Leveraging efficient indexing schema to support multigraph query answering

Vijay Ingalalli [1,2], **Dino Ienco** [1,2], **Pascal Poncelet** [1,2]

1. LIRMM - Montpellier, France
{vijay,pascal.poncelet}@lirmm.fr

2. IRSTEA - Montpellier, France
dino.ienco@irstea.fr

ABSTRACT. *Many real world datasets can be represented by graphs with a set of nodes interconnected with each other by multiple relations (e.g., social network, RDF graph, biological data). Such a rich graph, called multigraph, is well suited to represent real world scenarios with complex interactions. However, performing subgraph query on multigraphs is still an open issue since, unfortunately, all the existing algorithms for subgraph query matching are not able to adequately leverage the multiple relationships that exist between the nodes. Motivated by the lack of approaches for sub-multigraph query and stimulated by the increasing number of datasets that can be modelled as multigraphs, in this paper we propose* IMQA *(Index based Multigraph Query Answering), a novel algorithm to extract all the embeddings of a sub-multigraph query from a single large multigraph.* IMQA *is composed of two main phases: Firstly, it implements a novel indexing schema for multiple edges, which will help to efficiently retrieve the vertices of the multigraph that match the query vertices. Secondly, it performs an efficient subgraph search to output the entire set of embeddings for the given query. Extensive experiments conducted on real datasets prove the time efficiency as well as the scalability of* IMQA.

RÉSUMÉ. *De nombreuses données réelles peuvent être représentées par un réseau avec un ensemble de nœuds interconnectés via différentes relations (i.e. les réseaux sociaux, les données biologiques, les graphes RDF). Ce type de graphe, appelé multigraphe, est tout à fait adapté à la représentation de scénarios réels contenant des interactions complexes. La recherche de sous-multigraphe dans des multigraphes est un domaine de recherche ouvert et malheureusement les algorithmes existants pour faire de la recherche de sous-graphe ne sont pas adaptés et ne peuvent pas prendre en compte les différentes relations qui peuvent exister entre les nœuds. Motivés par le manque d'approches existantes et par le nombre croissant d'applications qui peuvent être modélisées via des multigraphes, nous proposons dans cet article* IMQA *un nouvel algorithme pour extraire tous les sous-multigraphes inclus dans un grand multigraphe.* IMQA *comporte deux étapes principales. Tout d'abord il implémente une nouvelle structure d'indexation pour les relations multiples qui est utilisée pour rechercher efficacement les sommets du multigraphe qui correspondent aux sommets de la requête. Ensuite, il réalise une recherche efficace de* *l'ensemble des sous-multigraphes correspondant à une requête donnée. Les nombreuses expérimentations menées sur des jeux de données réelles ont montré l'efficacité et le passage à l'échelle de* IMQA.

KEYWORDS: *multigraph query, indexing, subgraph query matching.*

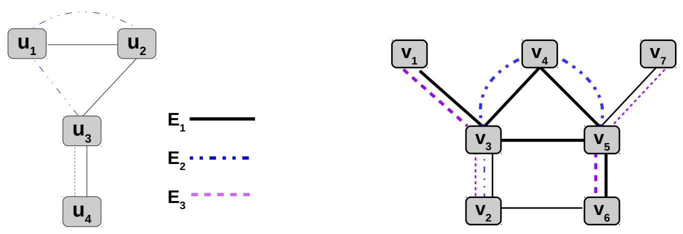
MOTS-CLÉS : *requête multigraphes, index, recherche de sous-multigraphes.*

DOI:10.3166/ISI.21.3.53-74 © 2016 Lavoisier

1. Introduction

Much of the real world data can be represented by a graph with a set of nodes interconnected with each other by multiple relations. Such a rich graph is called multigraph which allows different types of edges in order to represent different types of relations between vertices (Boden *et al.*, 2012; Bonchi *et al.*, 2014). Examples of multigraphs are: social networks spanning over the same set of people, but with different life aspects (e.g. social relationships such as Facebook, Twitter, LinkedIn, etc.); protein-protein interaction multigraphs created considering the pairs of proteins that have direct interaction/physical association or they are co-localised (Zhang, 2009); gene multigraphs, where genes are connected by considering the different pathway interactions belonging to different pathways; RDF knowledge graph where the same subject/object node pair is connected by different predicates (Libkin *et al.*, 2013).

One of the most crucial and difficult operation in graph data management is subgraph querying (Han *et al.*, 2013). The subgraph query problem belongs to NP-complete class (Han *et al.*, 2013) but, practically, we can find embeddings in real graph data by exploiting better matching order and intelligent pruning rules. In literature, different families of subgraph matching algorithms exist. A first group of techniques employ *feature based indexing* followed by a filtering and verification framework (Yan *et al.*, 2004; Cheng *et al.*, 2007; X. Zhao *et al.*, 2013; Lin, Bei, 2014). All these methods are developed for transactional graphs, i.e. the database is composed of a collection of graphs and each graph can be seen as a transaction of such database, and they cannot be trivially extended to the single multigraph scenario. A second family of approaches avoid indexing and it uses *backtracking algorithms* to find embeddings by growing the partial solutions. In the beginning, they obtain a potential set of candidate vertices for every vertex in the query graph. Then a recursive subroutine called SUBGRAPHSEARCH is invoked to find all the possible embeddings of the query graph in the data graph (Cordella *et al.*, 2004; Shang *et al.*, 2008; He, Singh, 2008). All these approaches are able to manage only graph with a single label on the vertex. Although index based approaches focus on transactional database graphs, some backtracking algorithms address the large single graph setting (Lee *et al.*, 2012). All these methods are not conceived to manage and query multigraphs and their extension to manage multiple relations between nodes cannot be trivial. A third and more recent family of techniques defines *Equivalent Classes* at query and/or database level, by exploiting vertex relationships. Once the data vertices are grouped into equivalence



(a) Query sub-multigraph S (b) Data multigraph G

Figure 1. A sample query and data multigraph

classes the search space is reduced and the whole process is speeded up (Han *et al.*, 2013; Ren, Wang, 2015). Adapting these methods to multigraph is not straightforward since, the different types of relationships between vertices can exponentially increase the number of equivalent classes (for both query and data graph) thereby drastically reducing the efficiency of these strategies. Among the vast literature on subgraph isomorphism, (Bonnici *et al.*, 2013) is the unique approach that is able to directly manage graph with (multiple) labels on the edges. It proposes an approach called RI that uses light pruning rules in order to avoid visiting useless candidates.

Due to the abundance of multigraph data and the importance of querying the multigraph data, in this paper, we propose a novel method IMQA that supports subgraph matching in a Multigraph via efficient indexing. Unlike the previous proposed approaches, we conceive an indexing schema to summarize information contained in a single large multigraph. IMQA involves two main phases: (i) an off-line phase that builds efficient indexes for the information contained in the multigraph; (ii) an on-line phase, where a sub-multigraph search procedure exploits the indexing schema previously built. The rest of the paper is organized as follows. Background and problem definition are provided in Section 2. An overview of the proposed approach is presented in Section 3, while Section 4 and Section 5 describe the indexing schema and the query subgraph search algorithm, respectively. Section 6 presents experimental results. Conclusions are drawn in Section 7.

2. Background

Formally, we define a multigraph G as a tuple of four elements (V, E, L_E, D) where V is the set of vertices and D is the set of dimensions , $E \subseteq V \times V$ is the set of undirected edges and $L_E : V \times V \rightarrow 2^D$ is a labelling function that assigns the subset of dimensions to each edge it belongs to. In this paper, we address the sub-multigraph problem for undirected and unattributed multigraphs.

