_rank, postorder_rank, preorder_rank_of_first_ancestor). The entries are sorted by their preorder ranks.

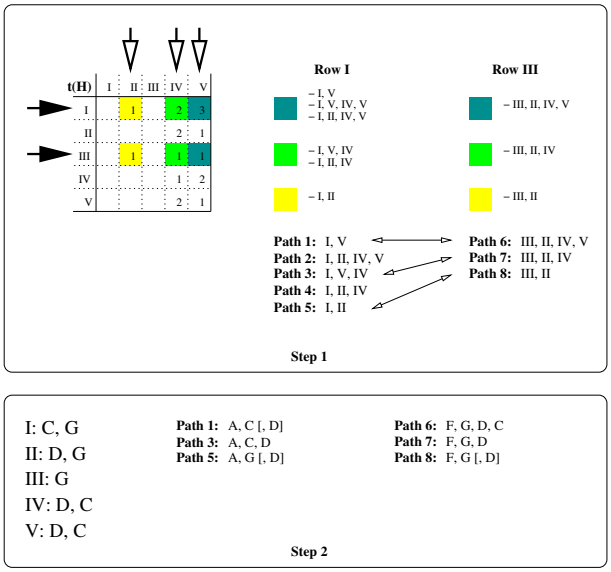


Figure 13: Evaluating ρ connect operator.

same column are retrieved because those precisely indicate a path from either node to a common place in the graph.

To demonstrate the whole procedure, the ρ connect operator is applied to a pair of nodes "1" and "13". Firstly, the information to which components the nodes "1" and "13" belong is retrieved. It is component "A" and "F" at the first level and "I" and "III" at the second level of the index. Secondly, the possible paths to a connecting nodes are retrieved using the top matrix $t(H)$. This step is depicted in Figure 13. In that figure, the pairs of colored fields represent a possible answer to the query. On the right of the matrix in this figure, the paths those fields represent are listed. Again, the paths containing cycles can be immediately omitted. The definition of ρ connection operator says that the two paths can have exactly one node in common. This fact states another rule that makes possible to omit some paths in the following computation even before verifying their connectedness. The procedure does not have to test those pairs of paths that have more than one node in common. In the lower part of the Figure 13, the arrows indicate which pairs of paths will be taken into account in the following computation.

In the second step, the procedure descends to a lower level analogously to Step 2 of the preceding section. The only difference between those two steps is that the procedure can not tell the terminus of each path in advance so it has to take into account all reachable multiple nodes in the target component. If the path on the lower level would contain cycle after adding some multiple node to it, the path can be omitted. The transformed paths are depicted in Figure 13 in Step 2. The possible extension of the path is indicated as a node in square brackets at the end of the particular path.

The procedure then descends even lower in the index where it actually finds the multiple node sequences that identify actual pairs of paths to the particular connecting node. In the example, the three connections found at Step 1 were

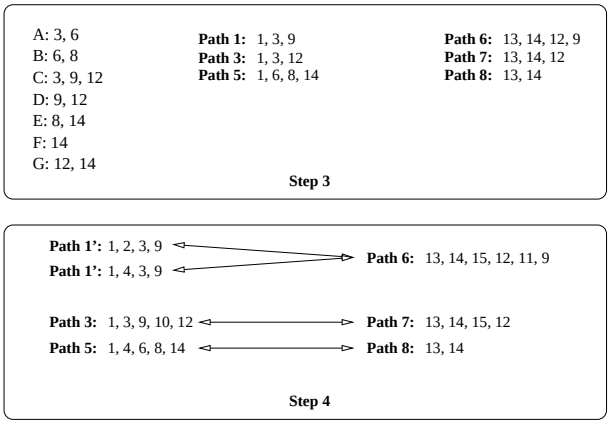


Figure 14: Evaluating ρ connect operator part 2.

all verified to be actual result of the ρ connection operator. Moreover, the first option tested turned out to be a representation of two connections at the lowest level. The reasons of the Path 1 refinement are the same as were presented in the preceding subsection. The result is presented in the Figure 14. It represents Step 4 and Step 5 of the procedure presented in the previous section. Finally, the answer to the query represented by the ρ connection applied to nodes "1" and "13" are four connections represented by seven paths and three connecting nodes.

