FI MU

Faculty of Informatics
Masaryk University Brno

rhoIndex – Designing and Evaluating an Indexing Structure for Graph Structured Data

by

Stanislav Bartoň
Pavel Zezula

FI MU Report Series **FIMU-RS-2006-07**

 September 2006

Publications in the FI MU Report Series are in general accessible via WWW:

http://www.fi.muni.cz/reports/

Further information can obtained by contacting:

Faculty of Informatics
Masaryk University
Botanická 68a
602 00 Brno
Czech Republic

rhoIndex – Designing and Evaluating an Indexing Structure for Graph Structured Data

Stanislav Bartoň
Faculty of Informatics, Masaryk University
Brno, Czech Republic
xbarton@fi.muni.cz

Pavel Zezula
Faculty of Informatics, Masaryk University
Brno, Czech Republic
zezula@fi.muni.cz

September 22, 2006

Abstract

An own design of an indexing structure for general graph structured data called ρ-index that allows an effective processing of special path queries is presented. These special queries represent for example a search for all paths lying between two arbitrary vertices limited to a certain path length. The ρ-index is a multilevel balanced tree structure where each node is created with a certain graph transformation and described by modified adjacency matrix. Hence, ρ-index indexes all the paths to a predefined length l inclusive. The search algorithm is then able to find all the paths shorter than or having the length l and some of the paths longer then the predefined l lying between any two vertices in the indexed graph. The designed search algorithm exploits a special graph structure, a transcription graph, to compute the result using the ρ-index . We also present an experimental evaluation of the process of creating the ρ-index on graphs with different sizes and also a complexity evaluation of the search algorithm that uses the ρ-index.

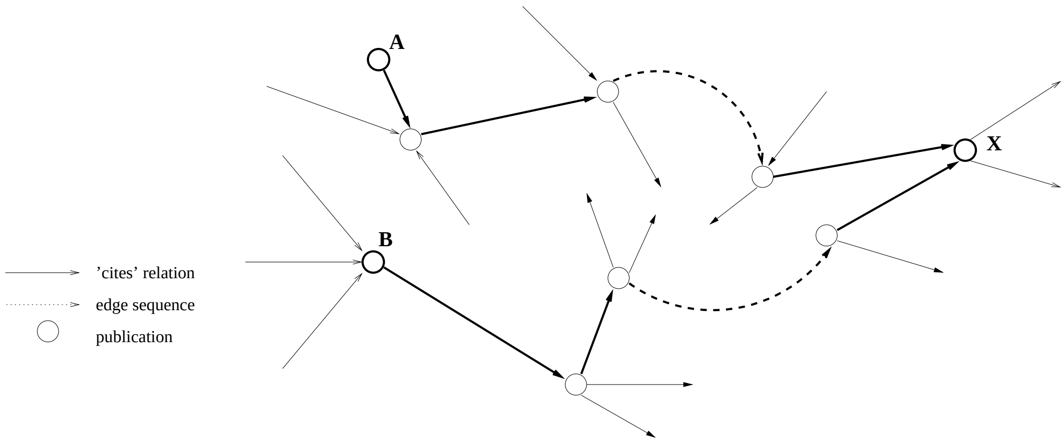

Figure 1: An example of a connection between vertices A and B. Two paths originated in A and B connected in a common vertex X.

1 Introduction

In the context of the Semantic Web, ρ-operators are proposed in [5] as a mean to explore complex relationships [16] between entities. The problem of searching for the complex relationships can be modeled as the process of searching paths in a graph where various entities represent vertices and edges the direct relationships between them. In case of the semantic web the resources or classes and edges the properties between them. The notion of complex relationships can be also identified in bibliographic digital libraries, where entities are publications and the relationship can represent references or direct citations between them.

As proposed in [5], we recognize two kinds of complex relationships. The first one is represented by *a path* lying between two inspected vertices. Speaking in terms of publications this means that one publication indirectly cites or references the other publication – a chain of publications can be built so that one cites another. The second type of complex relationship is *a connection* between two inspected vertices. This symbolizes a fact that the two inspected publications indirectly cite one common publication, see Figure 1 for an example of this kind of complex relationship.

The knowledge about complex relationships among publications can be used for example for ranking the result of the search for publications using the complex relationship discovery among entities present in the result and then sorting them according to that information. Another use case can be an automated recommendation of publications based on the preferred set of publications by searching for close connections between the publications from the preferred set. Intuitively, the complex relationship discovery has sense in any other field of interest that incorporates graph structured data. For that reason, this paper introduces an indexing technique called the ρ-index that enables efficient discovery of all complex relationships between any two inspected entities in large collections of arbitrary graph structured data.

This paper is then structured as follows, Section 2 presents related work in the field of indexing graph structured data, Section 3 is a brief insight into the design of the proposed indexing structure. Section 4 introduces a search algorithm that is used to discover all paths between any two vertices in the indexed graph using ρ-index. Consequently, the experimental evaluation of the designed indexing structure and the search algorithm is in Section 5. Finally, this paper is concluded and some directions of the future work are proposed in Section 6.

2 Related Work

The problem of answering various graph queries has two possible solutions. One is through an algorithmic on the fly query answering and the other one is preprocessing some indexing structure that would ease the computational complexity of the query processing.

Firstly we discuss one of the on the fly algorithmic approaches which is Tarjan's algorithmic solution to a *single source path expression problem* from [14, 15] which can be used to answer the queries for all paths lying between any two vertices in a graph. Hence, given a graph $G = (V, E)$ and a distinguished source vertex s, for each vertex v find a regular expression $P(s, v)$ which represents all paths from s to v in G. The problem is that the algorithm is designated to be used only on directed acyclic graph (DAG). Although, there is a transformation proposed to covert an arbitrary graph to DAG, the computational complexity of the algorithmic solution is $O(|E|)$ making it infeasible for efficient query processing.