DEFINITION 1. — Subgraph isomorphism for a multigraph. *Given a sub multigraph $S = (V^s, E^s, L^s_E, D^s)$ and a multigraph $G = (V, E, L_E, D)$, the subgraph isomorphism from S to G is an injective function $\psi : V^s \rightarrow V$ such that:*

$\forall (u_m, u_n) \in E^s$, $\exists\, (\psi(u_m), \psi(u_n)) \in E$ *and* $L_E^s(u_m, u_n) \subseteq L_E(\psi(u_m), \psi(u_n))$.

Problem Definition. Given a query multigraph S and a data multigraph G, the subgraph query problem is to enumerate the distinct embeddings of S in G.

For the ease of representation, in the rest of the paper, we simply refer to a data multigraph G as a *graph*, and a query multigraph S as a *subgraph*. We also enumerate (for unique identification) the set of query vertices by U and the set of data vertices by V. In Figure 1, we introduce a query multigraph S and a data multigraph G. The two valid embeddings for the subgraph S are marked by the thick lines in the graph G and are enumerated as follows: $R_1 := \{[u_1, v_4], [u_2, v_5], [u_3, v_3], [u_4, v_1]\}$; $R_2 := \{[u_1, v_4], [u_2, v_3], [u_3, v_5], [u_4, v_6]\}$, where, each query vertex u_i is matched to a distinct data vertex v_j, written as $[u_i, v_j]$.

3. An Overview of IMQA

In this section, we sketch the main idea of IMQA to address the subgraph query problem for multigraphs. The entire procedure can be divided into two parts: (i) an indexing schema for the graph G that exploits edge dimensions and the vertex neighbourhood structure (Section 4) (ii) a subgraph search algorithm that involves several steps to enumerate the embeddings of the subgraph (Section 5).

The overall idea of IMQA is depicted in Algorithm 1. Initially, we order the set of query vertices U using a heuristic proposed in Section 5.1. With an ordered set of query vertices U^o, we use the indexing schema to find a list of possible candidate matches $C(u_{init})$ only for the initial query vertex u_{init} by calling SELECTCAND (Line 4), as described in Section 5.2. Then, SUBGRAPHSEARCH is recursively called for each candidate solution $v \in C(u_{init})$, to find the matchings in a depth first manner until an embedding is found. The partial embedding is stored in $M = [M_S, M_G]$ - a pair that contains the already matched query vertices M_S and the already matched data vertices M_G. Once the partial embedding grows to become a complete embedding, the repository of embeddings R is updated.

4. Indexing

In this section, we propose the indexing structures that are built on the graph G that are used during the subgraph querying procedure. The primary goal of indexing is to make the query processing time efficient. For a lucid understanding of our indexing schema, we introduce a few definitions.

DEFINITION 2. — Vertex signature. *For a vertex v, the vertex signature $\sigma(v)$ is multiset containing all the multiedges that are incident on v, where any multiedge between v and a neighbouring vertex v' is represented by a set that corresponds to edge dimensions. Formally, $\sigma(v) = \bigcup_{v' \in N(v)} L_E(v, v')$ where $N(v)$ is the set of neighbourhood vertices of v, and $\cup$ is the union operator for multiset.*

Algorithm 1: IMQA

```
1  INPUT: subgraph $S$, graph $G$, indexes $\mathcal{T}, \mathcal{N}$ of $G$
2  OUTPUT: $R$: all the embeddings of $S$ in $G$
3  $U^o$ = ORDERQUERYVERTICES($S$)
4  $C(u_{init})$ = SELECTCAND($u_{init}$, $\mathcal{T}$) /* Ordered cand. vertices */
5  $R = \emptyset$                                  /* Embeddings of S in G */
6  for each $v_{init} \in C(u_{init})$ do
7      $M_S = u_{init}$;                /* Matched initial query vertex */
8      $M_G = v_{init}$;                /* Matched possible data vertex */
9      $M = [M_S, M_G]$                 /* Partial matching of $S$ in $G$ */
10     UPDATE: $R :=$ SUBGRAPHSSEARCH($R, M, \mathcal{N}, S, G, U^o$)
11 return $R$
```

For instance, in Figure 1, $\sigma(v_6) = \{\{E_1, E_3\}, \{E_1\}\}$. The vertex signature is an intermediary representation that is exploited by our indexing schema. All the vertex signatures of the vertices of the graph in Figure 1 are depicted in Table 1.

Table 1. Vertex signatures for the graph in Figure 1b

v_i	$\sigma(v)$
v_1	$\{\{E_1, E_3\}\}$
v_2	$\{\{E_2, E_3, E_1\}, \{E_1\}\}$
v_3	$\{\{E_2, E_3, E_1\}, \{E_1, E_3\}, \{E_1, E_2\}, \{E_1\}\}$
v_4	$\{\{E_1, E_2\}, \{E_1, E_2\}\}$
v_5	$\{\{E_1, E_3\}, \{E_1, E_3\}, \{E_1, E_2\}, \{E_1\}\}$
v_6	$\{\{E_1, E_3\}, \{E_1\}\}$
v_7	$\{\{E_1, E_3\}\}$

The goal of constructing indexing structures is to find the *possible candidate set* for the set of query vertices u, thereby reducing the search space for the SUBGRAPH-SEARCH procedure, making IMQA time efficient.

DEFINITION 3. — Candidate set. *For a query vertex u, the candidate set $C(u)$ is defined as $C(u) = \{v \in G | \sigma(u) \subseteq \sigma(v)\}$.*

In this light, we propose two indexing structures that are built offline: (i) given the vertex signature of all the vertices of graph G, we construct a vertex signature index $\mathcal{T}$ by exploring a set of features f of the signature $\sigma(v)$ (ii) we build a vertex neighbourhood index $\mathcal{N}$ for every vertex in the graph G. The index $\mathcal{T}$ is used to select possible candidates for the initial query vertex in the SELECTCAND procedure while the index $\mathcal{N}$ is used to choose the possible candidates for the rest of the query vertices during the SUBGRAPHSEARCH procedure.

4.1. Vertex Signature Index $\mathcal{T}$

This index is constructed to enumerate the possible candidate set only for the initial query vertex. Since we cannot exploit any structural information for the initial query vertex, $\mathcal{T}$ captures the edge dimension information from the data vertices, so that the non suitable candidates can be pruned away.

We construct the index $\mathcal{T}$ by organizing the information supplied by the vertex signature of the graph; i.e., observing the vertex signature of data vertices, we intend to extract some interesting features. For example, the vertex signature of v_6, $\sigma(v_6) = \{\{E_1, E_3\}, \{E_1\}\}$ has two sets of dimensions in it and hence v_6 is eligible to be matched with query vertices that have at most two sets of items in their signature. Also, $\sigma(v_2) = \{\{E_2, E_3, E_1\}, \{E_1\}\}$ has the edge dimension set of maximum size 3 and hence a query vertex must have the edge dimension set size of at most 3. More such features (e.g., the number of unique dimensions, the total number of occurrences of dimensions, etc.) can be proposed to filter out irrelevant candidate vertices. In particular, for each vertex v, we propose to extract a set of characteristics summarizing useful features of the neighbourhood of a vertex. Those features constitute a *synopses* representation (surrogate) of the original vertex signature.

