Subgraph Matching for Single Large Multigraphs
Subgraph Matching for Single Large Multigraphs

Anonyme Anonyme

[Research Report]

RR-2014030, LIRMM. 2014.

<lirmm-01076138>

Subgraph Matching for Single Large Multigraphs

Vijay Ingalalli, LIRMM - Montpellier, France {vijay@lirmm.fr}

Dino Ienco, IRSTEA - Montpellier, France {dino.ienco@irstea.fr}

Pascal Poncelet, LIRMM - Montpellier, France {pascal.poncelet@lirmm.fr}

Subgraph Matching for Single Large Multigraphs

Vijay Ingalalli
LIRMM, Montpellier, France
vijay@lirmm.fr

Dino Ienco
IRSTEA, Montpellier, France
dino.ienco@irstea.fr

Pascal Poncelet
LIRMM, Montpellier, France
pascal.poncelet@lirmm.fr

***Abstract*—Nowadays, many real world data can be represented by a network with a set of nodes interconnected with each other by multiple relations (multiple edges). Such a rich graph, called multigraph, is very appropriate to represent real world scenarios with complex interactions. However, performing sub-multigraph query on enriched graph is still an open issue since, unfortunately, all the existing algorithms for subgraph query matching fail to consider multiple edges between nodes and, nevertheless, they cannot be directly applied to handle multigraphs. 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 SUMGRA, a novel algorithm to extract all the embeddings of a query sub-multigraph from a single large multigraph. SUMGRA 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. Then, it performs an efficient recursive subgraph search to output the entire set of embeddings for the given query. Extensive experiments conducted on both real and synthetic datasets prove the time efficiency as well as the scalability of SUMGRA.**

I. INTRODUCTION

Nowadays, real-world data exhibits relational structure and most of the available information can be represented as a graph where nodes are entities and interactions between nodes are represented by edges. Examples of such graph data are chemical molecules, protein interaction networks, software repositories and social networks. All these real world applications motivate researchers to focus on efficient approaches to manage graph data.

One of the most important tasks in graph data management is subgraph query, where the challenge is to enumerate all the embeddings of a graph query q in a data graph g. The subgraph query matching problem involves the decision problem of subgraph isomorphism. While in theory it belongs to NP-complete class, practically we can find embeddings in real graph data by exploiting better matching order and intelligent pruning rules. All the previously proposed approaches to deal with subgraph query problem consider simple graphs [5], [10] or graphs with some additional information associated with vertices (attributes) [22]. Besides vertex attributes, there exists additional information in the form of multiple edges between nodes. Unfortunately, until now, no effort has been done to solve subgraph query problem on graphs containing multiple edges between nodes. Such kind of network structure are defined as multigraphs, and they allow different types of edges in order to represent different types of relations between vertices [1], [17], [2], [8].

Many real world scenarios can be modelled as multigraphs. For instance, by considering different social networks spanning over the same set of people, but with different life aspects (e.g. social relationships such as Facebook or Twitter, professional interaction such as LinkedIn, leisure time such as Last.FM, etc.), we can get as many edge types as the different aspects. Another example is supplied by social media that allows users to categorize their own connections in different social circles (e.g., 'lists' on Facebook or 'circles' on Google+) [14], such that the social network in itself contains the edges that explicitly represent different interaction types. In biology, protein-protein interaction multigraphs can be created considering pairs of proteins that have direct interaction, physical association or they are co-localised [23]. More examples can be quoted from a gene network where genes are connected by considering the different pathway interactions and recommendation networks [11]. In addition to these examples, Resource Description Framework (RDF) graphs can be naturally represented as multigraphs where the same subject/object node pair is linked by different predicates (properties) that describe different types of relationships [12].

Performing sub-multigraph query can be fruitful for all these domains as it allows data experts to express much richer queries, where they can specify the kind of interactions between nodes they are looking for. As an example, a biologist would find all the protein structures that physically interact with each other and are partially located on a particular area. Another expert that are working with RDF data graph would retrieve subgraph where a part of nodes are related by a certain predicate and another part of nodes connected by a different property.

Due to the availability of such kind of data and the importance of performing sub-multigraph query on multigraph data, in this paper, we introduce a new method called SUMGRA, that addresses the challenge of finding the embeddings of a query multigraph q in a multigraph g. SUMGRA involves two main phases: (1) an off-line phase in which indexing is performed on the data vertices of multigraph by considering the neighbourhood information of each data vertex; (2) an on-line phase, which leverages the indexing schema to retrieve data vertices, and then performs the sub-multigraph search considering powerful pruning rules to enumerate all the available embeddings of the query in the multigraph. The indexing schema exploits the rich structure supplied by the multigraph and it utilizes the information associated with the edge dimensions, in order to facilitate the retrieval of data nodes.

