# SuMGra: Querying Multigraphs via Efficient Indexing

Vijay Ingalalli[1,2](✉), Dino Ienco[2], and Pascal Poncelet[1]

[1] Université de Montpellier, LIRMM, Montpellier, France
{vijay,pascal.poncelet}@lirmm.fr
[2] IRSTEA Montpellier, UMR TETIS, F-34093 Montpellier, France
dino.ienco@irstea.fr

**Abstract.** Many real world datasets can be represented by a network with a set of nodes interconnected with each other by multiple relations. Such a rich graph is called a multigraph. Unfortunately, all the existing algorithms for subgraph query matching are not able to adequately leverage multiple relationships that exist between the nodes. In this paper we propose an efficient indexing schema for querying single large multigraphs, where the indexing schema aptly captures the neighbourhood structure in the data graph. Our proposal SuMGra couples this novel indexing schema with a subgraph search algorithm to quickly traverse though the solution space to enumerate all the matchings. Extensive experiments conducted on real benchmarks prove the time efficiency as well as the scalability of SuMGra.

## 1 Introduction

Many real world datasets can be represented by a network with a set of nodes interconnected with each other by multiple relations. Such a rich graph is called multigraph and it allows different types of edges in order to represent different types of relations between vertices [1,2]. Example 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 [15]; 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 [10].

One of the difficult operation in graph data management is subgraph querying [6]. Although subgraph querying is an NP-complete [6] problem, practically, we can find embeddings in real graph data by employing a good 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. During filtering, some graph patterns (subtrees or paths) are chosen as indexing features to minimize the number of candidate graphs. Then the verification step checks for the subgraph

© Springer International Publishing Switzerland 2016
S. Hartmann and H. Ma (Eds.): DEXA 2016, Part I, LNCS 9827, pp. 387–401, 2016.
DOI: 10.1007/978-3-319-44403-1_24

isomorphism using the selected candidates [4,11,14,16]. 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 on the single multigraph scenario. A second family of approaches avoids 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 [5,7,13]. All these approaches are able to manage graphs with only a single label on the vertex. Although index based approaches focus on transactional database graphs, some backtracking algorithms address the large single graph setting [9]. 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 recent family of techniques defines *equivalence classes* at query and/or database level, by exploiting vertex relationships. Once the data vertices are grouped into equivalence classes, the search space is reduced and the whole process is speeded up [6,12].

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, [3] is the unique approach, which is a backtracking 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 availability of multigraph data and the importance of performing query on multigraph data, in this paper, we propose a novel method SuMGra that supports subgraph matching in a multigraph via efficient indexing. In particular, we capture the multigraph properties in order to build the index structures, and we show that by exploiting multigraph properties, we are able to perform subgraph matching very efficiently. As observed in Table 1, the proposed SuMGra is almost one order of magnitude better than the benchmark approach *RI*, for the DBPEDIA dataset and for the query sizes from 3 to 11.

Deviating from all the previous proposed approaches, we conceive an indexing schema to summarize information contained in a single large multigraph. SuMGra 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 search procedure exploits the indexing schema previously built.

**Table 1.** Time (msec) taken by *RI* and SuMGra for DBPEDIA dataset

| Approach | 3 | 5 | 7 | 9 | 11 |
|---|---|---|---|---|---|
| SuMGra | 160.5 | 254.1 | 545.7 | 971.5 | 1610.9 |
| *RI* | 7045.7 | 7940.8 | 7772.7 | 6872.6 | 8502.5 |

The rest of the paper is organized as follows. Background and problem definition are provided in Sect. 2. An overview of the proposed approach is presented in Sect. 3, while Sects. 4 and 5 describe the indexing schema and the query subgraph search algorithm, respectively. Section 6 presents experimental results. Conclusions are drawn in Sect. 7.

## 2 Preliminaries and Problem Definition

Formally, we can 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-graph isomorphism problem for undirected multigraphs.