6. FUTURE WORK

The first object for the future work is to explore the results of the proposed procedures of implementing the ρ operators using the indexing structure. Then we would like to make optimizations to the process of creating the index. As it was mentioned at the examples of using the index, there can be several paths thrown away by observing the cycles they contain. The core of the optimization is that the matrix should not contain such redundant paths at the first place. So instead using the usual matrix operations '+' and '*' during the computation of its transitive closure, a special redefined operations should be used to eliminate this kind of redundancy.

The future work will contain also an examination of optimization of the graph to tree transformation since it is the major producer of the overhead in the index. Techniques to reduce the amount of nodes added to the index by the transformation will be studied.

Finally, experiments measuring the time needed to the evaluation of the ρ operators using the index will be carried out and compared to the graph algorithms computing the same task. Some of such graph algorithms are proposed in [7].

7. CONCLUDING REMARKS

The recursion proved to be promising approach to incorporate the additional information of mutual position of nodes in the graph into the indexing structure. The inconvenience of this approach is the slow convergence to reasonable amount of nodes in the top level. Though this difficulty was solved by the grouping of components.

In comparison to the index structure designed in [6] the approach proposed in this paper solves the problem of the inconvenient size of the matrix representing the path index of the RDF graph. The matrices in our solution are used at the top levels and the size of them is all in the control of the user.

The time complexity of creation of the index discussed in this paper does not depend on the amount of the nodes of the RDF graph to which the index is created but rather on the amount of the edges. That determines the amount of nodes that have to be divided to conform with the tree structure and the height of the tree, since it also indirectly proposes the amount of individual components in the transformed graph. So in the worst case the input is a complete graph and all the nodes have to be divided. Then the amount of components is equal to the amount of nodes in the original graph so the approach of grouping of the components must be used since the transformed graph is of the same size as the original one. Then the time complexity depends on the maximal size of the matrices that are built for each group. The maximal size is set before the creation of the index.

The aim of this project is to create a scalable indexing structure for RDF graphs accompanied with algorithms providing the ρ operators functionality with acceptable time and space computational complexity. In the present time the designed indexing structure provides solid base for such work.

8. REFERENCES

[1] Kemafor Anyanwu and Amit Sheth. The rho operator: discovering and ranking associations on the semantic web. *SIGMOD Rec.*, 31(4):42–47, 2002.

[2] Stanislav Bartoň. Designing indexing structure for discovering relationships in RDF graphs. In *Proceedings of the Dateso 2004 Annual International Workshop on DAtabases, TExts, Specifications and Objects*, pages 1–11, 2004.

[3] D. Brickley and R. V. Guha. Resource description framework schema specification. 2000.

[4] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A declarative query language for RDF. In *The 11th Intl. World Wide Web Conference (WWW2002)*, 2002.

[5] O. Lassila and R. R. Swick. Resource description framework: Model and syntax specification. 1999.

[6] Agarwal Minal, Gomadam Karthik, Krishnan Rupa, and Yeluri Durga. Rho: Semantic operator for extracting meaningful relationships from semantic content.

[7] Robert Endre Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, 1981.

[8] T.Grust. Accelerating xpath location steps. In *The 11th Intl. World Wide Web Conference (WWW2002)*, pages 109–120, 2002.

[9] Sanjeev Thacker, Amit Sheth, and Shuchi Patel. Complex relationships for the semantic web. In D. Fensel, J. Hendler, H. Liebermann, and

W. Wahlster, editors, *Spinning the Semantic Web*. MIT Press, 2002.

[10] Pavel Zezula, Giuseppe Amato, Franca Debole, and Fausto Rabitti. Tree signatures for XML querying and navigation. *Lecture Notes in Computer Science*, 2824:149–163, 2003.