Since many domains supply data that can be modelled as a single large multigraph, SUMGRA is especially tailored to perform sub-multigraph query over single large graph. However, it can also be easily extended to handle multigraph

[Research Report] RR-2014030, LIRMM. 2014. <lirmm-01076138>

transactional database where the database is composed of many small multigraphs.

The main contributions of this paper are the following:

- We formalize the subgraph query problem for multigraphs.
- We propose an efficient indexing approach that exploits vertex neighbourhood information to speed-up the retrieval of possible candidate vertices in the multigraph.
- We propose a novel subgraph search algorithm that recursively enumerates the embeddings.
- We evaluate our approach on both real-world and synthetic data. We show that our method is computationally efficient and the designed indexing schema remarkably reduces the time required to query a multigraph.

The rest of the paper is organized as follows. We discuss about the related works in Section II. Background and problem definition are provided in Section III. An overview of the proposed approach is presented in Section IV, while Section V and Section VI describe the indexing schema and the query subgraph search algorithm, respectively. Experimental results are discussed in Section VII. We discuss our experimental findings and further works in Section VIII. Conclusions are drawn in Section IX.

II. Related Works

As our work addresses the exact subgraph query processing, we will explore the related works that appear in the same hue and we present them under the theme that is paramount in defining them.

Feature based indexing approaches follow the filtering and verification framework. During filtering, some graph patterns are chosen as indexing features to minimize the number of candidate graphs. Then the verification step checks for the subgraph isomorphism using the selected candidates. GraphGrep [16] considers the length of the path within a threshold, as the indexing feature. Owing to the weak pruning power of GraphGrep, the concept of 'discriminative ratio' to select the set of features was introduced in gIndex [21]. Tree+Δ [25] uses discriminative subtrees as indexing features as they are more efficient than indexing frequent subgraphs. In another approach called FG-Index [3], both frequent subgraphs and edges are used as indexing features. An alternative approach of swift-index [15] has been proposed that uses tree features that maintains a prefix-tree structure. Since all these methods are developed for transactional graphs, we cannot exploit them in our single multigraph scenario.

Backtracking algorithms find embeddings by growing the partial solutions only if they fit to be the 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. The subroutine SubgraphSearch is executed in various steps that helps in finding the embeddings of the subgraph query. Ullmann [20] proposed the first algorithm under this framework. During SubgraphSearch, Ullmann adopts a very simple pruning rule (condition on the degree of the vertex) and follows the input order of the query vertices to choose the next vertex. On the other hand, VF2 [4] chooses the next vertex that is connected to the already matched data vertex. It also employs very efficient pruning rules that reduces the search space to find the embeddings. QuickSI [15] builds a minimum spanning tree to find the next query vertex, by assigning weights to the edges of the query graph, depending on the frequency of occurrence of query vertex in the data graph. GraphQL [6] and sPath [24] follow neighbourhood signature based pruning (in a much similar way) to choose the initial set of candidates (the aforementioned approaches simply choose vertices with matching labels), even before calling the SubgraphSearch. GraphQL additionally employs the pruning technique called pseudo subgraph isomorphism that recursively checks if adjacent subtree of a query vertex is subgraph isomorphic to the corresponding feasible data vertex.

Although index based approaches focus on transactional database graphs, many backtracking algorithms address the large single graphs. Also, in [10] we see that all the backtracking algorithms have been employed to test their performance for both database and single graphs. A much recent work Turbo$_{\text{ISO}}$ [5], not quite falling into any of the above themes, has been proposed that outperforms all the existing backtracking algorithms for both the kinds of graphs. They propose a novel concept of *candidate region exploration* to address matching order problem during subgraph isomorphism search, and a novel query processing strategy called *combine and permute* that avoids useless enumerations between query and data vertices. A very recent work exploits neighbourhood tree based approach to index the large graphs [13]. This work introduces the concept of Neighbourhood Trees (NTree), that records the neighbourhood relationships of each vertex in the large graph to filter the non-potential vertices.

Subgraph query algorithms on more rich graph structures data are presented in [22] and [9]. In [22], the authors propose a method to deal with graphs with multiple vertex attributes. To deal with multiple attributes, they transform the subgraph query problem into a high dimensional spatial query problem, and employ R-tree to perform indexing on vertex attributes. [9] promises faster subgraph matching in large graphs by proposing an edge based framework, that considers edges with single label.