The indexing structures that can be used for efficient search for all paths lying between two vertices in a graph were designed for RDF [10] graphs. A short example of a RDF graph is depicted in Figure 2. The first index [5] was designed directly for the purpose of implementing the ρ-operators that represent the search for the complex relationships in RDF graphs. Its concept is that it creates a matrix for each RDF schema [8] that takes part in the indexed RDF graph where each entry of the matrix represents

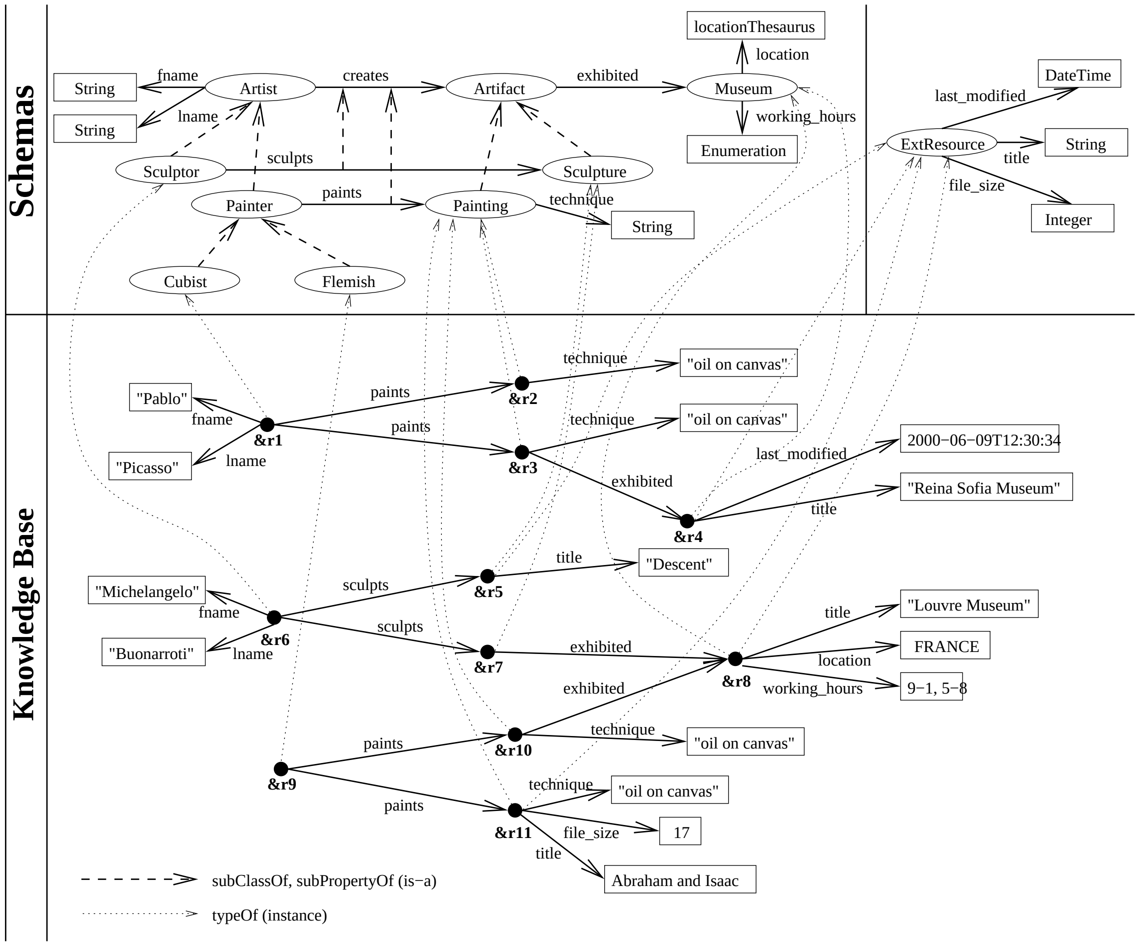

Figure 2: An example of the RDF graph.

all paths between the entities in the schema. This approach indexes only the schema part of the graph due to the computational and store complexity of the index. When candidate paths are retrieved from the index the actual existence of their instances in the knowledge base is checked.

The second indexing structure [1] that has been introduced for RDF graphs and can be used to process the queries concerning the complex relationships among vertices in a graph is based on path expressions and suffix arrays [11]. The base idea lies in extracting all possible path expressions from the indexed graph and consequently create all suffix arrays on string representations of the path expressions. The main drawback of this approach lays in its limitation of application to DAGs. Therefore, in this paper we introduce our own indexing structure for efficient query processing of path oriented queries.

Nonetheless, the search for complex relationships can be reduced to a reachability query answering. Simply, instead of returning all paths lying between two inspected vertices a single boolean value is returned answering a question whether the start vertex can reach the end one. There are numerous algorithmic approaches to solve this problem varying mostly by the structures they use to compute the transitive closure of the relation. They are either a matrix based like [3] that are based on Warshall's algorithm [17] or the graph based algorithms [12, 13] or combining both approaches which results in a algorithm [4]. The indexing structures for efficient reachability query processing are labeling schemes that stem from the XML and tree structure labeling schemes. The most popular labeling schemes are based on either interval labeling scheme [2] or on a structural approach like [9] or again combining both in a hierarchical labeling presented in [18]. Yet, these approaches can be used only to distinguish the existence of a complex relationship between two vertices, further inspection of the complex relationship itself is not possible.

3 Design of the index

The graph theory proved that a very handy representation of a directed graph is its adjacency matrix because using matrix algebra we can comfortably study the graph's properties. For instance, if the adjacency matrix is powered by two, each field in the resulting matrix contains a number of paths of length two lying between each two vertices in the original graph. If the computation continued, the result would contain amounts of all paths of an arbitrary length. Moreover, with a slight modification of the matrix that is introduced later in this section we would get not just the amounts of paths but the paths themselves.

Main difficulty of matrix representation of a graph is that its use is limited to fairly small graphs since the matrix grows in the quadratic space and the multiplication operation on matrices has even cubic time complexity. Therefore, we introduce graph transformation to enable the use of the matrix approach to graphs of arbitrary size.

3.1 Graph Segmentation

The graph transformation designed to simplify the graph we used is called *graph segmentation*. It takes the indexed graph and divides it into segments in a way that each vertex is contained in some segment and once assigned to a segment such vertex is not assigned to any other segment. Precise definition can be found in Definition 3.1. The main difference between subgraph and segment of a graph is that segment can contain edges which's both vertices are not in the same segment.

Definition 3.1. *Graph segment and a graph segmentation:*

- $\mathsf{LEFT\_V}(e) = v_1 \Leftrightarrow e = (v_1, v_2)$
- $\mathsf{RIGHT\_V}(e) = v_2 \Leftrightarrow e = (v_1, v_2)$

Segment S *in a graph* G *:* $S = (V_S, E_S) : V_S \subseteq V \ \wedge \ E_S = \{e \in E \mid \mathsf{RIGHT\_V}(e) \in V_S \vee \mathsf{LEFT\_V}(e) \in V_S\}$

Segmentation $S(G) = \{S | S \text{ is a segment of } G\} \ \wedge \forall S, S' \in S(G), S \neq S' : V_S \cap V_{S'} = \emptyset \ \wedge \bigcup\limits_{S \in S(G)} V_S = V$

Afterward, the vertices and edges between vertices within one segment form a subgraph of the indexed graph. The edges lying between vertices assigned to different segments form edges in the simplified graph. Segments then form the vertices in the simplified graph what we call a segment graph which is defined in Definition 3.2. By this transformation, multiple edges can appear between vertices in the new graph. Regardless, each multiple edge can be substituted by a single edge since from a path point of view it means a redundant information.

