UNIVERSITÀ DEGLI STUDI DI ROMA TRE
Dipartimento di Informatica e Automazione
Via della Vasca Navale, 79 – 00146 Roma, Italy

Keyword based Search over Semantic Data in Polynomial Time

PAOLO CAPPELLARI[2], ROBERTO DE VIRGILIO[1], MICHELE MISCIONE[1]

RT-DIA-160 **October 2009**

[1]Dipartimento di Informatica e Automazione
Università di Roma Tre
{devirgilio,miscione}@dia.uniroma3.it
[2]Department of Computing Science
University of Alberta, Canada
cappellari@cs.ualberta.ca

ABSTRACT

In pursuing the development of `Yanii`, a novel keyword based search system on graph structures, in this paper we present the computational complexity study of the approach, highlighting a comparative study with actual `PTIME` state-of-the-art solutions. The comparative study focuses on a theoretical analysis of different frameworks to define complexity ranges, which they correspond to, in the polynomial time class. We characterize such systems in terms of general measures, which give a general description of the behavior of these frameworks according to different aspects that are more general and informative than mere benchmark tests on a few test cases. We show that `Yanii` holds better performance than others, confirming itself as a promising approach deserving of further practical investigation and improvement.

1 Introduction

Nowadays data is disseminated in a number of different sources, from databases systems to the Web, from a traditional structured organization (relational) to a semi-structured (XML), up to the unstructured ones (text in Web documents). Although availability of data is constantly increasing, one principal difficulty users have to face is to find and retrieve the information they are looking for. To precisely access data, a user should know how data is organized in the source, and how to write a query in the language required by the source. Clearly, this represents an obstacle to information access to not expert users. For this reason keywords search based systems are increasingly capturing the attention of researchers. In fact, there are approaches building on top of traditional systems, like DBMSs: the goal is to free users from knowing the details of the query language or of the structure (schema) of the data.

Keyword based search has been applied to traditional relational sources, XML contexts and graph-based structures. Examples of systems developed for relational sources are `BANKS` [2], `DISCOVER` [7] and `DBXplorer` [1]. In XML context we find works like `Xsearch` and `Xrank` as well as others like the ones from Kaushik et al., respectively [3, 5]. All of these works (in any context) exploit one main advantage: data can be represented in trees. This simplifies the problem. In a more general representation, data are represented by a graph, requiring more efficiency and accuracy of the solution. This modeling approach is living a great momentum because: (i) data from disparate sources can be modelled with a graph structure, (ii) information search can be realized as a graph exploration, the latter being a topic counting many known techniques. In this context, a combination of IR and graph exploration techniques are used to discover the information matching the keywords and to rank it in a way that is more relevant for the user. In these frameworks, generally, the first (sub-)goal is to identify the portion of the graph holding information matching the keywords, possibly relying on a sophisticated index structure. Then, the second (sub-)goal is to search for a connection between such graph portions in order to build the most complete answers (or solutions) given the keywords. Finally, third (sub-)goal, the identified solutions are ranked according to the relevance for the user and then returned.

Navigating graph structures while building and ranking candidate solutions hides a number of difficulties, especially from a performance point of view. Because systems have to be fast in returning their answers back to users, many approaches implement heuristic pruning techniques to reduce the search space on the graph, or greedy-like techniques to (try to) aim directly to the most promising solution, or threshold techniques to cut less promising solutions (Top-K approaches).

While all these techniques can lead to the development of systems that perform reasonably well in many practical cases, not much attention is given to the study of the computational complexity of keywords search based approaches. Most papers present the evaluation of their approach in terms of performance comparisons (benchmark tests) with respect to other competing approaches. Usually, a few test cases are designed, run on the different systems with performance of each system traced and reported. While this is a genuine evaluation of a system, it does not describe how an alteration of the setting (size of a query, size of the graph, and so on) affects the performance.