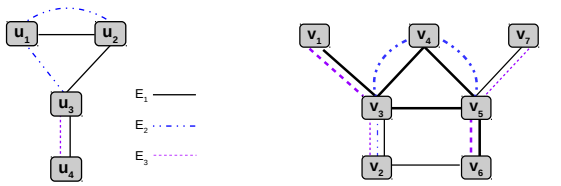
III. Background

A. Preliminaries and Problem Definition

In this section, we provide the basic definitions that are necessary to present our work in the rest of the paper.

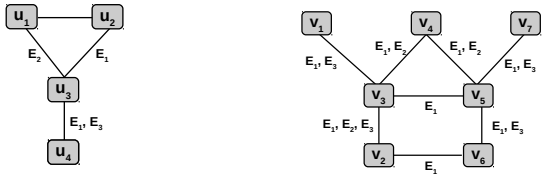
Given a set of dimensions $D = \{D_1, \ldots, D_{|D|}\}$, an unweighted *multigraph* G is defined as a tuple $(V, \{E_i\}_{i=1}^{|D|}, D)$ where, V is the set of vertices, $E_i \subseteq V \times V$ is the set of undirected edges over dimension $D_i \in D$.

However, this standard definition introduces a set of edges E_i for each of the dimension over which the multigraph G is defined. A different but equivalent definition can be formulated by considering each of the dimensions as a label, and assign



(a) Query multigraph q (b) Data multigraph g

Fig. 1: A sample query and data multigraph



(a) Query multigraph q (b) Data multigraph g

Fig. 2: An equivalent representation of multigraphs

the set of dimensions (labels) to each corresponding edge, as depicted in Figure 2.

Formally, we can now redefine a multigraph as $G = (V, E, L_E, D)$ where V and D are defined as before and, in this case, $E \subseteq V \times V$ is the set of undirected edges (without any dependency on the dimensions) 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 the rest of the paper we introduce our framework considering the latter definition of multigraph.

Definition 1: Subgraph isomorphism for a multigraph (SIM). Given a subgraph $S = (V^s, E^s, L_E^s, D^s)$ 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:

1) $\forall u \in V^s, L^s(u) \subseteq L(\psi(u))$
2) $\forall (u_m, u_n) \in E^s, \exists\ (\psi(u_m), \psi(u_n)) \in E$ and $L_E^s(u_m, u_n) \subseteq L_E(\psi(u_m), \psi(u_n))$.

B. Complexity of SIM

The decision problem of subgraph isomorphism is well known to be NP-complete [13]. This standard subgraph isomorphism problem can be seen as a particular case of *SIM* where both the labelling functions L_E^s and L_E always return the same subset of dimensions for all the edges in both S and G. As the standard subgraph isomorphism problem is a particular case of the subgraph isomorphism problem for a multigraph, *SIM* is at least as difficult as the standard subgraph isomorphism problem and hence, we can deduce that the decision problem behind *SIM* is at least NP-complete.

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

In the following work, we enumerate (for unique identification) query vertices by u and the data vertices by v, as shown in Figure 1. In Figure 1a, we observe that $\{E_1, E_2, E_3\}$ are the set of edge dimensions. With the latter definition of multigraph, we can represent the nodes of a graph by a multiset of edge dimensions. For example, in Figure 2, $u_2 := \{\{E_1, E_2\}, \{E_1\}\}$ and $u_3 := \{\{E_1\}\{E_2\}\{E_1, E_3\}\}$. The valid embeddings for the query graph q are marked by the thick lines in the data graph g, as depicted in Figure 1b. For the sake of understanding, we enumerate the complete set of embeddings R as: $R_1 := \{[u_1, v_4], [u_2, v_5], [u_3, v_3], [u_4, v_1]\}$ and $R_2 := \{[u_1, v_4], [u_2, v_3], [u_3, v_5], [u_4, v_6]\}$ where, each query vertex u_i is matched to a unique data vertex v_j, written as $[u_i, v_j]$.

IV. An Overview of SuMGra

In this section, we sketch the main idea of SuMGra to address the subgraph query problem for multigraphs. The entire procedure can be divided into two parts: (i) an indexing schema for the data graph g that exploits edge dimensions (Section V) (ii) a subgraph search algorithm that employs different procedures to enumerate the embeddings of the query graph (Section VI).