Definition 3.2. *Segment graph of* G*:*

$SG(G) = (S(G), X)$, $X = \{h | h = (S_i, S_j) \Leftrightarrow 1 \leq i, j \leq k \wedge \mathsf{EDGES\_OUT}(S_i) \cap \mathsf{EDGES\_IN}(S_j) \neq \emptyset\}$

where k *is the number of segments in* $S(G)$*.*

The segment graph $SG(G)$ has very similar properties as the graph G. Any path followed in the indexed graph can be observed also in the segment graph. Since we left out only the inner edges of each segment. This simplified path in the segment graph we call a sequence of segments just to avoid confusion of terms, see Definition 3.3. Intuitively, each path in the indexed path can be represented by only one sequence of segments. The method to transform a path into a sequence of segments is to replace each group of vertices and inner edges of each segment by that particular segment and to replace each edge lying between two two vertices assigned to different segments by a particular edge from X, with regard to the Definition 3.2 such edge always exists.

Definition 3.3. *Sequence of segments:*

$$\textit{EDGES\_OUT(S)} = \{e|e \in E_S \wedge \mathsf{LEFT\_V}(e) \in V_S \wedge$$
$$\wedge \mathsf{RIGHT\_V}(e) \notin V_S\}$$

$$\textit{EDGES\_IN(S)} = \{e|e \in E_S \wedge \mathsf{RIGHT\_V}(e) \in V_S \wedge$$
$$\wedge \mathsf{LEFT\_V}(e) \notin V_S\}$$

Sequence of segments $(S_1 \dots S_l) : S_1, \dots S_l \in S(G),\ 1 \leq i \leq l-1 : \mathsf{EDGES\_OUT}(S_i) \cap \mathsf{EDGES\_IN}(S_{i+1}) \neq \emptyset$

Thereafter, each of the segments can be represented by its path type adjacency matrix. A path type adjacency matrix is a modification of a usual adjacency matrix known from graph theory. It is designed to represent a graph in a path oriented way. It stores paths in its fields instead of just amounts of those paths. Initially, in each field path type matrix contains a path consisting of a single edge whenever there is an edge between two vertices in the graph. The convenience it presents over the usual adjacency matrix is that after the transitive closure of the path type matrix is computed – the fields contain not just an amount of paths lying between any two vertices, but also the paths themselves. Naturally, the mathematical operations on numbers $+$ and $*$ are replaced by the respective operations on paths – set union and concatenation.

Using the graph segmentation one large graph (G) can be transformed into a smaller simplified graph ($SG(G)$) by identifying certain number of segments and collapsing them into single vertices. The size of the segment by which we mean a number of vertices in the segment can be easily controlled. If the transformed graph is still too big to be described by its path type matrix the whole procedure can be repeatedly applied again taking as an input the already simplified graph. Thus we can acquire a multilevel indexing structure where each vertex represents a graph on the lower level.

Hence, the creation of the ρ-index accompanies a graph segmentation followed by a computation of the path type matrix for each segment. This step is repeated until we get a graph that we are able to describe by its path type adjacency matrix. A size of segments may vary on every particular level. Therefore the maximal sizes of the segments at each level form the parameter settings of the ρ-index. Examples of the parameter settings for ρ-index creation are discussed in Section 5. The visual outline of the indexing structure is in Figure 3.

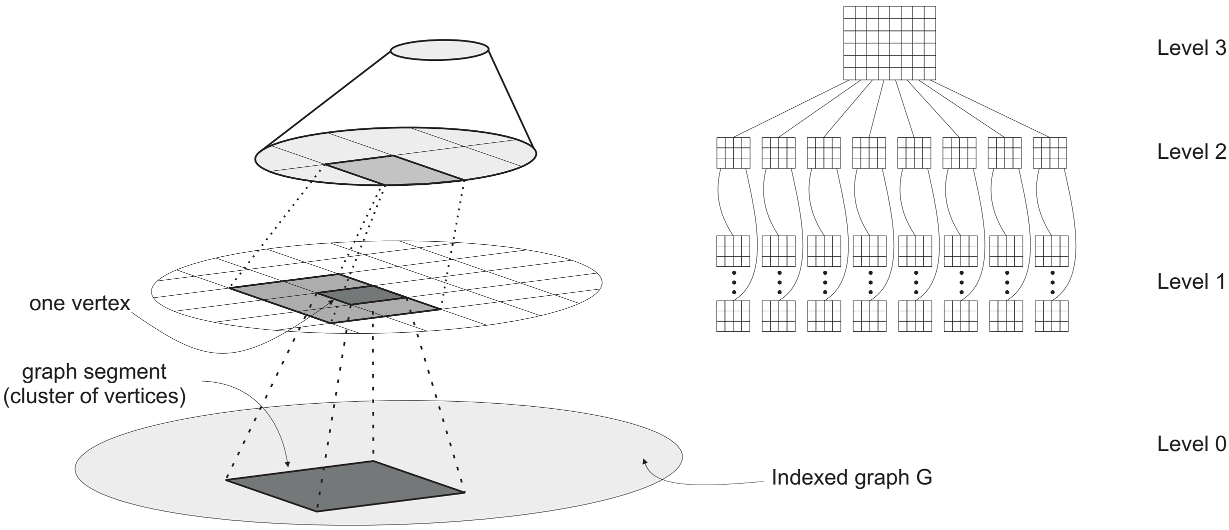

Figure 3: Structure outline of the ρ-index.

3.2 Graph Segmentation Method and Strategy

Various ways how to assign the vertices to segments have been identified and studied. One of them was a graph to forest of trees transformation which's result is a forest of trees and was proposed in [6]. Combination of vertex clustering and the graph to forest of trees transformation together with its preliminary evaluation can be found in [7]. Further implementation and evaluation showed that the graph to forest of trees makes the resulting indexing structure very tangled and therefore the search algorithm did not present good results.

Therefore, for the experimental evaluation presented in this paper we have chosen the vertex clustering as a segmentation method. Initially it puts a single vertex into set V_S. Afterwards it incrementally enlarges the segment with vertices to which or from leads an edge to this vertex. Those edges then form the set E_S. This continues until a maximal number of vertices in V_S is reached. For each level the maximal number of vertices in V_S is stated as a parameter.