In this paper we want to analyze the problem from a theoretical point of view. We study and compare three recent approaches to keywords search on graph structures: `BLINKS` [6], `SearchWebDB` [9] and `Yanii` [4]. We focus on them for two reasons: (i) they represent three different paradigm to keyword search, (ii) they are the only ones with computational

complexity, as is detailed later, in PTIME. BLINKS is an example of an approach based on a sophisticated index structure, which enables for a fast identification of the graph portion relevant for the results. SearchWebDB is an example of exploitation of the graph structure to build promising queries that should return information of interest for the user. Yanii is an example of a clustering approach with an off-line indexing that enables a low runtime execution cost in which solutions are combined and returned starting from the most promising, descending monotonically.

In the discussion we are interested to obtain a description of the systems in terms of a number of different measures like the size of the query in input, the size of the search space, the number of maximum solutions returned, and others as discussed in the next sections. The outcome of our study is to define complexity ranges in PTIME for each approach and to demonstrate that Yanii is a promising approach because it shows lower computational complexity than the others.

The paper is organized as follow. Section 2 briefly describes techniques and data structures used by BLINKS, SearchWebDB and Yanii. Section 3 discusses in detail the complexity of each framework and Section 4 illustrates a comparison between them. Finally Section 5 sketches some conclusions and future work.

2 State of the art

In this section, rather than discussing of the different approaches proposed in literature, we give in-depth details of the three approaches we study in this paper.

BLINKS BLINKS [6] provides a *Bi-Level INdexing Keyword Search* approach for data graph. To discuss BLINKS we have to introduces *Single-Level INdexing Keyword Search* (SLINKS) first. SLINKS works with two kinds of indexes that keep information about the data graph. The first type of index is a set of *keyword-node lists* (L_{KN}) which keeps track of the nodes that can reach a given node matching a keyword. The second type of index is the *node-keyword map* (M_{NK}), an hash table that stores the shortest distance between pairs of nodes and keyword matching nodes. L_{KN} and M_{NK} represent together the *single-level index*.

Since this index contains $|G| \times K$ entries, where $|G|$ is the number of nodes and K is the number of node matching keyword, it is impractical for large graphs. To overcome this problem, BLINKS partitions the entire data graph in many sub-graphs called *blocks*. The *Bi-Level Index* consists of the top-level *block index* and one *intra-block index*. The former maps information about nodes and keywords within the block, the latter indexes information inside the block similar to the *single-level index* for the whole data graph. The node-based partitioning used in BLINKS presents some node in common between different blocks: these particular nodes are called *portals* p_i. They can be *in-portals* in a block if they have at least one incoming edge from another block or *out-portals* if they have at least one outgoing edge to another block or both. More in detail, the *intra-block index* contains:

- *Intra-block keyword-node lists* L_{KN}: the lists of nodes in a given block that can reach a given keyword matching node without leaving the block.
- *Intra-block node-keyword map* M_{NK}: the shortest distances between pairs of nodes within a block.

- *Intra-block portal-node lists* L_{PN}: the lists of nodes in a given block that can reach a given out-portal node within block without leaving the block.
- *Intra-block node-portal distance map* D_{NP}: the shortest distances between a given node in a block and his closest out-portal within the block.

While the *block index* contains:

- *Keyword-block lists* L_{KB}: the lists of blocks containing keyword matching nodes for a given keyword q_i.
- *Keyword-block lists* L_{PB}: the lists of blocks containing a given out-portal p_i.

`BLINKS` provides expansions for both forward and backward search. In fact, backward expansion is used to expands from a keyword matching node to other nodes using inverse edges and forward expansion is used through the D_{NP} and M_{NK} to infer if a path from a node u to a keyword matching node within the block is sub-optimal, i.e. the path is optimal for the solution having u as root.