The overall idea of SuMGra is depicted in Algorithm 1. Initially, for all the query vertices of q, the possible candidate set $C(u)$ is obtained by calling the SelectCandidates procedure (Line 3). To achieve this, each of the query vertex $u_i \in q$ is matched with the data vertex $v_i \in g$, by exploiting the index built for g (Section V). Then, FindMST is called, that returns a minimum spanning tree [1] q_{mst} (Line 4). Later, in Section VI, we shall see how this can be helpful. Then, the recursive subroutine SubgraphSearch is called to find all the possible embeddings (Line 10). SubgraphSearch begins to find the embeddings starting with the possible matches for the initial query vertex u_{init} (Lines 6-10). Since u_{init} has $C(u_{init})$ possible matches, SubgraphSearch iterates through $|C(u_{init})|$ solution trees. Further, SubgraphSearch is recursively called to find the matchings that correspond to all the query vertices in the spanning tree q_{mst} (Algorithm 4, Line 10). 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, it is updated in R.

Algorithm 1: SuMGra

```
1  Input: query graph q, data graph g, index I on g
2  Output: R: all the embeddings of q in g
3  C(u) = SelectCandidates(q,I)
4  q_mst = FindMST(q, C(u))
5  u_init = u_i ∈ q_mst | |C(u_i)|= argmin{|C(u_1)|, ..., |C(u_|q|)|}
6  R = ∅                                   /* Embeddings of query q in g */
7  for each v ∈ C(u_init) do
8      M_q = u_init                        /* Matched query vertices */
9      M_g = v /*                            Matched data vertices */
10     M = [M_q, M_g]                      /* Partial matching of q in g */
11     Update: R := SubgraphsSearch(R, M, C(u), q, g, q_mst)
12 return R
```

V. Indexing

This section describes the indexing structure that we have developed to efficiently store and query a multigraph. The idea

[1] a minimum spanning tree connects the entire graph with the least possible weights on its edges.

behind the indexing is to store the data graph in an efficient way, to perform faster subgraph querying.

Indexing includes (i) an extraction phase to obtain useful features from the data graph to optimize vertex indexing (ii) a query processing phase to select the set of candidate vertices from the data graph. The latter phase is implemented by the SELECTCANDIDATES procedure. For a lucid understanding of our indexing schema, we introduce a few definitions.

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	$\{\{E_1, E_3\}\}$

TABLE I: Vertex signatures for the data graph in Figure 2b

Definition 2: Vertex neighborhood signature. For a vertex v, the vertex neighbourhood signature $\sigma(v)$ is a multiset of edge dimensions, that contain the set of edges for all the dimensions that are incident on v. More formally, $\sigma(v) = \cup_{v_j \in N(v)} L_E(v, v_j)$ where $N(v)$ is the set of neighbourhood vertices of v and $\cup$ is the union operator for the multiset that allows repetition of the elements.

For instance, in Figure 2, $\sigma(v_6) = \{\{E_1, E_3\}, \{E_1\}\}$. The vertex signature is an intermediary representation that is exploited by our indexing schema. All the vertex neighborhood signatures of the vertices of the data graph in Figure 2 are written in Table I.

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

For instance, in Table III, the *candidate set* for vertex u_1 is given by $C(u_1) = \{v_3, v_4\}$. In the following sections, we discover the intermediate steps involved in obtaining this candidate set.

A. Offline Index Construction

We build two types of indexes offline: (i) Given the vertex signature of all the vertices of g, we construct the index for the multiset of edge dimensions by exploring some features of the signature σ (ii) We build a trie index for each of the signature for the entire data graph g.

1) Edge Dimension Index $\mathcal{I}$: The edge dimension index is built to organize the structural information supplied by the vertex neighbourhood signature of the data graph. Considering the multiset supplied by the neighbourhood signature, we can observe some interesting features that we can exploit. For example, in Table I, the *neighbourhood signature* of v_6, $\sigma(v_6) = \{\{E_1, E_3\}, \{E_1\}\}$ has two sets of dimensions in it and hence we only need to match it with query vertices that have at most two sets of items in their *neighbourhood signature*. Also, $\sigma(v_2) = \{\{E_2, E_3, E_1\}, \{E_1\}\}$ has a dimension set of maximum size 3 and hence a query vertex must have the maximum 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,

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

TABLE II: Synopses for all the data vertices in Figure1b

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 *neighbourhood signature*. For the synopses we propose the following six ($|f| = 6$) different useful features that we illustrate with the help of the vertex neighbourhood signature of v_3 ($\sigma(v_3) = \{\{E_2, E_3, E_1\}, \{E_1, E_3\}, \{E_1, E_2\}, \{E_1\}\}$):

- F_1 Cardinality of the neighbourhood signature, e.g., $F_1(v_3) = 4$.
- F_2 The number of unique dimensions in the neighbourhood signature, e.g., $F_2(v_3) = 3$.
- F_3 The number of all occurrences of the dimensions (includes repeated dimensions), e.g., $F_3(v_3) = 8$.
- F_4 Minimum index value of the dimensional alphabet (based on the position of the sequenced alphabet), e.g., $F_4(v_3) = 1$.
- F_5 Maximum index value of the dimensional alphabet (based on the position of the sequenced alphabet), e.g., $F_5(v_3) = 3$.
- F_6 Maximum cardinality of the neighbourhood signature, e.g., $F_6(v_3) = 3$.