In this light, we propose six $|f| = 6$ useful features that will be illustrated with the help of the vertex signature $\sigma(v_3) = \{\{E_1, E_2, E_3\}, \{E_1, E_3\}, \{E_1, E_2\}, \{E_1\}\}$:

f_1 Cardinality of vertex signature, $(f_1(v_3) = 4)$

f_2 The number of unique dimensions in the vertex signature, $(f_2(v_3) = 3)$

f_3 The number of all occurrences of the dimensions (with repetition), $(f_3(v_3) = 8)$

f_4 Minimum index of lexicographically ordered edge dimensions, $(f_4(v_3) = 1)$

f_5 Maximum index of lexicographically ordered edge dimensions, $(f_5(v_3) = 3)$

f_6 Maximum cardinality of the vertex sub-signature, $(f_6(v_3) = 3)$

Table 2. Synopses for all the data vertices in Figure1b

Data vertex	Synopses					
v	f_1	f_2	f_3	f_4	f_5	f_6
v_1	1	2	2	1	3	2
v_2	2	3	4	1	3	3
v_3	4	3	8	1	3	3
v_4	2	2	4	1	2	2
v_5	4	3	7	1	3	2
v_6	2	2	3	1	3	2
v_7	1	2	2	1	3	2

In Table 2 we list the synopses for each data vertex shown in Figure 1b, for clear understanding.

By exploiting the aforementioned features, we build the synopses to represent the vertices in an efficient manner that will help us to select the eligible candidates during query processing.

Once the synopsis representation for each data vertex is computed, we store the synopses in an efficient data structure. Since each vertex is represented by a synopsis of several fields, a data structure that helps in efficiently performing range search for multiple elements would be an ideal choice. For this reason, we build a $|f|$-dimensional R-tree, whose nodes are the synopses having $|f|$ fields.

The general idea of using an R-tree structure is as follows: A synopses $F = \{f_1, \ldots, f_{|f|}\}$ of a data vertex spans an axes-parallel rectangle in an f-dimensional space, where the maximum co-ordinates of the rectangle are the values of the synopses fields $(f_1, \ldots, f_{|f|})$, and the minimum co-ordinates are the origin of the rectangle (filled with zero values). For example, a data vertex represented by the synopses with two features $F_v = (2, 3)$ spans a rectangle in a 2-dimensional space in the interval range $([0, 2], [0, 3])$. Now if we consider synopses of two query vertices, $F_{u_1} = (1, 3)$ and $F_{u_2} = (1, 4)$, we observe that the rectangle spanned by F_{u_1} is wholly contained in the rectangle spanned by F_v but F_{u_2} is not wholly contained in F_v. Formally, the possible candidates for vertex u can be written as $\mathcal{P}(u) = \{v|\forall_{i\in[1,\ldots,f]} F_{u(i)} \leq F_{v(i)}\}$, where the constraints are met for all the $|f|$-dimensions. Since we apply the same inequality constraint to all the fields, we need to pre-process few synopses fields; i.e., the field f_4 contains the minimum value of the index, and hence we negate f_4 so that the rectangular containment problem still holds good. Thus, we keep on inserting the synopses representations of each data vertex v into the R-tree and build the index $\mathcal{T}$, where each synopses is treated as an $|f|$-dimensional node of the R-tree.

4.2. Vertex Neighbourhood Index $\mathcal{N}$

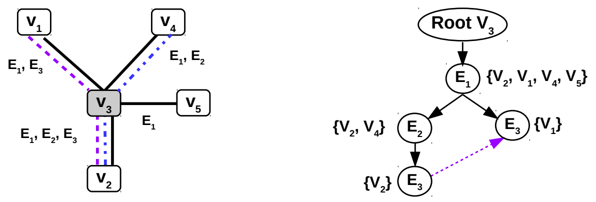
The aim of this indexing structure is to find the possible candidates for the rest of the query vertices.

Since the previous indexing schema enables us to select the possible candidate set for the initial query vertex, we propose an index structure to obtain the possible candidate set for the subsequent query vertices. The index $\mathcal{N}$ will help us to find the possible candidate set for a query vertex u during the SUBGRAPHSEARCH procedure by retaining the structural connectivity with the previously matched candidate vertices, while discovering the embeddings of the subgraph S in the graph G.

The index $\mathcal{N}$ comprises of neighbourhood trees built for each of the data vertex v. To understand the index structure, let us consider the data vertex v_3 from Figure 1b, shown separately in Figure 2a. For this vertex v_3, we collect all the neighbourhood information (vertices and multiedges), and represent this information by a tree structure. Thus, the tree representation of a vertex v contains the neighbourhood vertices and their corresponding multiedges, as shown in Figure 2b, where the nodes of the tree structure are represented by the edge dimensions.

In order to construct an efficient tree structure, we take inspiration from (Terrovitis *et al.*, 2006) to propose the structure - Ordered Trie with Inverted List (OTIL). Consider a data vertex v_i, with a set of n neighbourhood vertices $N(v_i)$. Now, for every pair $(v_i, N^j(v_i))$, where $j \in \{1, \ldots, n\}$, there exists a multiedge (set of edge dimensions) $\{E_1, \ldots, E_d\}$, which is inserted into the OTIL structure. Each multiedge is ordered (with the increasing edge dimensions), before inserting into OTIL structure, and the order is universally maintained for both query and data vertices. Further, for every edge dimension E_i that is inserted into the OTIL, we maintain an *inverted list* that contains all the neighbourhood vertices $N(v_i)$, that have the edge dimension E_i incident on them. For example, as shown in Figure 2b, the edge E_2 will contain the list $\{v_2, v_4\}$, since E_2 forms an edge between v_3 and both v_2 and v_4.

To construct the OTIL index as shown in Figure 2b, we insert each ordered multiedge that is incident on v at the root of the trie structure. To make index querying more time efficient, the OTIL nodes with identical edge dimension (e.g., E_3) are internally connected and thus form a linked list of data vertices. For example, if we want to query the index in Figure 2b with a vertex having edges $\{E_1, E_3\}$, we do not need to traverse the entire OTIL. Instead, we perform a pre-ordered search, and as soon as we find the first set of matches, which is $\{V_2\}$, we will be redirected to the OTIL node, where we can fetch the matched vertices much faster (in this case $\{V_1\}$), thereby outputting the set of matches as $\{V_2, V_1\}$.



(a) Neighbourhood structure of v_3 (b) OTIL structure for v_3

Figure 2. Building Neighbourhood Index for data vertex v_3

5. Subgraph Query Processing

We now proceed with the subgraph query processing. In order to find the embeddings of a subgraph, we not only need to find the valid candidates for each query vertex, but also retain the structure of the subgraph to be matched. In this section, we discuss in detail about the various procedures involved in Algorithm 1.

5.1. Query Vertex Ordering

Before performing query processing, we order the set of query vertices U into an ordered set of query vertices U^o. It is argued that an effective ordering of the query vertices improves the efficiency of subgraph querying (Lee *et al.*, 2012). In order to achieve this, we propose a heuristic that employs two scoring functions.

The first scoring function relies on the number of multiedges of a query vertex. For each query vertex u_i, the number of multiedges incident on it is assigned as a score; i.e., $r_1(u_i) = \sum_{j=1}^{m} |\sigma(u_i^j)|$, where u_i has m multiedges, $|\sigma(u_i^j)|$ captures the number of edge dimensions in the j^{th} multiedge. Query vertices are ordered in ascending order considering the scoring function r_1, and thus $u_{init} = \text{argmax}(r_1(u_i))$. For example, in Figure 1a, vertex u_3 has the maximum number of edges incident on it, which is 4, and hence is chosen as an initial vertex.

The second scoring function depends on the structure of the subgraph. We maintain an ordered set of query vertices U^o and keep adding the next *eligible* query vertex. In the beginning, only the initial query vertex u_{init} is in U^o. The set of next eligible query vertices U^o_{nbr} are the vertices that are in the 1-neighbourhood of U^o. For each of the next eligible query vertex $u_n \in U^o_{nbr}$, we assign a score depending on a second scoring function defined as $r_2(u_n) = |\{U^o \cap adj(u_n)\}|$. It considers the number of the adjacent vertices of u_n that are present in the already ordered query vertices U^o.