**Definition 1.** Subgraph isomorphism for undirected multigraph. *Given a multigraph $Q = (V^q, E^q, L^q_E, D^q)$ and a multigraph $G = (V, E, L_E, D)$, the subgraph isomorphism from $Q$ to $G$ is an injective function $\psi : V^q \rightarrow V$ such that:*

$$\forall (u_m, u_n) \in E^q, \exists\, (\psi(u_m), \psi(u_n)) \in E \text{ and } L^q_E(u_m, u_n) \subseteq L_E(\psi(u_m), \psi(u_n)).$$

**Problem Definition.** Given a query multigraph $Q$ and a data multigraph $G$, the subgraph query problem is to enumerate all the embeddings of $Q$ 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 $Q$ 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 Fig. 1, we introduce a query multigraph $Q$ and a data multigraph $G$. The two valid embeddings for the subgraph $Q$ 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 $[u_i, v_j]$.

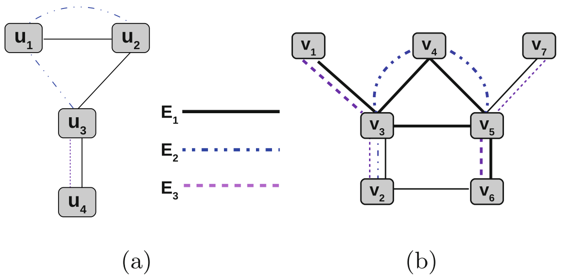


**Fig. 1.** A sample (a) query multigraph $Q$ and (b) data multigraph $G$

## 3 An Overview of SuMGra

In this section, we sketch the main idea behind our proposal. 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 (Sect. 4) (ii) a subgraph search algorithm, that integrates recent advances in the graph data management field, to enumerate the embeddings of the subgraph (Sect. 5).

The overall idea of SuMGra is depicted in Algorithm 1. Initially, we order the set of query vertices $U$ using a heuristic proposed in Sect. 5.1. With an ordered set of query vertices $U^o$, we use the indexing schema to find a list of possible candidate matches only for the initial query vertex $u_{init}$ by calling SelectCand (Line 5), as described in Sect. 5.2. Then, for each possible candidate of the initial query vertex, we call the recursive subroutine SubgraphSearch, that performs the subgraph isomorphism test.

The SubgraphSearch procedure (Sect. 5.3), finds the embeddings starting with the possible matches for the initial query vertex $u_{init}$ (Lines 7–11). Since $u_{init}$ has $|C_{u_{init}}|$ possible matches, SubgraphSearch iterates through $|C_{u_{init}}|$ solution trees in a depth first manner until an embedding is found. That is, SubgraphSearch is recursively called to find the matchings that correspond to all ordered query vertices $U^o$. The partial embedding is stored in $M = [M_q, M_g]$ - a pair that contains the already matched query vertices $M_q$ 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.

**Algorithm 1.** SuMGra

```
1  Input: subgraph $Q$, graph $G$, indexes $\mathcal{S}$, $\mathcal{N}$
2  Output: $R$: all the embeddings of $Q$ in $G$
3  $U^o$ = OrderQueryVertices($Q$, $G$)
4  $u_{init} = u | u \in U^o$
5  $C_{u_{init}}$ = SelectCand($u_{init}$, $\mathcal{S}$)
6  $R = \emptyset$                                   /* Embeddings of $Q$ in $G$ */
7  for each $v_{init} \in C_{u_{init}}$ do
8      $M_q = u_{init}$;                             /* Matched initial query vertex */
9      $M_d = v_{init}$;                             /* Matched possible data vertex */
10     $M = [M_q, M_g]$                              /* Partial matching of $Q$ in $G$ */
11     Update: $R$ := SubgraphsSearch($R, M, \mathcal{N}, Q, G, U^o$)
12 return $R$
```

## 4 Indexing

In this section, we propose the indexing structures that are built on the data multigraph $G$, by leveraging the multigraph properties in specific; this index is 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 a multiset containing all the multiedges that are incident on $v$, where a 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.*

For instance, in Fig. 1(b), $\sigma(v_6) = \{\{E_1, E_3\}, \{E_1\}\}$. The vertex signature is an intermediary representation that is exploited by our indexing schema.