SearchWebDB `SearchWebDB` [9] presents alternative techniques to solve the problem. First of all in the indexing phase a graph reduction is applied, i.e. the initial data graph is reduced to yield to a *summary graph* that contains only some kind of nodes and edges. Authors' intuition is that only nodes representing classes need to be stored in the working memory, independently from the query request. This is because nodes representing entities and containing values are ending elements for the graph, i.e. they are not connecting elements. Once they run the algorithm with a query we compute the *augmentation*: the summary graph is augmented with nodes and edges by which we can reach the keyword matching elements. The resulting graph is called *augmented summary graph*. The algorithm does not return top-N sub-graphs as solutions but *conjunctive queries* from which the user can perform more precise selections. The final selection is submitted again and it is computed using the database engine with an element-to-query mapping. All the conjunctive queries correspond to sub-graphs of the augmented summary graph. Therefore top-N queries can be compared against the top-N solutions of the other keyword search algorithms.

Now, let us take a look at the data structures used to implements this approach. The searching is supported by a *cursor* that allows to keep track of the visited paths. In detail, a cursor c is represented as $c(n, k_i, p, d, w)$ where n is the graph node just visited, k is the keyword matching element from which the path starts, p is the parent cursor of c, d and w represent respectively the distance (length) and the cost of the path. To a cursor c it corresponds the path between n and k, easily determined by a recursive traversing of parent cursors. This characteristic is useful when we need to generate a possible sub-graph solution with n as root of the sub-graph. In fact, we have to merge the paths starting from n and terminating in at least one matching keyword element.

Yanii `Yanii` [4] is based on a clustering approach. A cluster is a partition over paths in the graph starting from a *root* and ending into a node matching a keyword. We mean root as a node without incoming edges and we call such paths as *informative paths*. At each path corresponds a $template$, i.e. the order list of label on the edges in the path. Each template defines a cluster and consequently all the paths with the corresponding template belongs to it.

In order to retrieve efficiently informative paths into a graph, we use *Lucene*[1] to index the data graph. An interesting characteristic of `Yanii` is that it computes top-N solutions in the first N iterations of the algorithm, one solution at each iteration. It combines solutions by merging the highest-scoring paths from the several clusters, in general one path each cluster for each solution. It merges more paths from the same cluster in the case of the same scoring. `Yanii` has an off-line pre-processing step in which an index structure associating a root node with values (possible keywords) is built. Then, `Yanii` has an on-line process composed of the following steps: (i) the query submission for the path retrieval, (ii) the clustering and (iii) the construction of the top-N solutions.

3 Complexity of Keyword Search

Formally, the problem we are trying to solve may be defined as follows. Given a directed graph $G = (V, E)$, where each node (resource) $v \in V$ and each edge (property) $e \in E$ present a label (i.e. the URI of the resource, the name of the property), a query Q composed of a set of keywords $q_1, \ldots, q_m$, we find the answers $S_1, \ldots, S_k$ to Q where S_i is a subgraph of G.

Following this scenario, we study the complexity of the approaches presented above. Such complexity is evaluated in terms of the number of basic operations to compute in the worst case. Due to space constraints, we do not report the algorithms of the approaches but during the analysis we make exact reference to the lines of corresponding pseudo-codes in the respective works [6, 9, 4].

Let us introduce the notation we use:

- N: number of solutions.
- $|Q|$: length of Q (i.e. number of keywords $q_i \in Q$).
- $|G|$: number of nodes in the graph G.

3.1 BLINKS

In this section we show that the complexity of `BLINKS` is $O(\texttt{BLINKS}) \in O(|G|^2 \times |Q|^2)$. With respect to the previous notation, we have to introduce the following terms:

- K: number of matching elements in `BLINKS`.
- B: number of blocks in the `BLINKS` partition.
- P: number of portals p_i.
- R: number of roots in the graph G.
- $|A|$: number of current determined solutions
- $|RC|$: number of candidates to be solution