The nature of the ρ-index tree structure is very dependent on the settings for the maximal number of vertices in V_S at each level. Intuitively, by setting small sizes of the segments a slim and high tree can be created. On the other hand, using a large number at first level a wide and low tree is acquired. The evaluation of different parameter settings and how they affect the search itself is demonstrated in Section 5.

3.3 Sequence of Segments Properties

As we mentioned above, each path on a lower level can be represented by some segment sequence on the upper level. Intuitively, some two different paths can be represented by one segment sequence. Some of those path are called *connecting paths* and are defined in Definition 3.4. The main property of a connecting path is that it starts with an common edge of first two segments and ends with a common edge of last two segments in the sequence of segments.

Definition 3.4. *Connecting path in a sequence of segments:*

Common edges CE_i *for* $(S_1 \dots S_l)$: $1 \leq i \leq l-1 : CE_i = EDGES\_OUT(S_i) \cap EDGES\_IN(S_{i+1})$

Connecting path $p = (e_1 e_2 \dots e_n) \in (S_1 \dots S_l) : e_1 \in CE_1 \wedge e_n \in CE_{l-1} \wedge \exists i_2, i_3, \dots i_{l-1} : 1 < i_2 < i_3 < \dots < i_{l-1} < n : \{e_2, \dots e_{i_2}\} \subseteq E_{S_2} \wedge \{e_{i_2}, \dots e_{i_3}\} \subseteq E_{S_3} \wedge \dots \wedge \{e_{i_{l-2}}, \dots e_{i_{l-1}}\} \subseteq E_{S_{l-1}}$

In general, each segment sequence can represent a huge amount of paths of different lengths. This is because each segment represents a subgraph in which paths with different lengths can be found. Important to us is knowledge of a length of the shortest path that the particular segment sequence represents. Obviously, the shortest path is one of the connecting paths. The length of the shortest path is then referred to as a *weight* of the sequence of segments. We have chosen weight instead of length because length of a segment sequence means the length of the sequence but the more important to us is the length of the shortest path it actually represents. Therefore, if we want to compute all the paths to the length l we have to store all the segment sequences having its weight less or equal to l. This parameter l then forms a path length limit that is to be indexed.

Seemingly, to compute the weight of the sequence of segments $(S_1 \dots S_l)$ we would have to compute all its connecting paths to find out which of them is the shortest. But an enhanced algorithm does not compute all the connecting paths but only one shortest connecting path for each combination of common edges picked from all CE_is, see Definition 3.4. Thus we have an upper bound on a number of connecting paths to be computed for each sequence of segments.

Due to the fact that the weight of a segment sequence represents the length of a shortest path it represents, it also represents some of the paths that are longer then its minimal weight. Therefore, using ρ-index we can compute surely all the paths to the length l but also some of the paths that are actually longer than the specified l. As we will show in the evaluation in Section 5 the amounts of paths longer then l is not insignificant, yet we realize that this fact highly depends on the nature of the data on which the ρ-index is being used.

4 The search algorithm

In this section we describe the algorithm for discovery of all paths to a certain length between two vertices in the indexed graph using ρ-index. Firstly, the algorithm looks up the segments the start and end vertex are assigned to. If we have more than two levels in the ρ-index it looks up to which segments on the upper level are assigned the segments acquired in the previous step. This continues until we reach the top level of the ρ-index or we get one common segment for both vertices. This process goes from the bottom of the structure to the top. From the definition of the graph segmentation each vertex or segment belongs to one segment on the upper level. Therefore, for each vertex in the original graph only one segment exists at each level that contains it.

4.1 The Transcription Graph

A special graph structure is used to represent the result throughout the algorithm computation. It is a transcription graph where the vertices and edges are replaced by subgraphs retrieved from the ρ-index. The vertices in the transcription graph are either the segments of the ρ-index or the vertices of the indexed graph. Those are considered to form the lowest level of the ρ-index. The transcription graph contains four special kinds of edges:

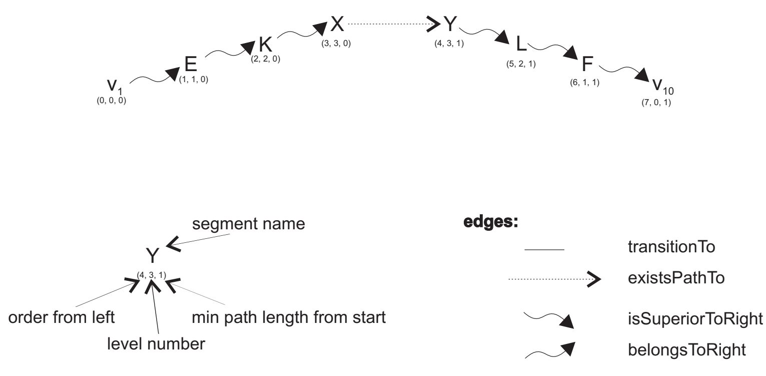

Figure 4: Example of an initial transcription graph.

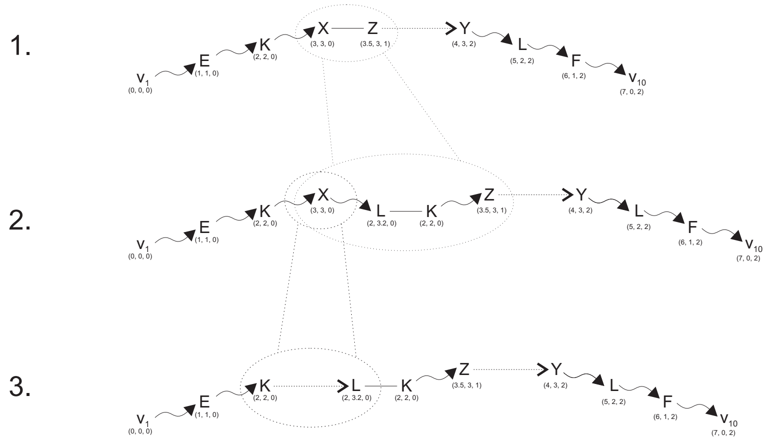

Figure 5: Transcription of a transition to a lower level.

transitionTo denotes an existance of a transition (edge) between vertices at the particular level.

existsPathTo indicates an edge that can be replaced by a subgraph from ρ-index consisting of vertices at the same level and transitions between them, representing all sequences of segments lying between these two vertices. This edge may be only between two vertices that are assigned to one common segment on a higher level.

belongsToRight represents the relationship of containment, a vertex from a lower level belongs to a vertex on a higher level.

isSuperiorToRight is an opposite of the previous relationship, it means that the vertex at a higher level contains the vertex on a lower level.