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 SubgraphSearch procedure, making SuMGra 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 $\mathcal{S}$ 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 $\mathcal{S}$ is used to select possible candidates for the initial query vertex in the Select-Cand 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 $\mathcal{S}$

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, $\mathcal{S}$ captures the edge dimension information from the data vertices, so that the non suitable candidates can be pruned away.

We construct the index $\mathcal{S}$ 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 $|f| = 6$ features, that leverage the multigraph properties; the features 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 (repetition allowed), ($f_3(v_3) = 8$)

$f_4$ Minimum index of the lexicographically ordered edge dimensions, ($f_4(v_3) = 1$)
$f_5$ Maximum index of the lexicographically ordered edge dimensions, ($f_5(v_3) = 3$)
$f_6$ Maximum cardinality of the vertex sub-signature, ($f_6(v_3) = 3$)

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; e.g., 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 $\mathcal{S}$, where each synopses is treated as an $|f|$-dimensional node of the R-tree.

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

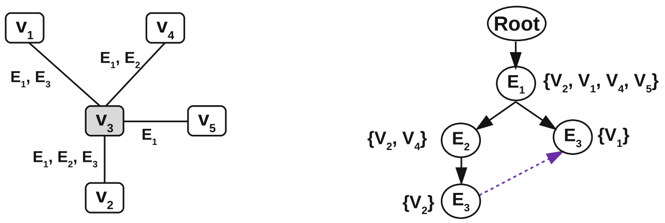
The aim of this indexing structure is to find the possible candidates for the rest of the query vertices.

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 SUBGRAPH-SEARCH procedure by retaining the structural connectivity with the previously matched candidate vertices, while discovering the embeddings of the subgraph $Q$ 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 Fig. 1(b), shown separately in Fig. 2(a). 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 Fig. 2(b), where the nodes of the tree structure are represented by the edge dimensions.

In order to construct an efficient tree structure, we 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 Fig. 2(b), 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 Fig. 2(b), 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 Fig. 2(b) 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) (b)

**Fig. 2.** (a) Neighbourhood structure of $v_3$ and (b) Neighbourhood index for 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.

### 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 [9]. 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 Fig. 1(a), 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 $\mathcal{S}$ to retrieve the set of possible candidate data vertices, thereby pruning the unwanted candidates for the reduction of search space.

During the SelectCand procedure (Algorithm 1, Line 5), we retrieve the possible candidate vertices from the data graph by exploiting the vertex signature index $\mathcal{S}$. However, since querying $\mathcal{S}$ would not prune away all the unwanted vertices for $u_{init}$, the corresponding partial embeddings would be discarded during the SubgraphSearch 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 $\mathcal{S}$. 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 $\{v_3, v_5\}$.

### 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

**Algorithm 2.** SubgraphSearch($R, M, \mathcal{N}, Q, G, U^o$)

```
Fetch $u_{nxt} \in U^o$                                  /* Fetch query vertex to be matched */
$M_C$ = FindJoinable($M_q, M_g, \mathcal{N}, u_{nxt}$)       /* Matchable candidate vertices */
if $|M_C| \neq \emptyset$ then
    for each $v_{nxt} \in M_C$ do
        $M_q = M_q \cup u_{nxt}$;
        $M_g = M_g \cup v_{nxt}$;
        $M = [M_q, M_g]$                                 /* Partial matching grows */
        SubgraphSearch($R, M, \mathcal{N}, Q, G, U^o$)
        if $(|M| == |U^o|)$ then
            $R = R \cup M$                               /* Embedding found */
return $R$
```