[1] `http://lucene.apache.org/java/docs/`

The search algorithm of BLINKS (i.e. searchBLINKS) calls a main procedure visitNode that is supported by the functions sumDist and sumLBDist. For a node u, just visited but not yet determined as the root of a solution, these functions are used, respectively, to calculate the distance and the lower bound distances from u to individual keywords. Therefore:

$$\texttt{sumDist}(u) = \sum_{i=0}^{|Q|} Dist_i(u) \in O(|Q|)$$

$$\texttt{sumLBDist}(u) = \sum_{i=0}^{|Q|} LBDist_i(u) \in O(|Q|)$$

About visitNode we can say
$lines[21-27]$: $O(|Q|)$ because it computes two hash table accesses and one update $\forall q_i \in Q$.
$lines[28-29] : O(\texttt{sumLBDist}) \in O(|Q|)$.
$lines[30-31] : O(1)$.
$lines[32-34] : O(\texttt{sumDist}) \in O(|Q|)$.

Since those lines are in an if-then-else instruction, we have $O(\texttt{visitNode}) \in O(|Q| + |Q|) = O(|Q|)$.

For $lines[2-5]$ we have $O(|Q| \times B)$ because in $lines[2-5]$ the algorithm creates $|Q|$ queues and for each of them it inserts $O(B)$ times a cursor in it. But if one consider that there is only one queue insertion for each matching element, then we have $O(|Q|+K)$. Since $|Q| \leq K$, then we conclude $O(K)$ in $lines[2-5]$.

In $lines[6-17]$ we need to distinguish between portals p_i and normal nodes u_i.

- Nodes p_i:

  $line[10] : O(\texttt{visitNode}) \in O(|Q|)$.
  $lines[11-14] : O(B)$.
  $line[16] : O(|RC| - |A|) \times O(sumLBDlist)$.

  Since $O(|RC|) \in O(|G|)$, $O(|A|) \in O(N)$, $O(\texttt{sumLBDlist}) \in O(|Q|)$, and $|G| \gg N$, then in $line[16]$ we have $O(|G| \times |Q|)$.

  We remind the so-called Lemma 1 of BLINKS: "*the number of cursors opened by searchBLINKS for each query keyword is O(P), where P is the number of portals in the partitioning of the data graph*". Hence, considering $line[11]$ and Lemma 1, iterations on $lines[6-17]$ are at most $|Q| \times P$, i.e. a portal p_i cannot be traversed more than once for each keyword q_i
  .

- Nodes u_i:

  $line[10] : O(\texttt{visitNode}) \in O(|Q|)$.
  $line[11-14] : O(1)$ because $LPB(u) = \emptyset$ in the worst case.
  $line[16] : O(|G| \times |Q|)$ as showed above.

  Iterations on these nodes occur $(|G| \times |Q|) - (|Q| \times P)$, because the total number of iterations is less than the number of iterations for the portals.

To sum up, the global complexity results as follows:

$lines[2-5] : O(K)$
$lines[6-17] : O((|Q| \times P) \times (|Q| + B + (|G| \times |Q|)))$, for the iterations on portals p_i
$lines[6-17] : O((|G| \times |Q| - |Q| \times P) \times (|G| \times |Q|))$, for the iterations on nodes u_i

Considering that $|G| > 1, |G| \geq B, |Q| \geq 1$ we have:

$lines[6-17] : O((|Q| \times P) \times O(|G| \times |Q|))$, for the iterations on portals p_i
$lines[6-17] : O((|G| \times |Q| - |Q| \times P) \times O(|G| \times |Q|))$, for the iterations on nodes u_i

Furthermore, because $K \leq |G|$:

$$O(\texttt{BLINKS}) \in O(K + |G|^2 \times |Q|^2 - |Q|^2 \times |G| \times P + |Q|^2 \times |G| \times P)$$

$$\in O(|G|^2 \times |Q|^2).$$

The complexity result we obtain for BLINKS is aligned with the experimental results authors present in [6]. There is no dependence from the number of keyword matching elements. However BLINKS shows longer response time for queries with more keywords. This is a strong proof since $|Q|$ is much smaller than K. Moreover the computing time is not affected by the block size of the partitioning. Authors state that query response times longer than 90 seconds are trunked. This means that in some case the algorithm down-performs the average behavior. We believe that the problem relies in aiming at finding the exact matching sub-graph which tend easily to fall in the worst case scenario. In fact if there is no matching keyword node for at least one keyword the algorithm cannot return an answer due to the condition in $line[16]$, and neither prune the search in $line[28]$. It just can return for the condition into the `while` instruction that means exploring all the search space.