Then, among the set of next eligible query vertices U^o_{nbr} for the already ordered U^o, we give first priority to function r_2 and the second priority to function r_1. Thus, in case of any tie ups, w.r.t. r_2, the score of r_1 will be considered. When both r_2 and r_1 leave us in a tie up situation, we break such tie at random.

5.2. *Select Candidates for Initial Query Vertex*

For the initial query vertex u_{init}, we exploit the index structure $\mathcal{T}$ to retrieve the set of possible candidate data vertices, thereby pruning the unwanted candidates for the reduction of search space.

THEOREM. — 1. *Querying the vertex signature index $\mathcal{T}$ constructed with synopses, guarantees to output at least the entire set of valid candidate vertices.*

PROOF. — Consider the field f_1 in the synopses that represents the cardinality of the vertex signature. Let $\sigma(u)$ be the signature of the query vertex u and $\{\sigma(v_1), \ldots, \sigma(v_n)\}$ be the set of signatures on the data vertices. By using f_1 we need to show that $C(u)$ has at least all the valid candidates. Since we are looking for a superset of query vertex signature, and we are checking the condition $f_1(u) \leq f_1(v_i)$, where $v_i \in \{v_1, \ldots, v_n\}$, v_i is pruned if it does not match the inequality criterion, since it can never be an eligible candidate. We can extend this analogy to all the synopses fields, since they all can be applied disjunctively. ■

During the SELECTCAND procedure (Algorithm 1, Line 4), we retrieve the possible candidate vertices from the data graph by exploiting the vertex signature index $\mathcal{T}$. However, since querying $\mathcal{T}$ would not prune away all the unwanted vertices for u_{init}, the corresponding partial embeddings would be discarded during the SUBGRAPH-SEARCH procedure. For instance, to find candidate vertices for $u_{init} = u_3$, we build the synopses for u_3 and find the matchable vertices in G using the index $\mathcal{T}$. As we recall, synopses representation of each data vertex spans a rectangle in the d-dimensional space. Thus, it remains to check, if the rectangle spanned by u_3 is contained in any of rectangles spanned by the synopses of the data vertices, with the help of R-tree built on data vertices, which results in the candidate set $\{v_3, v_5\}$.

Once we obtain the candidate vertices for U_{init}, we order the candidate data vertices in the decreasing order of the synopses fields, with decreasing priorities from f_1 to f_6. Thus, if $v_1, \dots, v_c$ compose the ordered set of candidate vertices, the rectangles spanned by the synopses $F(v_1)$, will be of maximum size and that of $F(v_c)$ will be of minimum size.

Algorithm 2: SUBGRAPHSEARCH($R, M, \mathcal{N}, S, G, U^o$)

```
1  FETCH $u_{nxt} \in U^o$        /* Fetch query vertex to be matched */
2  $M_C$ = FINDJOINABLE($M_S, M_G, \mathcal{N}, u_{nxt}$)      /* Matchable candidate
   vertices */
3  if $|M_C| \neq \emptyset$ then
4      for each $v_{nxt} \in M_C$ do
5          $M_S = M_S \cup u_{nxt}$;
6          $M_G = M_G \cup v_{nxt}$;
7          $M = [M_S, M_G]$                /* Partial matching grows */
8          SUBGRAPHSEARCH($R, M, \mathcal{N}, S, G, U^o$)
9          if ($|M|$ == $|U^o|$) then
10             $R = R \cup M$                    /* Embedding found */
11 return R
```

5.3. *Subgraph Searching*

The SUBGRAPHSEARCH recursive procedure is described in Algorithm 2. Once an initial query vertex u_{init} and its possible data vertex $v_{init} \in C_{u_{init}}$, that could be a potential match, is chosen from the set of select candidates, we have the partial solution pair $M = [M_S, M_G]$ of the subgraph query pattern we want to grow. If v_{init} is a right match for u_{init}, and we succeed in finding the subsequent valid matches for U^o, we will obtain an embedding; else, the recursion would revert back and move on to next possible data vertex to look for the embeddings.

In the beginning of SUBGRAPHSEARCH procedure, we fetch the next query vertex u_{nxt} from the set of ordered query vertices U^o, that is to be matched (Line 1). Then FINDJOINABLE procedure finds all the valid data vertices that can be matched with the next query vertex u_{nxt} (Line 2). The main task of subgraph matching is done by the FINDJOINABLE procedure, depicted in Algorithm 3. Once all the valid matches for u_{nxt} are obtained, we update the solution pair $M = [M_S, M_G]$ (Line 5-7). Then we recursively call SUBGRAPHSEARCH procedure until all the vertices in U^o have been matched (Line 8). If we succeed in finding matches for the entire set of query vertices U^o, then we update the repository of embeddings (Line 9-10); else, we keep on looking for matches recursively in the search space, until there are no possible candidates to be matched for u_{nxt} (Line 3).

```
Algorithm 3: FINDJOINABLE($M_S, M_G, \mathcal{N}, u_{nxt}$)
1 $A_S := M_S \cap adj(u_{nxt})$            /* Matched query neighbours */
2 $A_G := \{v | v \in M_G\}$ /* Corresponding matched data neighbours */
3 INTIALIZE: $M_C^{temp} = 0, M_C = 0$
4 $M_C^{temp} = \cap_{i=1}^{|A_S|}$ NEIGHINDEXQUERY($\mathcal{N}, A_G^i, (A_S^i, u_{nxt})$)
5 for each $v_c \in M_C^{temp}$ do
6     if $\sigma(u_{nxt}) \subseteq \sigma(v_c)$ then
7         add $v_c$ to $M_C$                  /* A valid matchable vertex */
8 return $M_C$
```

The FINDJOINABLE procedure guarantees the structural connectivity of the embeddings that are outputted. Referring to Figure 1, let us assume that the already matched query vertices $M_S = \{u_2, u_3\}$ and the corresponding matched data vertices $M_G = \{v_3, v_5\}$, and the next query vertex to be matched $u_{nxt} = u_1$. Initially, in the FINDJOINABLE procedure, for the next query vertex u_{nxt}, we collect all the neighbourhood vertices that have been already matched, and store them in A_S; formally, $A_S := M_S \cap adj(u_{nxt})$ and also collect the corresponding matched data vertices A_G (Line 1-2). For instance, for the next query vertex u_1, $A_S = \{u_2, u_3\}$ and correspondingly, $A_G = \{v_3, v_5\}$.

THEOREM. — 2. *The algorithm* FINDJOINABLE *guarantees to retain the structure of the embeddings.*

PROOF. — Consider a query S of size $|U|$. For $n = 1$, let as assume the first matching M_d^1 corresponds to the initial query vertex M_q^1. Now, A_S and A_G contain all the adjacent vertices of the previously matched vertices M_q^1 and M_d^1 respectively, thus maintaining the connectivity with the partially matched solution M. Hence for $n > 1$, by induction, the structure of entire embedding (that corresponds to the subgraph) is retained. ■

Now we exploit the neighbourhood index $\mathcal{N}$ in order to find the valid matches for the next query vertex u_{nxt}. With the help of vertex $\mathcal{N}$, we find the possible candidate vertices M_C^{temp} for each of the matched query neighbours A_S^i and the corresponding matched data neighbour A_G^i.