solution pair $M = [M_q, M_g]$ 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 $M = [M_q, M_g]$ (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($M_q, M_g, \mathcal{N}, u_{nxt}$)

```
$A_q := M_q \cap adj(u_{nxt})$                        /* Matched query neighbours */
$A_g := \{v | v \in M_g\}$                  /* Corresponding matched data neighbours */
Intialize: $M_C^{temp} = 0, M_C = 0$
$M_C^{temp} = \cap_{i=1}^{|A_q|}$ NeighIndexQuery($\mathcal{N}, A_g^i, (A_q^i, u_{nxt})$)
for each $v_c \in M_C^{temp}$ do
    if $\sigma(v_c) \supseteq \sigma(u_{nxt})$ then
        add $v_c$ to $M_C$                              /* A valid matchable vertex */
return $M_C$
```

The FindJoinable procedure guarantees the structural connectivity of the embeddings that are outputted. Referring to Fig. 1, let us assume that the already matched query vertices $M_q = \{u_2, u_3\}$ 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 $A_q$; formally, $A_q := M_q \cap adj(u_{nxt})$ and also collect the corresponding matched data vertices $A_g$ (Line 1–2). For instance, for the next query vertex $u_1$, $A_q = \{u_2, u_3\}$ and correspondingly, $A_g = \{v_3, v_5\}$.

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 $A_q^i$ 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 $A_q^i$. We now take the multiedge $(A_q^i, u_{nxt})$ and query the index structure $\mathcal{N}$ of the correspondingly matched data vertex $A_g^i$ (Line 4). For instance, with $A_q^i = u_2$, 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 Fig. 2(b). Now, when we look up for the multiedge $(A_q^i, u_{nxt})$, which is nothing but a set of edge dimensions, in the OTIL structure $\mathcal{N}(A_g^i)$, two possibilities exist. (1) The multiedge $(A_q^i, u_{nxt})$ has no matches in $\mathcal{N}(A_g^i)$ and hence, there are no matchable data vertices for the next query vertex $u_{nxt}$. (2) The multiedge $(A_q^i, u_{nxt})$ has matches in $\mathcal{N}(A_g^i)$ and hence, NeighIndexQuery returns a set of possible candidate vertices $M_C^{temp}$. The set of vertices $M_C^{temp}$, 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 $A_g^i$, 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 Fig. 2(a).

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 $A_q$, and hence we collect only those possible candidate vertices $M_C^{temp}$, 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 $A_q$ and the corresponding matched data vertices $A_g$ to ensure structural connectivity (Line 4). For instance, with $A_q^1 = u_2$ and corresponding $A_g^1 = v_3$, we have $M_C^{temp1} = \{v_2, v_4\}$; with $A_q^2 = u_3$ and corresponding $A_g^2 = v_5$, we have $M_C^{temp2} = \{v_4\}$, since the multiedge between $(A_q^i, u_{nxt})$ 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 $M_C^{temp} = v_4$.

The set of matchable candidates $M_C^{temp}$ are 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 $v \in M_C^{temp}$ (Line 11–13). This is possible since, the vertex signature $\sigma$ 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 $\sigma(v_4)$

$\supseteq \sigma(u_1)$. 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 [8]. 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* [8] algorithm.

## 6 Experimental Evaluation

In this section, we evaluate the performance of SuMGra on real multigraphs.

We evaluate the performance of SuMGra by comparing it with two baseline approaches (its own variants) and a competitor *RI* [3]. The two baseline approaches are: (i) SuMGra-No-SC that does not consider the vertex signature index $\mathcal{S}$ and it initializes the candidate set of the initial vertex $C(u_{init})$ with the whole set of data nodes; (ii) SuMGra-Rand-Order that consider all the indexing structure but it employs a random ordering of the query vertices preserving connectivity. The *RI* approach is able to manage graphs with multi-edges, and we obtain the implementation from the original authors. For the purpose of evaluation. we consider three real world multigraphs: *DBLP* data set built by following the procedure adopted in [1]; *FLICKR*[1] crawled from Flickr, which is an image and video hosting website, web services suite, and an online community; *DBPEDIA*[2] that is the well-known knowledge base built by the Semantic Web Community. For *DBLP*, vertices correspond to different authors and each dimensions represent one of the top 50 Computer Science conferences. Two authors are connected over a dimension if they co-authored at least one paper together in that conference. In *FLICKR*, users are represented by nodes, and blogger's friends are represented using edges. Multiple edges exist between two users if they have common multiple memberships. The RDF format in which *DBPEDIA* is stored can naturally be modeled as a multigraph where vertices are subjects and objects of the RDF triplets and edges represent the predicates between them. Benchmark characteristics are reported in Table 2.