3.2 SearchWebDB

The complexity of SearchWebDB should consider the augmentation of the summary graph (AUGMENTATION) and the Top-N query computation (SEARCH). We have to introduce the following terms:

- k_i: number of elements matching the keyword q_i.
- K: number of matching elements.
- S: number of augmented summary graph nodes.

We show that such complexity is $(O(\texttt{AUGMENTATION}) + O(\texttt{SEARCH})) \in O(|G| \times K^2)$. In this approach, we should consider S as the number of nodes in the exploration, i.e. the number of augmented summary graph nodes we use for top-N query retrieval. Moreover, because we are analyzing the worst case, we consider $O(S) \in O(|G|)$.

$O(\texttt{AUGMENTATION}) \in O(K)$ because we have to insert K elements to generate the summary graph.

The procedure SEARCH is supported by the function TOP-N for the query computation. Referring to this function, we say

- In $lines[1-7]$ `TOP-N` possibly completes a sub-graph by merging the paths from the given connecting element and to some keyword element. In the algorithm a path can be computed efficiently due to the presence of a cursor that keeps track of his parent cursor, which, recursively, defines a path up to some element matching a keyword. For each node K cursors can be generated at most. Then we have $O(K)$.

- In $lines[11-16]$, we have $O(N)$, where we can ignore function $maptoQuery$ having constant complexity.

Since $lines[11-16]$ are executed only if the top-N solutions are found (i.e. only once in the execution), we do not include the complexity of this block in the study. Therefore $O(\texttt{TOP}-\texttt{N}) \in O(K)$.

Going back to the procedure `SEARCH`, in $lines[1-6]$ we have $O(|Q| \times max\{k_1, k_2, \ldots, k_{|Q|}\})$ because $\forall q_i \in Q$ the algorithm computes k_i basic inserts. However a cursor is inserted for each matching element, then the complexity in $lines[1-6]$ is $O(K)$. Analyzing $lines[7-27]$, in the worst case the condition of $line[10]$ fails at all iterations. This makes ineffective the heuristic $minCostCursor$ at $line[8]$, implying that all the cursors generated by the algorithm have to be extracted later from their respective queues. This means that iterations in $lines[7-27]$ will be computed once for each generated cursor, where the total number of generated cursors is $|G| \times K$ (since $lines[1-6]$ generated K cursors). Therefore in $lines[7-27]$ we have $O(|G| \times K) \times O(TOP-N) \in O(|G| \times K^2)$.

If we consider $K \geq 1$, the global complexity of the algorithm is:

$$O(\texttt{SearchWebDB}) \in O(\texttt{AUGMENTATION}) + O(\texttt{SEARCH})$$

$$\in O(K) + O(|G| \times K^2) \in O(K + |G| \times K^2)$$

$$\in O(|G| \times K^2)$$

As for `BLINKS`, complexity of `SearchWebDB` is aligned with the experiments presented in [9]. The complexity of `SearchWebDB` is independent from the query length. However authors noticed better performance when the number of keyword is large and it can correspond averagely to a large K. Additionally authors say that there is a linear dependence in N that we did not demonstrate because we analyzed the worst case which did not take into account pruning conditions. In fact we considered $|G|$ instead of S, which is supposed to be much smaller than $|G|$. On the other hand, we noticed that query processing time increases when $|Q|$ changes for a given $N > 3$. In particular in top-20 cases the query processing time does not increase linearly in the query length, but is (almost) quadratic.

3.3 Yanii

In this section we discuss the complexity of `Yanii`. We introduce the following terms

- C: number of clusters $c_i \in$ CL.

- T: number of paths $p_i \in$ PT.