Figure 4 demonstrates an initial state of the transcription graph for a search of all paths between vertices 1 and 10 in ρ-index having four levels. The vertices are assigned to respective segments on upper levels and on the topmost level an existence of a path is supposed between the segments at the highest level.

The concept of the transcription process is to take the initial transcription graph and transform it to a graph which comprises of only vertices at the lowest level and all edges are of the *transitionTo* type. To achieve this, all the segments and edges at the higher levels need to be processed – transcribed – into entities at lower levels until we achieve the stop condition of the algorithm. Firstly, it replaces the *existsPathTo* edge by a respective subgraph of sequences of segments lying between the two vertices where all the edges are transitions. Secondly, all of the transitions concerning the particular segment, that is to be transformed into entities on the lower level, each transition originated or terminated at this segment is replaced by a subgraph of segments at a lower level connected to this segment by the type of edges connecting segments on different levels. This transformation is demonstrated in Figure 5 as the first step in the process. The transition between the segments X and Z is transformed into a transition between segments L and K, but on a lower level. This fact indicates, that there exists a border edge between segments X and Y which is originated in K and terminated in L, where K belongs to segment X and L is assigned to segment Y. If there existed any other border edges they would also appear in the transcription graph at this point.

Once the segment has only edges connecting it to other segments on a lower level it is transformed into lower level entities by connecting each entity on the left side with each entity on the right side with an *existPathTo* type of an edge going from left to right. This is demonstrated in Figure 5 by a step number 2 and 3. The transformed segment and all its connecting edges to lower levels are removed from the graph.

As for the transcription strategy during the transcription process, each vertex in the transcription graph is assigned two important numbers which are kept updated through the whole computation. The first number is the vertex's *order from left* and the other one is a length of a shortest path between the start vertex and this particular vertex. The left order number makes possible to have the vertices sorted by their position in the transcription graph as the algorithm processes its vertices strictly from left to right. Since the left order number is a floating point number, every time the process needs to insert a vertex between other two vertices there is always a gap between their left orders. Therefore, the transcription graph forms a special type of a directed graph referred to as *a network* which is also a DAG. Since, vertices can be ordered by its left order number and it is true that there is no edge pointing from a vertex with higher left order to a vertex with a smaller left order.

The length of a shortest path from the start vertex is used to limit the weight of segment sequences that are retrieved from the index to replace the path edges in the transcription graph. It considers the length of an already computed piece of path from the start vertex to the particular vertex. The segment sequences of a maximal weight of a difference of the already computed piece of the result and the maximal length of a desired path, our l, are retrieved and placed into the transcription graph. This fact assures that the algorithm will actually stop for any input because if it is not possible to reach the end vertex from a segment by a sequence of segments with a weight less then

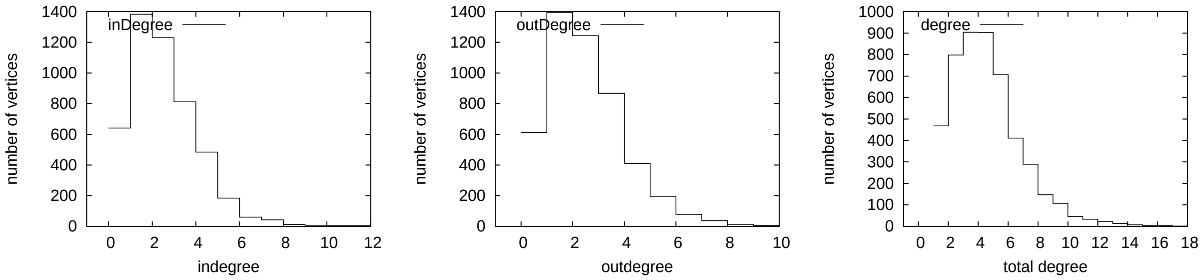

Figure 6: Vertex degree distribution in the synthetic graph G5000.

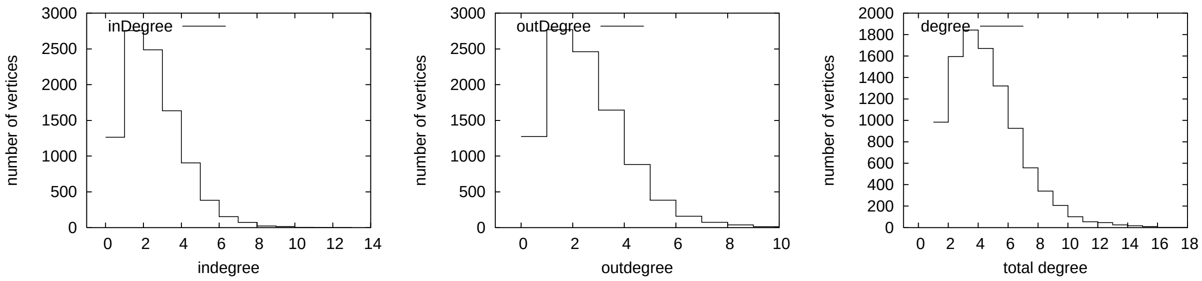

Figure 7: Vertex degree distribution in the synthetic graph G10000.

l considering the already minimal length of a path, the whole branch is removed from the transcription graph.

When the process finishes the resulting transcription graph represents either a network of all paths initiated in the start vertex and terminated in the end vertex with a length lower or equal to the predefined l and some paths longer than l due to the nature of the graph segmentation. All that with respect to the paths that are in the indexed graph. If there are no paths shorter than l between the start and end vertex the resulting transcription graph will have only two vertices and no edges.

5 Experimental evaluation

In this section we present and discuss the results gained by the indexing structure and its search algorithm introduced in this paper. As a testing data we have used generated synthetic data which's properties are described later in this section.

As follows, the set of experiments performed took as a testing data generated graphs having sizes growing from 5,000 to 30,000 vertices and from 10,000 to 60,000 edges. The graphs were generated iteratively using a small core graph in the first step. In

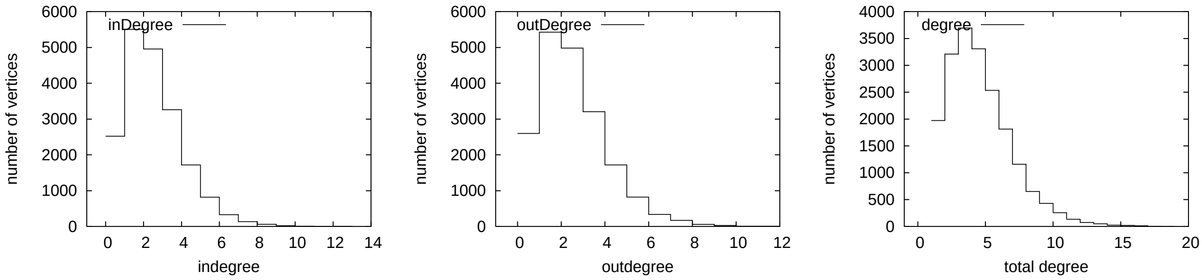

Figure 8: Vertex degree distribution in the synthetic graph G20000.