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

[2] http://dbpedia.org/.

**Table 2.** Benchmark statistics.

| Dataset | Nodes | Edges | Dim | Density |
|---|---|---|---|---|
| *DBLP* | 83 901 | 141 471 | 50 | 4.0e-5 |
| *FLICKR* | 80 513 | 5 899 882 | 195 | 1.8e-3 |
| *DBPEDIA* | 4 495 642 | 14 721 395 | 676 | 1.4e-6 |

**Table 3.** Index construction time (secs.).

| Dataset | $\mathcal{S}$ | $\mathcal{N}$ |
|---|---|---|
| *DBLP* | 1.15 | 0.37 |
| *FLICKR* | 1.55 | 8.89 |
| *DBPEDIA* | 64.51 | 66.59 |

To test the behavior of the different approaches, we generate *random* queries [7,13] varying their size (in terms of vertices) from 3 to 11 in steps of 2. 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 [6,11], 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 [7,11]).

All the experiments were run on a server, with 64-bit Intel 6 processors @ 2.60 GHz, and 250 GB RAM, running on a Linux OS - Ubuntu. Our methods have been implemented using C++.

## 6.1 Performance of SuMGra

Table 3 reports the index construction time of SuMGra for each of the employed dataset. As we can observe for the bigger datasets like *FLICKR*, and *DBPEDIA*, construction of the index $\mathcal{N}$ takes more time when compared to the construction of $\mathcal{S}$. This happens due to either huge number of edges, or nodes or both in these two datasets. For *DBLP* we can observe the opposite phenomenon. This can be explained by the small number of edges and dimensions present in this dataset. Among all the datasets, *DBPEDIA* is the most expensive dataset in terms of indices construction but it always remains reasonable as time consumption for the off-line step is around one minute for each index.

**Query Processing Time.** Figures 3, 4 and 5 summarize time results. All the times we report are in milliseconds; the Y-axis (logarithmic in scale) represents the query matching time; the X-axis represents the increasing query sizes.

We also analyse the time performance of SuMGra by varying the number of edge dimensions in the subgraph. In particular, we perform experiments for query multigraphs with two different edge dimensions: $d = 2$ and $d = 4$. That is, a query with $d = 2$ has at least one edge that exists in at least 2 dimensions. The same analogy applies to the queries with $d = 4$.

For *DBLP* dataset, we observe in Fig. 3 that SuMGra performs the best in all the situations, and in fact it outperforms the other approaches by a huge margin. This happens thanks to both: a rigorous pruning of candidate vertices