To perform querying on the index structure $\mathcal{N}$, we fetch the multiedge that connects the next matchable query vertex u_{nxt} and the i^{th} previously matched query vertex A_S^i. We now take the multiedge (A_S^i, u_{nxt}) and query the index structure $\mathcal{N}$ of the correspondingly matched data vertex A_G^i (Line 4). For instance, with $A_S^i = u_2$, and $u_{nxt} = u_1$ we have a multiedge $\{E_1, E_2\}$. As we can recall, each data vertex v_j has its neighbourhood index structure $\mathcal{N}(v_j)$, represented by an OTIL structure. The elements that are added to OTIL are nothing but the multiedges that are incident on the vertex v_j, and hence the nodes in the tree are nothing but the edge dimensions. Further, each of these edge dimensions (nodes) maintain a list of neighbourhood (adjacent) data vertices of v_j that contain the particular edge dimension as depicted in Figure 2b. Now, when we look up for the multiedge (A^i_S, u_{nxt}), which is nothing but a set of edge dimensions, in the OTIL structure $\mathcal{N}(A^i_G)$, two possibilities exist. (1) The multiedge (A^i_S, u_{nxt}) has no matches in $\mathcal{N}(A^i_G)$ and hence, there are no matchable data vertices for the next query vertex u_{nxt}. (2) The multiedge (A^i_S, u_{nxt}) has matches in $\mathcal{N}(A^i_G)$ and hence, NEIGHINDEXQUERY returns a set of possible candidate vertices M^{temp}_C. The set of vertices M^{temp}_C, present in the OTIL structure as a linked list, are the possible data vertices since, these are the neighbourhood vertices of the already matched data vertex A^i_G, and hence the structure is maintained. For instance, multiedge $\{E_1, E_2\}$ has a set of matched vertices $\{v_2, v_4\}$ as we can observe in Figure 2.

Further, we check if the next possible data vertices are maintaining the structural connectivity with all the matched data neighbours A_G, that correspond to matched query vertices A_S, and hence we collect only those possible candidate vertices M^{temp}_C, that are common to all the matched data neighbours with the help of intersection operation $\cap$. Thus we repeat the process for all the matched query vertices A_S and the corresponding matched data vertices A_G to ensure structural connectivity (Line 4). For instance, with $A^1_S = u_2$ and corresponding $A^1_G = v_3$, we have $M^{temp_1}_C = \{v_2, v_4\}$; with $A^2_S = u_3$ and corresponding $A^2_G = v_5$, we have $M^{temp_2}_C = \{v_4\}$, since the multiedge between (A^i_S, u_{nxt}) is $\{E_2\}$. Thus, the common vertex v_4 is the one that maintains the structural connectivity, and hence belongs to the set of matchable candidate vertices $M^{temp}_C = v_4$.

The set of matchable candidates M^{temp}_C contains the valid candidates for u_{nxt} both in terms of edge dimension matching and the structural connectivity with the already matched partial solution. However, at this point, we propose a strategy that predicts whether the further growth of the partial matching is possible, w.r.t. to the neighbourhood of already matched data vertices, thereby pruning the search space. We can do this by checking the condition whether the vertex signature $\sigma(u_{nxt})$ is contained in the vertex signature of $v \in M^{temp}_C$ (Line 11-13). This is possible since the vertex signature σ contains the multiedge information about the unmatched query vertices that are in the neighbourhood of already matched data vertices. For instance, v_4 can be qualified as M_C since $\sigma(u_1) \subseteq \sigma(v_4)$. That is, considering the fact that we have found a match for u_1, which is v_4, and that the next possible query vertex is u_4, the superset containment check will assure us the connectivity (in terms of edge dimensions) with the next possible query vertex u_4. Suppose a possible candidate data vertex fails this superset containment test, it means that, the data vertex will be discarded by FINDJOINABLE procedure in the next iteration, and we are avoiding this useless step in advance, thereby making the search more time efficient.

In order to efficiently address the superset containment problem between the vertex signatures $\sigma(v_c)$ and $\sigma(u_{nxt})$, we model this task as a maximum matching problem on a bipartite graph (Hopcroft, Karp, 1973). Basically, we build a bipartite graph whose nodes are the sub-signatures of $\sigma(v_c)$ and $\sigma(u_{nxt})$; and an edge exists between a pair of nodes only if the corresponding sub-signatures do not belong to the same signature, and the i^{th} sub-signature of v_c is a superset of j^{th} sub-signature of u_{nxt}. This construction ensures to obtain at the end a bipartite graph. Once the bipartite graph is built we run a maximum matching algorithm to find a maximum match between the two signatures. If the size of the maximum match found is equal to the size of $\sigma(u_{nxt})$, the superset operation returns true otherwise $\sigma(u_{nxt})$ is not contained in the signature $\sigma(v_c)$. To solve the maximum matching problem on the bipartite graph, we employ the *Hopcroft-Karp* (Hopcroft, Karp, 1973) algorithm.

6. Experimental Evaluation

In this section, we evaluate the performance of IMQA on real multigraphs and compare it with a state of the art method that is able to manage edge labels. We consider five real world multigraphs that have very different characteristics in terms of size (nodes, edges, dimensions) and density. All the experiments were run on a server, with 64-bit Intel 6 processors @ 2.60GHz, and 250GB RAM, running on a Linux OS - Ubuntu. Our methods have been implemented using C++.

6.1. Description of Datasets

To validate the correctness, efficiency and versatility of IMQA, we consider five real world datasets that span over biological and social network data. All the multigraphs considered in this work are undirected and they do not contain any attribute on the vertices. Table 3 offers a quick description of all the characteristics of the benchmarks.

For our analysis, we consider five real world data sets: *DBLP* data set is built by following the procedure adopted in (Boden *et al.*, 2012). Vertices correspond to different authors and the dimensions represent the top 50 Computer Science conferences. Two authors are connected over a dimension if they co-authored at least two paper together in that conference. *BIOGRID* dataset (Bonchi *et al.*, 2014) is a protein-protein interactions network, where nodes represent proteins and the edges represent interactions between the proteins. *FLICKR* [1] dataset has been crawled from Flickr website. Nodes represent users, and the blogger's friends are represented using edges (since edge network is the friendship network among the bloggers). Edge dimensions represents friendship network in the online social media. *YOUTUBE* dataset (Tang *et al.*, 2012) treats users as the nodes and the various connections among them as multiedges. The edge information includes the contacts, mutual-contact, co-subscription network,

1. http://socialcomputing.asu.edu/pages/datasets

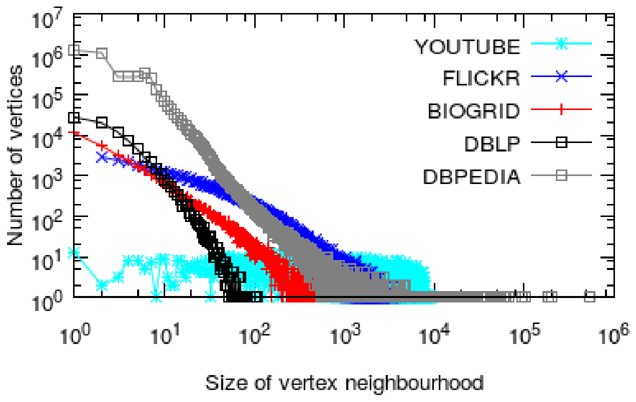


Figure 3. # vertices against size of vertex neighbourhood

co-subscribed network. *DBPEDIA* [2] is a well-known knowledge base, in RDF format, built by the Semantic Web Community. The RDF format can naturally be modeled as a multigraph where vertices are subjects and objects of the RDF triplets and edges represent the predicate between them.

Table 3. Statistics of datasets

Dataset	Nodes	Edges	Dim	Density	A_{deg}	A_{dim}
DBLP	83 901	141 471	50	4.0e-5	1.7	1.126
BIOGRID	38 936	310 664	7	4.1e-4	8.0	1.103
FLICKR	80 513	5 899 882	195	1.8e-3	73.3	1.046
YOUTUBE	15 088	19 923 067	5	1.8e-1	1320	1.321
DBPEDIA	4 495 642	14 721 395	676	1.4e-6	3.2	1.063

To support the analysis of the results, for all the real graphs, we provide the vertex neighbourhood distribution as depicted in Figure 3, where the distribution of the number of vertices with the increasing size of vertex neighbourhood is plotted on a logarithmic scale.