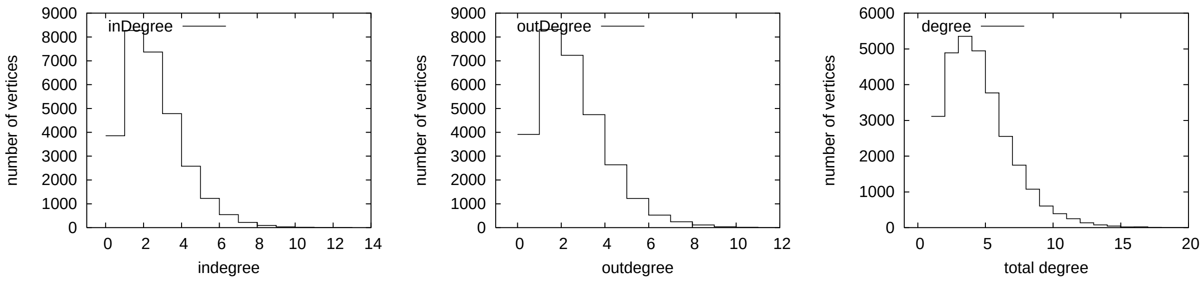

Figure 9: Vertex degree distribution in the synthetic graph G30000.

each iteration a smaller graph was enlarged by randomly adding edges between newly added vertices or between a new vertex and an old vertex with a random direction. The probability of where the edge was placed was equal to the proportion of the number of vertices in the smaller graph to the number of vertices in the newly built graph. In the rest of this paper we will refer to these graphs as G5000, G10000, G20000 and G30000 with respect to the number of vertices contained in the testing graph. The vertex degree distribution of the testing graphs is illustrated in Figures 6, 7, 8 and 9.

This way we gained graphs with different sizes and having the property that the smaller graph is always a subgraph of any of the larger graphs. This property is very important when we evaluate the experiments that compare the search results in graphs with different sizes, because the result of a search performed on a smaller graph is also a subset of a search result of the same search performed on any larger graph. So its true that G5000 $\subseteq$ G10000 $\subseteq$ G20000 $\subseteq$ G30000.

As we performed all the experiments described in the following sections we stated the maximal indexing length l to be 10. The ρ-index then was built to index all the paths up to this length and the search then returns all the paths to this length and some paths

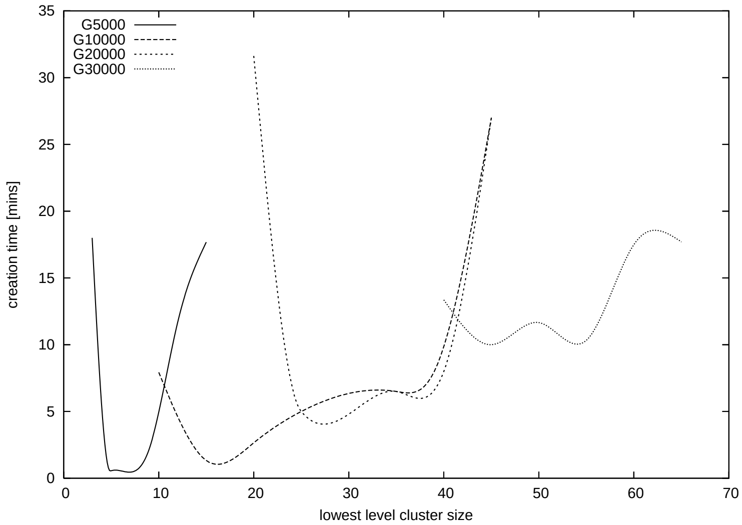

Figure 10: A ρ-index creation time.

longer. Due to the space limitations of this paper we will not present a detailed insight into what means *some* in exact numbers. Yet we briefly tackle this issue in Section 5.3.

As for the machinery on which we executed all our experiments concerning the ρ-index, the computer is a dual double core Athlon Opteron 2.4 GHz with a 12 GB of RAM. During the time the tests were run the computer was not dedicated to only that task so all the experiments were ran multiple times and the results depicted are averages of the results thus gained.

5.1 ρ-index Creation Time

First of all we present how much time the creation of the ρ-index consumes for certain sizes of the testing graphs and particular parameter setting. Figure 10 represents the experiments performed on our four testing graphs. The ρ-index created for each graph had 4 levels. The maximal size of a segment on the lowest level is represented by the values on the x-axis. The other parameters were chosen to be 10 at the second level, 5 on the third and 2 at the top level. Just to remind the parameters are the maximal sizes of the segment at the particular level.

The results of this evaluation showed that the ρ-index is sensitive to underfill of the structure. This can be observed for the case of the smallest graph when even the ideal parameter setting which is around the value 8 for the max segment size for the lowest level lead into a creation time which was greater than the best time of a graph twice as

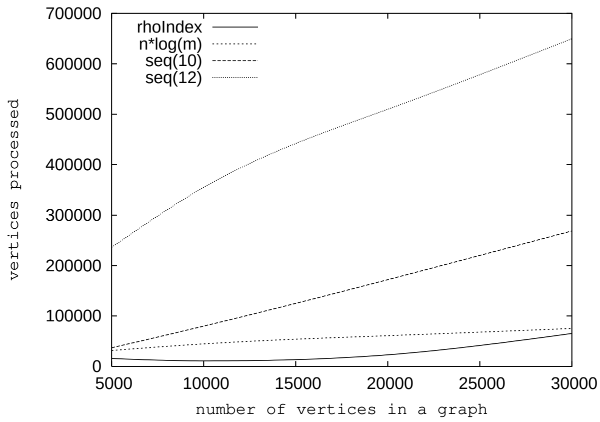

Figure 11: A ρ-index search complexity with respect to the graph size.

large. We assume that this is caused by a inadequate ρ-index setting. The ρ-index for this testing graph should have been created using only three levels or smaller maximal segment sizes at the second and third level.