We show that $O(\texttt{Yanii}) \in O(C \times T)$. This result is the sum of complexities in three subphases of the on-line computation: $O(\texttt{QUERYSUBMISSION}) + O(\texttt{CLUSTERING}) + O(\texttt{BUILDING})$.

Algorithm 1: Clustering of Informative Paths

Input : An List PT of informative paths, a query Q
Output: A Priority Queue CL of clusters

```
1  CL' ← CreateSet() ;
2  PT' ← subsumedDelete(PT);
3  while PT' is not empty do
4      PT' - {pt};
5      if ∃Cl_i ∈ CL' : pt ≈ t_{Cl_i} then
6          Enqueue(pt, Score(pt,Q), Cl_i);
7          UpdateScore(Cl_i);
8      else
9          Cl_i ← CreateCluster(pt);
10         Enqueue(pt, Score(pt,Q), Cl_i);
11         CL' ∪ {Cl_i};
12 CL ← OrderClusters(CL');
13 return CL;
```

Since there is direct access to each keyword through the index, $O(\texttt{QUERYSUBMISSION}) \in O(|Q|)$.

With respect to [4] we improved the clustering phase as shown in the Algorithm 1.
In $lines[5-7]$ the algorithm checks if there exists a cluster Cl_i matching the path template. Hence we have $O(C)$. In $lines[8-11]$ we have $O(1)$. In $lines[3-11]$ the iteration is computed for each path (i.e T times). As a consequence, the complexity in $lines[3-11]$ is $T \times O(C) \in O(T \times C)$. In $line[12]$ we obtain $O(C \times log_2 C)$ to order the queue. Finally $O(\texttt{CLUSTERING}) \in O(T \times C) + O(C \times \log_2 C)$. Knowing that $C, T > 1$ and $T > C$ then $O(\texttt{CLUSTERING}) \in O(T \times C)$.

Referring to BUILDING, we have

$lines[4-6] : O(1)$.
$line[8] : O(T)$.
$lines[12-15] : O(T)$
$lines[16-17] : O(T)$
$lines[18-21] : O(1)$

In $lines[9-22]$ we consider the complexity of $lines[12-15]$ and the maximum complexity between executions in $lines[16-17]$ and $lines[18-21]$. Since we have C iterations, in $lines[9-22]$ we have $C \times (O(T) + O(T))$. Notice that in $lines[12-15]$ and $lines[16-17]$ we iterate all the paths at most. Hence we have $O(T)$ and $O(T)$, and, consequently, in $lines[9-22]$ the complexity is $O(T) + O(T) \in O(T)$. In $lines[24-26]$ we have $O(C)$ because we just iterate on the visited cluster list. Since $\forall Cl : \exists pt \in PT \Rightarrow T \geq C$, final complexity becomes

$$O(\texttt{BUILDING}) \in N \times (O(1) + O(T) + O(T) + O(C))$$

$$\in N \times O(T) \in O(N \times T)$$

case	BLINKS	SearchWebDB	Yanii
(1). $T = R \times K$	$O(\|Q\|^2 \times \|G\|^2)$	$O(K^2 \times \|G\|)$	$O(R \times K \times C)$
(2). $T = K$	$O(\|Q\|^2 \times \|G\|^2)$	$O(K^2 \times \|G\|)$	$O(K \times C)$
(3). $C = K$	$O(\|Q\|^2 \times \|G\|^2)$	$O(K^2 \times \|G\|)$	$O(T \times K)$
(4). $K^2 = \|Q\|^2, T = R \times K$	$O(K^2 \times \|G\|^2)$	$O(K^2 \times \|G\|)$	$O(R \times K \times C)$
(5). $T = K, C = K, \|Q\| = K$	$O(\|G\|^2 \times K^2)$	$O(\|G\| \times K^2)$	$O(K^2)$
(6). $T = \frac{\|G\|}{K}, \|Q\| = K, C = K$	$O(\|Q\|^2 \times \|G\|^2)$	$O(\|Q\|^2 \times \|G\|)$	$O(K \times \frac{\|G\|}{K}) = O(\|G\|)$