Referring to Figure 3 and Table 3, we make few observations on the data sets. The *YOUTUBE* data set has a flat spectrum of vertex distribution due to its high density of 1.8e-1, and is mostly concentrated in the region of larger neighbourhood size, given its high average degree A_{deg} = 1320. *FLICKR*, *BIOGRID*, *DBLP* and *DBPEDIA* datasets are less dense and hence exhibit a more common power law distribution. Also, as the A_{deg} values reduce from *FLICKR* to *BIOGRID* to *DBPEDIA* and finally to *DBLP*, the distribution shifts towards the smaller neighbourhood size. The sparsest multigraph we consider is *DBPEDIA* that has a density of 1.4e-6 while it exhibits a very high

2. http://dbpedia.org/

number of dimensions and is the biggest real multigraph, in terms of vertices, with more than 4M nodes.

6.2. Description of Query Subgraphs

To test the behavior of our approach, we generate *random* queries and *clique* queries at random, as done by standard subgraph querying methods (He, Singh, 2008; Shang *et al*., 2008). The size of the generated queries for random queries vary from 3 to 11 in steps of 2, while for clique queries, we vary the size from 3 to 9. The size of a subgraph is the number of vertices the subgraph contains. For the *DBPEDIA* dataset, we are not able to generate enough multigraph clique queries due its high sparsity.

All the generated queries contain one (or more) edge with at least two dimensions. In order to generate queries that can have at least one embedding, we sample them from the corresponding multigraph.

For each dataset and query size we obtain 1 000 samples. Following the methodology previously proposed for random query matching algorithms (Han *et al*., 2013; Lin, Bei, 2014), we report the average time values considering the first 1 000 embeddings for each query. It should be noted that the queries returning no answers were not counted in the statistics (the same statistical strategy has been used by (P. Zhao, Han, 2010; He, Singh, 2008; Lin, Bei, 2014)).

6.3. Baseline Approaches

We compare the performance of IMQA w.r.t. the RI approach recently proposed in (Bonnici *et al*., 2013). The RI method is a subgraph isomorphism algorithm that employs light pruning rules in order to avoid visiting useless candidates. The goal of this algorithm is to maintain a balance between the size of the generated search space and the time needed to visit it. It is composed of two main steps, the first one is devoted to find a static order of the query nodes using a set of three heuristics that consider the structure of the subgraph. The second step is the subgraph search procedure that makes use of pruning rules to traverse the search space and find embeddings that match the query. The implementation is obtained from the original authors.

In order to evaluate the effectiveness of our indexing schema we introduce a variant of our proposal, which we call IMQA-No-SC. This approach constitutes a baseline w.r.t. our proposal. Practically, it does not consider constructing the vertex signature index $\mathcal{T}$, and hence does not select any candidates for the initial query vertex u_{init}. Thus, it initializes the candidate set of the initial vertex $C(u_{init})$ with the whole set of data nodes. This baseline can help us to have a more clear picture about the impact of the $\mathcal{T}$ index over the performance of our submultigraph isomorphism algorithm.

6.4. Performance of IMQA

In Section 4, we gave emphasis on constructing the vertex signature index $\mathcal{T}$ to store vertex signatures with the help of synopses representation, and the neighbourhood vertex signature $\mathcal{N}$ to organize vertex neighbourhood by exploiting the set of edge dimensions. We recall that IMQA constructs both $\mathcal{T}$ and $\mathcal{N}$ offline. While index $\mathcal{T}$ is explored during the query processing, to retrieve valid candidates for the initial query vertex u_{init}, the index $\mathcal{N}$ is used to retrieve neighbourhood vertices in the subgraph search routine. Table 4 reports the index construction time of IMQA for each of the employed dataset.

All the benchmarks show reasonable time performance and it is strictly related to the size and density of the considered multigraph. As we can observe, construction of the index $\mathcal{N}$ takes more time when compared to the construction of $\mathcal{T}$ for all the datasets except *DBLP*. The behaviour is evident for the bigger datasets like *FLICKR*, *YOUTUBE*, *DBPEDIA* , owing to either huge number of edges, or nodes or both. Considering *DBLP* and *BIOGRID*, we can note that the difference in time construction is strictly related to the size in terms of vertices and edges of the two benchmarks. *DBLP* has a bigger number of vertices than *BIOGRID*, which influences the construction time of the $\mathcal{T}$ index while the construction time of the $\mathcal{N}$ index reflects the difference in terms of edge size between the two data sets. Among all the datasets, *DBPEDIA* is the most expensive dataset to construct both $\mathcal{T}$ and $\mathcal{N}$, since it has huge number of nodes and relatively more edges.

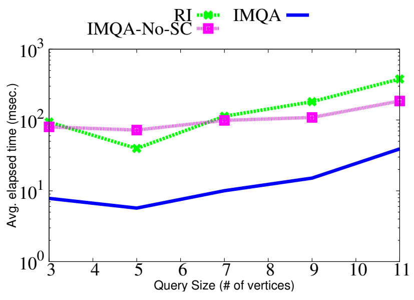
In Table 4, we also give an overall picture of the memory consumption of our proposed algorithm. We capture the memory usage during the runtime when we build our indexing structures. As we can observe, the cost of storing the index structures increases with increasing density of graphs, as well as with the increasing number of nodes and edges. Among all data sets, *YOUTUBE* is the most expensive in terms of space consumption.

To conclude, we highlight that the offline step is fast enough since, in the worst case, for *DBPEDIA*, we need a bit more than two minutes to index 4 million nodes and 14 million edges, with a reasonable memory consumption.

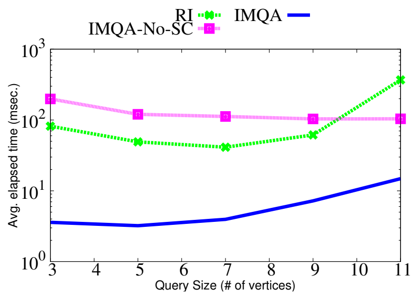
Table 4. Execution time and memory usage for offline index construction

Data set	Index $\mathcal{T}$	Index $\mathcal{N}$	Index $\mathcal{T} + \mathcal{N}$
	Time (seconds)	Time (seconds)	Size (Mega bytes)
DBLP	1.15	0.37	161
BIOGRID	0.45	0.50	266
FLICKR	1.55	8.89	448
YOUTUBE	1.55	41.81	862
DBPEDIA	64.51	66.59	552

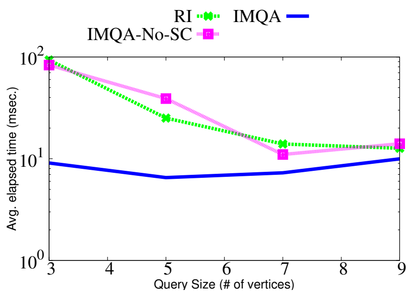
Figures 4-7 summarise the time performance of IMQA. All the times we report are in milliseconds; the Y-axis (logarithmic in scale) represents the query matching time, which includes query processing time, query ordering time, time required to select the candidate vertices for the initial query vertex and the subgraph matching time; the X-axis represents the increasing query sizes. Except for *DBPEDIA* dataset (due to unavailability of clique queries), we produce plots for both random subgraph and clique queries.