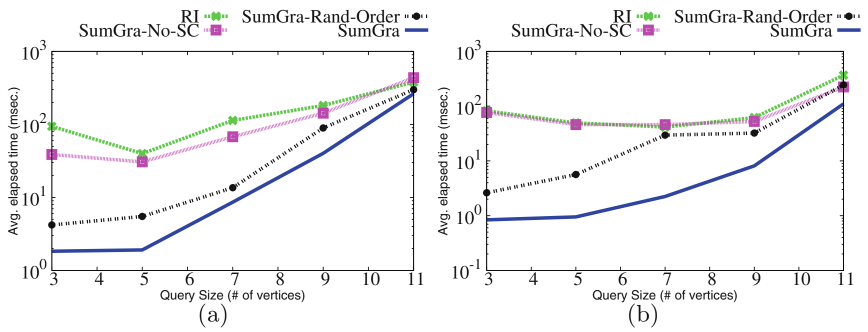


**Fig. 3.** Query time on *DBLP* dataset for (a) with $d = 2$ (b) with $d = 4$

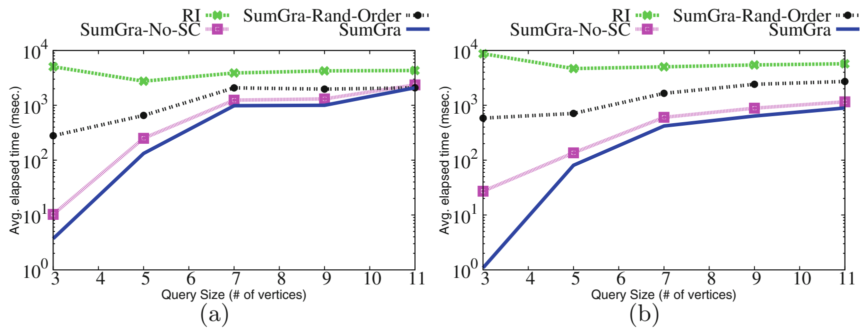


**Fig. 4.** Query time on *FLICKR* dataset for (a) with $d = 2$ (b) with $d = 4$

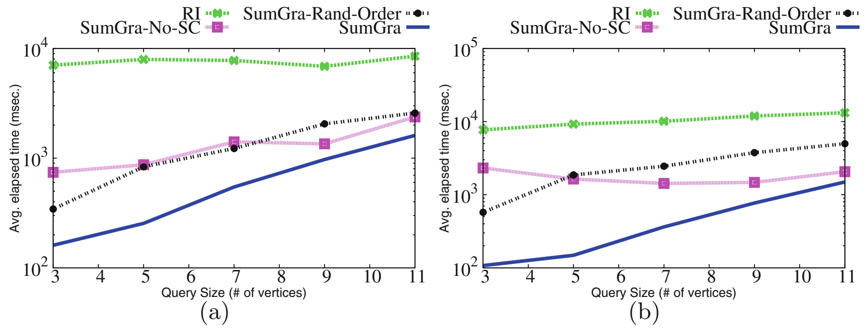


**Fig. 5.** Query time on *DBPEDIA* dataset (a) for $d = 2$ (b) $d = 4$

for initial query vertex as underlined by the gain w.r.t. SuMGra-No-SC and an efficient query vertex ordering strategy as highlighted by the difference w.r.t. SuMGra-Rand-Order. For the *FLICKR* dataset (Fig. 4) SuMGra, SuMGra-No-SC and SuMGra-Rand-Order outperform *RI*. For many query instances, especially for *FLICKR*, SuMGra-No-SC obtains better performance than RI while SuMGra still outperforms competitors. We can observe that random query ordering drastically affects the performance pointing out the importance of this step. Moving to *DBPEDIA* dataset in Fig. 5, we observe a significant deviation between *RI* and SuMGra, with SuMGra winning by a huge margin.

To conclude, we note that SuMGra outperforms the considered competitors, for all the employed benchmarks for all query size. Its performance is reported as best for multigraphs having many edge dimensions - *FLICKR* and high sparsity - *DBPEDIA*. Thus, we highlight that SuMGra is robust in terms of time performance varying both the query size and dimensions.

**Assessing the Set of Synopses Features.** In this experiment 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 them 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 Sect. 4.1. Using all the six features results in the proposed approach SuMGra.