Table 1: Comparison with relations

We can conclude that the overall complexity of Yanii is $O(\texttt{QUERYSUBMISSION}) + O(\texttt{CLUSTERING}) + O(\texttt{BUILDING}) \in O(|Q|) + O(T \times C) + O(N \times T) \in max\{O(T \times C), O(N \times T)\}$. Given that most of times $C > N$, we result the final complexity:

$$O(\texttt{Yanii}) \in O(C \times T).$$

4 Comparison

We now compare the results obtained in the previous section. Recall that:

$$O(\texttt{BLINKS}) \in O(|Q|^2 \times |G|^2)$$

$$O(\texttt{SearchWebDB}) \in O(|G| \times K^2)$$

$$O(\texttt{Yanii}) \in O(C \times T)$$

Comparison can be performed in two ways: (i) by expressing relations between the terms of all complexities, or (ii) by a pairwise comparison. We present both the manners.

Comparison with relations in this case we have to define some *relations* between the different terms used in each study. In Table 1 we show such relations between the terms we used in our study and provide the resulting complexities. We now explain each single relation taking into account that we combined them in some case.

- $T = R \times K$: it means that we can reach a keyword matching node from all the roots with different paths. It is an average/worst case for Yanii because we can have more than one path starting from a given root to a keyword matching node but even no paths connecting a root to a keyword matching node.
- $T = K$: it means that each path contains a different keyword matching node. It is a good case for Yanii since usually there are more paths sharing the same keyword.
- $C = K$: it means that we have a different cluster for each keyword matching element. In this case we are dependent on data graph structure even if usually $C < K$.
- $K^2 = |Q|^2, |Q| = K$: they mean that we have one keyword matching element for each keyword q_i. SearchWebDB take advantages against BLINKS with these relations since usually $|Q| < K$.

- $T = \frac{|G|}{K}$: it means that all the nodes in the data graph are involved in a path and one node cannot belong to more than one path. This is a very extreme case.

Using these relations we produce six different cases, as shown in Table 1. In particular the cases (5) and (6) result from the combination of the previous ones. Such cases clearly show the complexity ranges of each approach. Using an approximation measure d of complexity we can say:

- $O(\texttt{BLINKS}) \in [O(d^3), O(d^4)]$
- $O(\texttt{SearchWebDB}) \in [O(d^2), O(d^3)]$
- $O(\texttt{Yanii}) \in [O(d), O(d^2)]$

This is a relevant result. All the presented algorithms are promising representative of an efficient solution for keyword based search in PTIME class complexity. However we demonstrated how Yanii is more efficient with respect to the others, presenting a quadratic complexity as upper-bound.

Pairwise Comparison we demonstrate that:

$$O(\texttt{BLINKS}) > O(\texttt{SearchWebDB}) > O(\texttt{Yanii})$$

By comparing the complexity result from previous sections, for the algorithms we have:
BLINKS vs SearchWebDB:

$$\frac{O(|G| \times K^2)}{O(|Q|^2 \times |G|^2)} = \frac{O(K \times K)}{O(|Q| \times |Q| \times |G|)}$$

In a very extreme case we have that each node of the graph matches the query (i.e. all keywords). Therefore:

$$O(K \times K) \approx O(|Q| \times |Q| \times |G|)$$

From this result, we infer that in a real case:

$$O(K^2) \ll O(|Q|^2 \times |G|) \Longrightarrow O(\texttt{BLINKS}) > O(\texttt{SearchWebDB})$$

SearchWebDB vs Yanii:

In this case we consider again S instead of $|G|$ to be more effective. Therefore we use $O(C \times T)$ and $O(S \times K^2)$. We separately compare $O(T)$ against $O(S \times K)$ and $O(C)$ against $O(K)$. First of all we can say $C \leq K$ because in the worst case each matching keyword element corresponds to a path having a different template (i.e. a different cluster). Furthermore we can consider $O(C) \in O(K)$ that allows to focus the comparison to $O(T)$ against $O(S \times K)$. In fact, $C < T$ and $K < K \times S^2$. In an extreme case we have that for each node v_c of the summary graph (sg) and for each matching element v_k we have an informative path pt such that pt starts with v_c (i.e. the root of the path) and ends in v_k (i.e. the last node of the path), where there doesn't exist an informative path pt', ending into v_k (we denote with $last(pt')$), that subsumes pt (i.e. $pt \lhd pt'$). In other words

$$\forall v_c \in sg, \forall v_k$$

$$\exists pt \in \text{PT} : pt = v_c - \ldots - v_k$$
$$\text{and } \nexists pt' \in \text{PT} : pt \lhd pt', last(pt') = v_k$$

In this case we have

$$\frac{O(T)}{O(S \times K)} = 1$$

Hence, we infer that in a real case:

$$O(T) \ll O(S \times K) \Longrightarrow O(\texttt{SearchWebDB}) > O(\texttt{Yanii})$$

`BLINKS` vs `Yanii`:

From the previous results $O(\texttt{SearchWebDB}) > O(\texttt{Yanii})$ and $O(\texttt{BLINKS}) > O(\texttt{SearchWebDB})$. Hence we deduce $O(\texttt{BLINKS}) > O(\texttt{Yanii})$.

Given the above results, we can make the following call:

$$O(\texttt{BLINKS}) > O(\texttt{SearchWebDB}) > O(\texttt{Yanii})$$

5 Conclusion and Future Work

We have precisely described the computational complexity of three representative approaches to keyword based search over graph structures. We have defined which are the measures of reference for a complexity comparison. The final result we have obtained is that `Yanii` shows the lowest computational cost among the approaches considered, which is an indication of the potentiality behind the intuition in `Yanii`. A lower complexity means that the approach is less susceptible to variations of the keyword search setting. From a theoretical point of view, future directions focus on improving the search algorithm of `Yanii` to reach a linear time complexity.

References

[1] S. Agrawal, S. Chaudhuri, G. Das. Dbxplorer: enabling keyword search over relational databases. In *Int. Conf. on Management of Data (SIGMOD'02), USA*, 2002.

[2] G. Bhalotia, Hulgeri, A., Nakhe, C., Chakrabarti, S., Sudarshan, S. Keyword searching and browsing in databases using banks. In *Int. Conf. on Data Engineering (ICDE'02)*, 2002.

[3] S. Cohen, J. Mamou, Kanza, Y., Sagiv, Y. Xsearch: A semantic search engine for xml. In *Int. Conf. on Very Large DataBase (VLDB'03), Germany*, 2003.

[4] R. De Virgilio, P. Cappellari, M. Miscione. Cluster-based exploration for Effective Keyword Search over Semantic Datasets. In *Proc. of the 28th International Conference on Conceptual Modeling (ER'09)*, 2009.

[5] L. Guo, Shao, F., Botev, C., Shanmugasundaram, J. Xrank: Ranked keyword search over xml documents. In *Int. Conf. on Management of Data (SIGMOD'03), USA*, 2003.

[6] H. He, H. Wang, J. Yang, P.S. Yu. Blinks: ranked keyword searches on graphs. In *Proc. of the International Conference on Management of Data (SIGMOD'07), China*, 2007.

[7] V. Hristidis, Papakonstantinou, Y. Discover: Keyword search in relational databases. In *Int. Conf. on Very Large DataBase (VLDB'02), China*, 2002.

[8] Kaushik, R., Krishnamurthy, R., Naughton, J.F., Ramakrishnan, R. On the integration of structure indexes and inverted lists. In *Int. Conf. on Management of Data (SIGMOD'04), France*, 2004.

[9] T. Tran, H. Wang, S. Rudolph, P. Cimiano. Top-k exploration of query graph candidates for efficient keyword search on rdf. In *Proc. of the International Conference on Data Engineering (ICDE'09), China*, 2009.