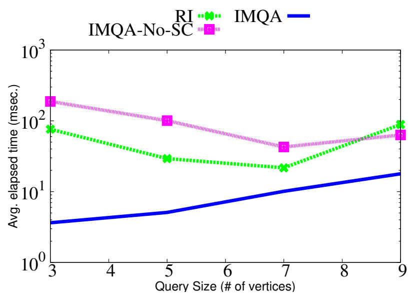
(a)



(b)



(c)



(d)

Figure 4. Query Time on DBLP for (a) Random subgraphs with d=2 (b) Random subgraphs with d=4 (c) Cliques with d=2 (d) Cliques with d=4

We also analyse the time performance of IMQA by varying the number of edge dimensions in the subgraph. We perform experiments for query multigraphs with two different edge dimensions: $d = 2$ and $d = 4$: a query with $d = 2$ has at least one edge that exists in at least 2 dimensions. The same analogy applies to queries with $d = 4$. We use both setting to generate random subgraph and clique queries.

For *DBLP* dataset, we observe in Figure 4 that IMQA performs the best in all the situations, it outperforms the other approaches by a huge margin thanks to the rigorous pruning of candidate vertices for initial query vertex. However, IMQA-No-SC approach and RI give a tough competition to each other. Since *DBLP* is a relatively small and yet sparse dataset, the only indexing $\mathcal{N}$ used by IMQA-No-SC seems to cause a little bit of overhead even when compared to RI.

Figure 5 for *BIOGRID* and Figure 6 for *FLICKR* show similar behaviour for both random subgraph and clique queries. For these two datasets, both IMQA and IMQA-No-SC outperform RI. For many query instances, especially for *FLICKR*, IMQA-No-

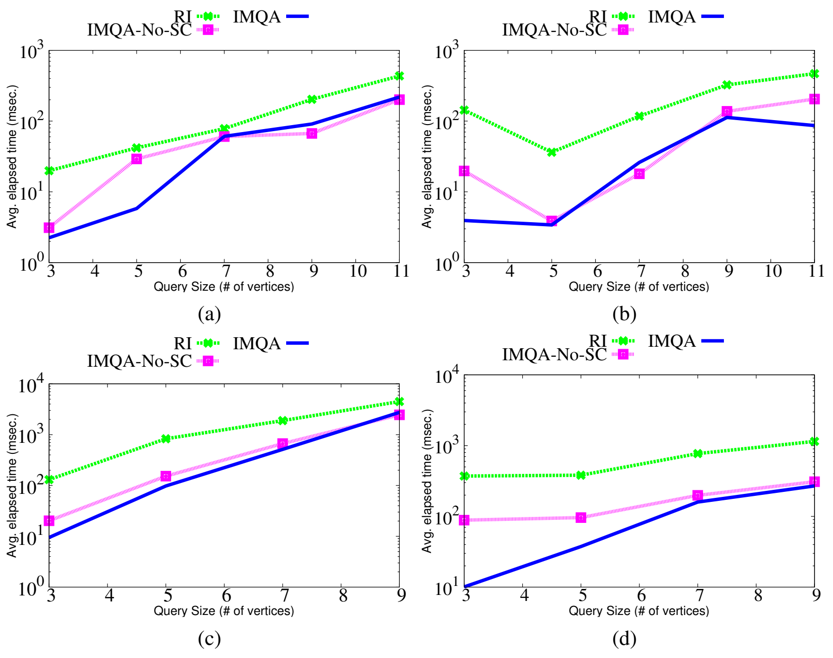


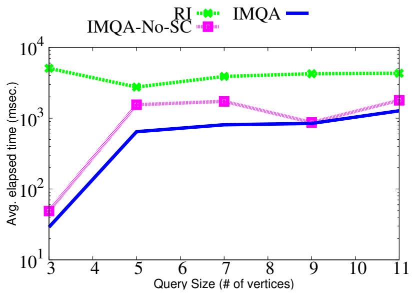
Figure 5. Query Time on BIOGRID for (a) Random subgraphs with d=2 (b) Random subgraphs with d=4 (c) Cliques with d=2 (d) Cliques with d=4

SC obtains better performance than RI while IMQA still outperforms both competitors.

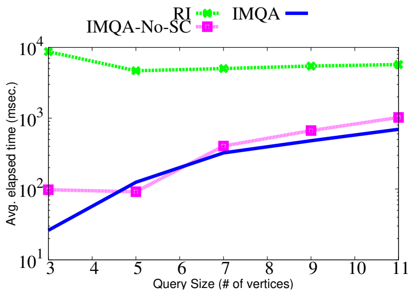
For *YOUTUBE* dataset (Figure 7), again IMQA is the clear winner. However, in this case, RI is better than IMQA-NO-SC, for random queries, although IMQA-NO-SC is better than RI for cliques. This could be the case because, cliques exploit the neighbourhood structure to the maximum extent and thanks to the vertex neighbourhood indexing scheme $\mathcal{N}$, they both can outperform RI. Since random subgraph queries do not exploit much of the neighbourhood information, and due to the very high density of the data graph, IMQA-NO-SC has a poor performance.

Moving to *DBPEDIA* dataset in Figure 8, we observe a significant deviation between RI and IMQA, with IMQA winning by a huge margin.

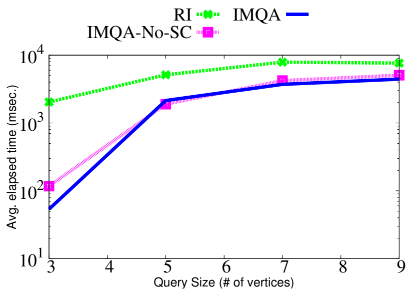
To conclude, we note that IMQA outperforms the considered base line approaches, for a variety of different real datasets. Its performance is reported as best for small datasets (*DBLP*, *BIOGRID*), for multigraphs having many edge dimensions (*FLICKR*, *DBPEDIA*), high density (*YOUTUBE*), high sparsity (*DBPEDIA*). Thus, we highlight that IMQA is robust in terms of time performance considering both subgraph and clique queries, with varying dimensions.



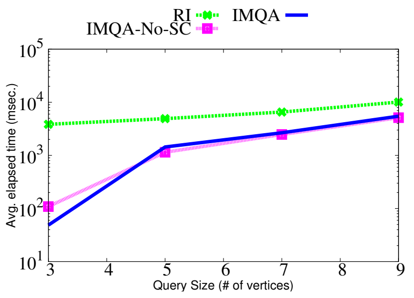
(a)



(b)



(c)



(d)

Figure 6. Query Time on FLICKR for (a) Random subgraphs with d=2 (b) Random subgraphs with d=4 (c) Cliques with d=2 (d) Cliques with d=4

6.5. Assessing the Set of Synopses Features

In this section we assess the quality of the features composing the synopses representation for our indexing schema. To this end, we vary the features we consider to build the synopsis representation to understand if some of the features can be redundant and/or do not improve the final performance. Since visualizing the combination of the whole set of features will be hard, we limit this experiment to a subset of combinations. Hence, we choose to vary the size of the feature set from one to six, by considering the order defined in Section 4.1. Using all the six features results in the proposed approach IMQA. We denote the different configuration with the number of features it contains; for instance $|f| = 3|$ means that it considers only three features to build synopses and in particular it employs the feature set $\{f_1, f_2, f_3\}$. We also compare these six tests with the IMQA-No-SC approach, where no synopses are used and hence no candidates are selected for the initial query vertex.

Due to space constraints, we report plots for only two datasets: *DBLP* for subpgraph queries with $d = 4$ and *YOUTUBE* with subgraph queries with $d = 2$. We select these datasets as they are representative cases of the behavior of our indexing schema.