The creation times of the remaining three graphs indicate that the ρ-index is highly dependent on the parameter setting. We can observe that the creation times form a curve of a parabolic shape for all graphs and the size of the testing graph determine the shift of the values on the y-axis. This implies that the optimum parameter setting can be easily predicted for graphs at this particular graph size category upon these experimental results. As a category we consider a graphs of a similar size and connectivity. In this case the category is formed by graphs having from 10,000 to 30,000 vertices and 20,000 to 60,000 edges respectively.

5.2 Search Complexity

This group of experiments performed describe the complexity of the search algorithm using the ρ-index to search all paths to a certain length in respect to the size of the graph on which the search was performed. Figure 11 demonstrates the experiments where the parameter settings were fixed and the size of the graph grew. As we mentioned earlier in this section, the result of the search of the larger graph contains all the search results of the smaller graphs, thus they are comparable.

Both parts of Figure 11 refer to the same results of the same experiments. They only differ in the y-axis scale. The left part depicts the results in the whole scale, the right part depicts them ranging from 0 to 80,000 of processed vertices.

The particular curves represent the algorithms used to perform the search of all paths lying between two vertices. The lines labeled with the prefix *seq* represent a sequential algorithm. This algorithm represents an upper bound of a way to solve the problem of searching all path between two vertices to a certain length. It is a depth first search that tries to recursively build a path of a some maximum length. The number in the label states the maximal length of a searched path.

In Figure 11 are present two results for a sequential algorithm *seq*. This is caused by the nature of the ρ-index and its search algorithm which results in a fact that all paths to a specified length l to which the ρ-index was built are returned and some of the paths longer than l are also returned by the search algorithm. That implies for the result of the search using ρ-index that is true: *seq*$(l) \subseteq \rho$-index $\subseteq$*seq*$(l+k)$ for a particular k, where the set inclusion is meant on the results of the search algorithms. For that reason we also present the complexity of the algorithm *seq*(12) which represents the sequential scan for all paths to the length 12. The rough comparison of the complexity measured for the sequential algorithm with the length set to 10 and 12 we can observe that the growth is exponential.

Another approach to the problem of searching all paths lying between two vertices in a directed graph is a direct computation using the Tarjan's algorithms described in [14] and [15]. The algorithm works in a time complexity $n * \log(m)$ where n represents the number of edges in the graph and m the number of vertices in the graph. The algorithm takes a flow graph on the input and a start vertex and returns the path expressions (regular expressions where the letters are edges of the flow graph) representing all paths to all vertices in a graph on the output. A flow graph is a special type of a directed graph which allows only one source vertex in the graph and no cycles. There exist a non-trivial transformation of an arbitrary directed graph into a flow graph. This computational overhead of the graph transformation is not included in the complexity of the Tarjan's algorithm.

In the progress of the search computational complexity of our designed index structure and algorithm a decrease of the complexity can be observed for the graphs G5000 and G10000. As we mentioned in the previous subsection, this is due to the underfill of the search structure. The parameter settings used to built the ρ-indexfor each of the testing graphs were the optimal ones for each particular graph. Again only the max size of the segment on the lowest level varied and the rest of the parameter settings remained same for all testing graphs.

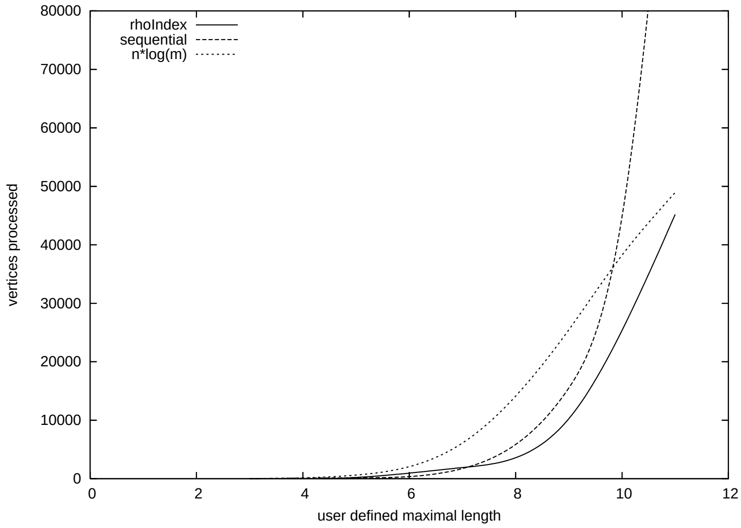

Figure 12: A ρ-index search complexity of queries with different maximal search length.

5.3 Search Complexity of Queries with Limited Maximal Length

To this point we always considered the maximal length of the searched path by the search algorithm to be the same as the maximal length that was used to create the ρ-index. In this subsection we explore the behavior of the search algorithm when the maximal length of the searched path is its parameter.

As we refer to the maximal length l of the indexed path, we refer to the maximal length of a searched path as `softL`. Setting this parameter does not limit the search to return paths longer than `softL` but again it must not necessarily find all of them.

Thus Figure 12 represents searches executed on the graph G10000 and with the parameters set to 30, 10, 5 and 2. The ρ-index was computed with l equal to 10. The x-axis then represents the values of the `softL` parameter and the curves represent the respective algorithms used to compute the result.

To make the Tarjan's approach comparable with ours and the sequential algorithm we approximated the computational complexity by limiting the input graph to only those vertices and edges that are reachable within `softL` steps.

As for the number of the found paths, the sequential algorithm finds all paths to the length of `softL`, our algorithm finds all the paths to the length of `softL` and some of the paths that are longer than that. Figure 13 represents the percentage of paths not found that have length greater than `softL` for each particular length. Although we ran the experiment for the `softL` value of 3 the curve representing returned results is not

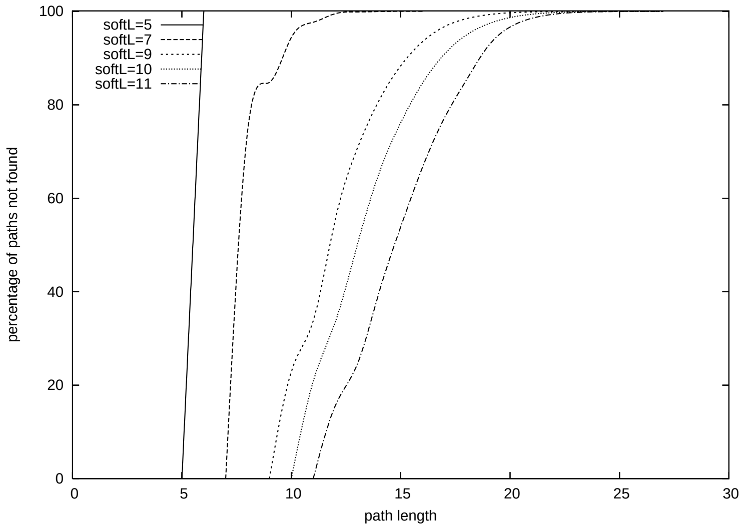

Figure 13: A ρ-index percentage of paths longer than `softL` not found.

present here since it returns no paths for this `softL` value. For `softL` value 5 it finds no path longer than 5 so the curve reaches immediately 100 percent at length 6.