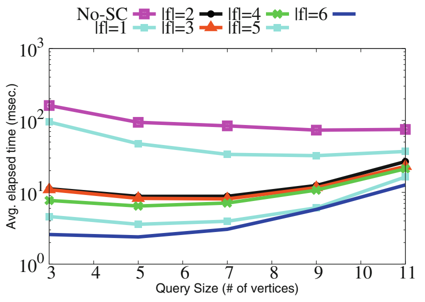


**Fig. 6.** Query time with varying synopses fields for *DBLP* with $d = 4$

We perform experiments with different configurations that have varying number of synopses features; for instance $|f| = 3|$ means that it considers only first three features to build synopses. Although we report plots only for *DBLP* for queries with $d = 4$, the behaviour for different datasets has similar behaviour. Results are reported in Fig. 6. We note that, considering the entire set of features drastically improves the time performance, when compared to a subset of these six features. We conclude that the different features are not redundant and they are all helpful in pruning the useless data vertices.

## 7 Conclusion

We proposed an efficient strategy to support Subgraph Matching in a Multigraph via efficient indexing. The proposed indexing schema leverages the rich structure available in the multigraph. The different indexes are exploited by a subgraph search procedure that works on multigraphs. The experimental section highlights the efficiency, versatility and scalability of our approach over different real datasets. The comparison with a state of the art approach points out the necessity to develop specific techniques to manage multigraphs.

As a future work, we are interesting in testing new synopses features as well as try novel vertex ordering strategies more rigorously. Further, we will be addressing dynamic multigraphs where nodes and multiedges are being added or removed over time.

**Acknowledgments.** This work has been funded by Labex NUMEV (NUMEV, ANR-10-LABX-20).

## References

1. Boden, B., Günnemann, S., Hoffmann, H., Seidl, T.: Mining coherent subgraphs in multi-layer graphs with edge labels. In: KDD, pp. 1258–1266 (2012)
2. Bonchi, F., Gionis, A., Gullo, F., Ukkonen, A.: Distance oracles in edge-labeled graphs. In: EDBT, pp. 547–558 (2014)
3. Bonnici, V., Giugno, R., Pulvirenti, A., Shasha, D., Ferro, A.: A subgraph isomorphism algorithm and its application to biochemical data. BMC Bioinform. **14**(S–7), S13 (2013)
4. Cheng, J., Ke, Y., Ng, W., Lu, A.: Fg-index: towards verification-free query processing on graph databases. In: SIGMOD, pp. 857–872. ACM (2007)
5. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: A (sub) graph isomorphism algorithm for matching large graphs. IEEE TPAMI **26**(10), 1367–1372 (2004)
6. Han, W.-S., Lee, J., Lee, J.-H.: Turbo ISO: towards ultrafast and robust subgraph isomorphism search in large graph databases. In: SIGMOD, pp. 337–348. ACM (2013)
7. He, H., Singh, A.K.: Graphs-at-a-time: query language and access methods for graph databases. In: SIGMOD, pp. 405–418. ACM (2008)
8. Hopcroft, J.E., Karp, R.M.: An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. SIAM J. Comput. **2**(4), 225–231 (1973)
9. Lee, J., Han, W.-S., Kasperovics, R., Lee, J.-H.: An in-depth comparison of subgraph isomorphism algorithms in graph databases. In: PVLDB, pp. 133–144 (2012)
10. Libkin, L., Reutter, J., Vrgoč, D.: Trial for RDF: adapting graph query languages for RDF data. In: PODS, pp. 201–212. ACM (2013)
11. Lin, Z., Bei, Y.: Graph indexing for large networks: a neighborhood tree-based approach. Knowl. Based Syst. **72**, 48–59 (2014)
12. Ren, X., Wang, J.: Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. PVLDB **8**(5), 617–628 (2015)
13. Shang, H., Zhang, Y., Lin, X., Yu, J.X.: Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. PVLDB **1**(1), 364–375 (2008)
14. Yan, X., Yu, P.S., Han, J.: Graph indexing: a frequent structure-based approach. In: SIGMOD, pp. 335–346. ACM (2004)
15. Zhang, A.: Protein Interaction Networks: Computational Analysis. Cambridge University Press, Cambridge (2009)
16. Zhao, X., Xiao, C., Lin, X., Wang, W., Ishikawa, Y.: Efficient processing of graph similarity queries with edit distance constraints. VLDB J. **22**(6), 727–752 (2013)