Results are reported in Figure 9. We can note that, considering the entire set of features drastically improves the time performance, when compared to a subset of

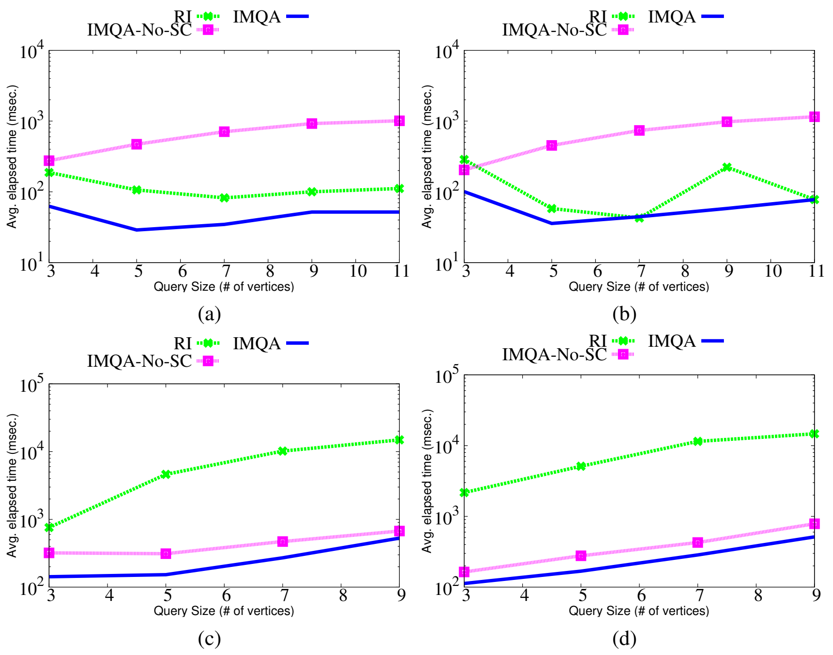


Figure 7. Query Time on YOUTUBE for (a) Random subgraphs with d=2 (b) Random subgraphs with d=4 (c) Cliques with d=2 (d) Cliques with d=4

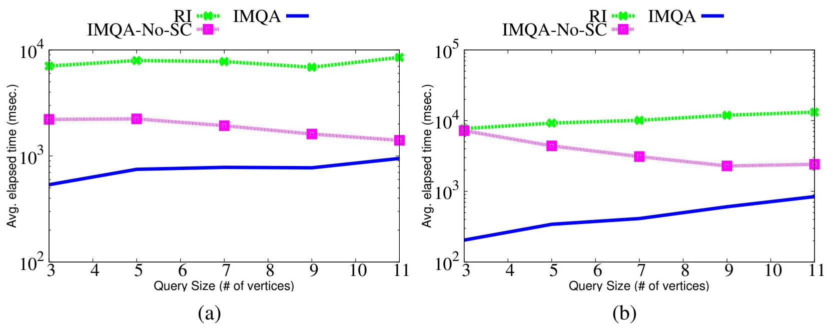


Figure 8. Query Time on DBPEDIA (a) for d=2 (b) d=4

these six features. This behaviour can be highlighted for the subgraphs of almost all size. This experiment provides evidence about the usefulness of considering the entire feature set to build synopsis. The different features are not redundant and they are all helpful in pruning the useless data vertices.

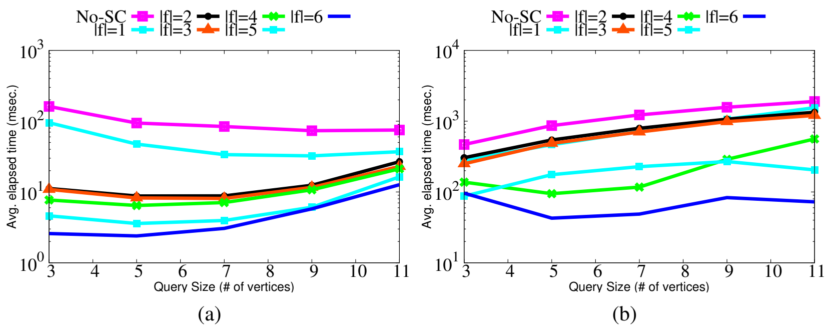


Figure 9. Query time with varying synopses fields for (a) DBLP dataset with d=4 (b) YOUTUBE dataset with d=2

7. Conclusion

We proposed an efficient algorithm IMQA that can perform subgraph matching on multigraphs. The main contributions included the construction of indexing for the edge dimensions ($\mathcal{T}$) to prune the possible candidates, followed by building an ordered tree with inverted lists ($\mathcal{N}$) to retain only the valid candidates. Then we proposed a subgraph search procedure that efficiently works on multigraphs. The experimental section highlights the efficiency, versatility and scalability of our approach over very different datasets.

Acknowledgments:

This work has been funded by Labex NUMEV (NUMEV, ANR-10-LABX-20) and French National Research Agency, framework of the program "Investissements d'Avenir" (GEOSUD, ANR-10- EQPX-20).

References

Boden B., Günnemann S., Hoffmann H., Seidl T. (2012). Mining coherent subgraphs in multi-layer graphs with edge labels. In *Kdd*, pp. 1258–1266.

Bonchi F., Gionis A., Gullo F., Ukkonen A. (2014). Distance oracles in edge-labeled graphs. In *Edbt*, p. 547-558.

Bonnici V., Giugno R., Pulvirenti A., Shasha D., Ferro A. (2013). A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinformatics*, Vol. 14, No. S-7, pp. S13.

Cheng J., Ke Y., Ng W., Lu A. (2007). Fg-index: towards verification-free query processing on graph databases. In *Sigmod*, pp. 857–872.

Cordella L. P., Foggia P., Sansone C., Vento M. (2004). A (sub) graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 26, No. 10, pp. 1367–1372.

Han W.-S., Lee J., Lee J.-H. (2013). Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Sigmod*, pp. 337–348.

He H., Singh A. K. (2008). Graphs-at-a-time: query language and access methods for graph databases. In *Sigmod*, pp. 405–418.

Hopcroft J. E., Karp R. M. (1973). An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, Vol. 2, No. 4, pp. 225–231.

Lee J., Han W.-S., Kasperovics R., Lee J.-H. (2012). An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, Vol. 6, No. 2, pp. 133–144.

Libkin L., Reutter J., Vrgoč D. (2013). Trial for rdf: adapting graph query languages for rdf data. In *Pods*, pp. 201–212.

Lin Z., Bei Y. (2014). Graph indexing for large networks: A neighborhood tree-based approach. *Knowledge-Based Systems*, Vol. 72, pp. 48–59.

Ren X., Wang J. (2015). Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *PVLDB*, Vol. 8, No. 5, pp. 617–628.

Shang H., Zhang Y., Lin X., Yu J. X. (2008). Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, Vol. 1, No. 1, pp. 364–375.

Tang L., Wang X., Liu H. (2012). Community detection via heterogeneous interaction analysis. *Data Mining and Knowledge Discovery*, Vol. 25, pp. 1-33.

Terrovitis M., Passas S., Vassiliadis P., Sellis T. (2006). A combination of trie-trees and inverted files for the indexing of set-valued attributes. In *Cikm*, pp. 728–737.

Yan X., Yu P. S., Han J. (2004). Graph indexing: a frequent structure-based approach. In *Sigmod*, pp. 335–346.

Zhang A. (2009). Protein interaction networks: Computational analysis. *Cambridge University Press*.

Zhao P., Han J. (2010). On graph query optimization in large networks. *PVLDB*, Vol. 3, No. 1-2, pp. 340–351.

Zhao X., Xiao C., Lin X., Wang W., Ishikawa Y. (2013). Efficient processing of graph similarity queries with edit distance constraints. *VLDB Journal*, Vol. 22, No. 6, pp. 727–752.