At this point we have to point out that the amount of paths increases in exponential manner, what means that the amount of paths of length 12 between the testing start and end vertex actually present in G1000 is 1821 and the amount of paths of length 14 between the same two vertices is 12644. So even if we find really low percentage of the paths present in the indexed graph, their amount can easily reach tens of thousands. For illustration, for the $\mathtt{softL} = l = 10$ and a path length of 24 the amount of found paths is 72,000 and the longest found path has a length of 42.

5.4 Search Complexity Affected by the Parameter Settings

Since the ρ-index can be created for one particular graph using different parameter settings and as we could see from this section, also having different properties, we explore the correlation between certain parameter setting and the complexity of the searches performed on the respective indexing structures built upon one particular testing graph.

Again we have chosen the testing graph G10000. The parameter settings differed in the maximal size of a segment on the lowest level, the upper level settings remained the same for all tests. Consequently, Figure 14 depicts the relation between the parameter settings and the average search complexity for thus created ρ-index. This curve is falling with the increase of the cluster size. The dashed curve in Figure 14 reflects the creation

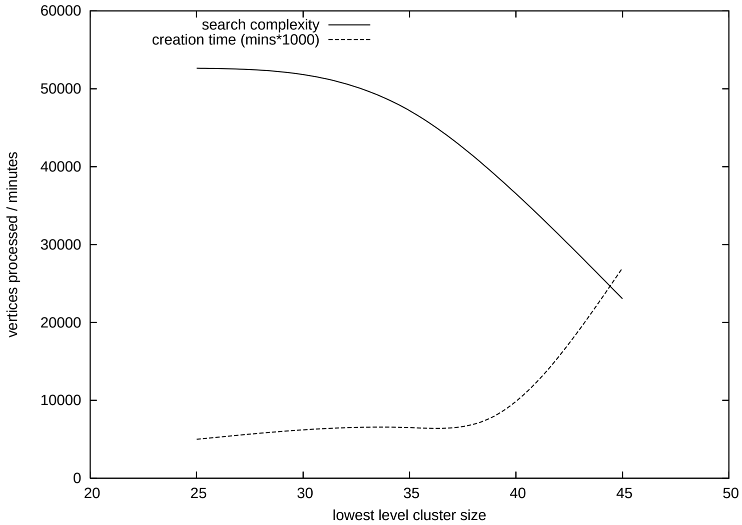

Figure 14: A ρ-index search complexity related to the parameter setting.

time of the ρ-index for that particular parameter setting. The time is is in minutes multiplied by 1000 to make the curve visible in this scale. On the contrary, the progress of this curve is rising as the ρ-index structure is becoming underfilled. We have already seen this behavior in Figure 10 at all graph sizes at the rising part of the parabolas.

These facts represent a creation and search tradeoff. We gain better creation time results for certain parameter settings but on the other hand we get worse search complexity results. This tradeoff has even one more dimension which is the amount of paths returned that are longer then l. Due to the space limitations we are not able to discuss this dependence more in detail.

6 Concluding Remarks & Future Work

Our goal was to design an indexing structure that would make possible an effective discovery of paths having special properties in a large graph. The first objective was to find all paths to certain length l between any two vertices. Also we still get some amount of paths longer than the specified l as an approximation.

For brevity, in this paper we did not presented all the experiments we have conducted in respect to explore the behavior of the designed ρ-index. For example we tackled the issue of the parameter settings modification also at the upper levels of the indexing structure and their impact on the particular ρ-index creation time and consequent search complexity.

Hence, to this relates also our next future work. Firstly we want to carry out more tests to be able to precisely predict the ρ-index properties under certain parameter settings and hence to be able to find optimal settings for the testing data. Afterwards, we would like to carry out tests on more testing data in order to investigate the scalability of the ρ-index in respect to the number of vertices and to the number of edges in the indexed data. In our near future work we would also like to implement the algorithm for discovering all connections between two vertices as the ρ-index allows such utilization according to [6].

References

[1] M. Y. A. Matono, T. Amagasa and S. Uemura. An indexing scheme for RDF and RDF Schema based on suffix arrays. In *Proceedings of SWDB'03, The first International Workshop on Semantic Web and Databases, Co-located with VLDB 2003*, 2003.

[2] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 253–262. ACM Press, 1989.

[3] R. Agrawal, S. Dar, and H. V. Jagadish. Direct transitive closure algorithms: design and performance evaluation. *ACM Transactions on Database Systems*, 15(3):427–458, 1990.

[4] R. Agrawal and H. V. Jagadish. Hybrid transitive closure algorithms. In D. McLeod, R. Sacks-Davis, and H.-J. Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 326–334. Morgan Kaufmann, 1990.

[5] K. Anyanwu and A. Sheth. The ρ-operator: Enabling querying for semantic associations on the semantic web. In *Proceedings of the twelfth international conference on World Wide Web*, pages 690–699. ACM Press, 2003.

[6] S. Bartoň. Indexing structure for discovering relationships in RDF graph recursively applying tree transformation. In *Proceedings of the Semantic Web Workshop at 27th Annual International ACM SIGIR Conference*, pages 58–68, 2004.

[7] S. Bartoň and P. Zezula. Rho-index - an index for graph structured data. In *8th International Workshop of the DELOS Network of Excellence on Digital Libraries*, pages 57–64, 2005.

[8] D. Brickley and R. V. Guha. Resource Description Framework Schema specification. 2000.

[9] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *Proceedings of the 13th ACM-SIAM SODA*, pages 937–946, 2002.

[10] O. Lassila and R. R. Swick. Resource Description Framework: Model and Syntax specification. 1999.

[11] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. In *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[12] P. W. Purdom. A transitive closure algorithm. *BIT*, 10:76–94, 1970.

[13] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on computing*, pages 146–160, 1972.

[14] R. E. Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, 1981.

[15] R. E. Tarjan. A unified approach to path problems. *J. ACM*, 28(3):577–593, 1981.

[16] S. Thacker, A. Sheth, and S. 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.

[17] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[18] X. Yan, P. S. Yu, and J. Han. Graph indexing based on discriminative frequent structure analysis. *ACM Transactions on Database Systems*, 30(4):960–993, 2005.