Table II lists the synopses for all the vertices from data graph g shown in Figure 2.

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. To this end, we should be able to represent the synopses with an efficient data structure. Since each vertex is represented by a synopses of several fields, a data structure that helps in efficiently performing range search for multiple dimensions would be an ideal choice. For this reason, we build a d-dimensional R-tree, whose nodes are the synopses with f fields, where a synopses is nothing but the surrogate representation of the vertices of the multigraph.

The general idea of using an R-tree structure is as follows: A synopses of a data vertex spans an axes-parallel rectangle in an f-dimensional space, where the maximum co-ordinates of the rectangle being the values in the synopses itself, and the minimum co-ordinates being the origin of the rectangle (zero values). For example, a two dimensional synopses of a data vertex $S_v = (2, 3)$ 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, $S_u^1 = (1, 3)$ and $S_u^2 = (1, 4)$, it is not difficult to observe that the rectangle spanned by S_u^1 is wholly contained in the rectangle spanned by S_v but S_u^2 is not

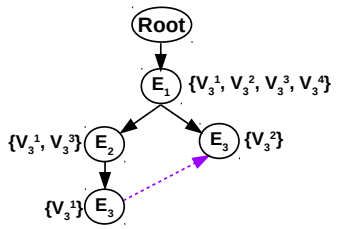


Fig. 3: Trie structure for the signature of data vertex v_3

wholly contained in S_v. More formally, the possible candidate for vertex u can be written as $\mathcal{P}(u) = \{v | \forall_{i \in [1,\ldots,f]} S_u(i) \leq S_v(i)\}$, 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 $\mathcal{I}$, where each synopses is treated as an f-dimensional leaf of the R-tree.

2) Trie Index $\mathcal{T}$: Since the previous indexing schema enables us to select the valid candidate set $C(u)$ in an approximate manner, we build a separate Trie index for every data vertex $v \in g$, in order to obtain the exact valid candidate set $C(u)$. Here we represent the vertex neighbourhood signature $\sigma(v)$ by a tree data structure, where the nodes of the tree are represented by the edge dimensions. In order to achieve this, we take inspiration from [19] to propose our approach - Ordered Trie with Inverted List (OTIL). In our context, we treat each sub-signature v^i (e.g., $\{E_2, E_3\}$) as a word that can be inserted into the trie structure. In addition, each v^i in itself has ordered edge dimensions in it. For example in Table I, the sub-signature of v_3^1 is $\{E_2, E_3, E_1\}$, and is ordered as $\{E_1, E_2, E_3\}$. This ordering is universally maintained for the sub-signatures of both $\sigma(u)$ and $\sigma(v)$. Further, we construct a *vocabulary* (a set of distinct edge dimensions) for each $\sigma(v)$, and build an *inverted list* for each edge dimension E_i that contains the identifiers of the sub-signatures v^i. For example, for vertex v_3, as shown in Figure 3, the edge E_2 will contain the list $\{v_3^1, v_3^3\}$, since E_2 is present in the sub-signatures v_3^1 and v_3^3.

To construct the trie as shown in Figure 3, we insert all the ordered sub-signatures of the data vertex v at the root of the trie. The trie-nodes are nothing but the edge dimension themselves. As and when a new sub-signature is added to the trie, the nodes in the trie maintain a list of sub-signature identifiers. It is to be noted that the leaves with identical edge dimension (e.g., E_3) are internally connected and thus form a linked list of sub-signature identifiers, which offers a compact representation and faster querying. For instance, consider the trie structure for node v_3, as shown in Figure 3, and the query vertex u_3 with its signature $\sigma(u_3) := \{\{E_1, E_3\}, \{E_2\}, \{E_1\}\}$, where $u_3^1 = \{E_1, E_3\}$, $u_3^2 = \{E_2\}$ and $u_3^3 = \{E_1\}$. Although querying u_3^2 and u_3^3 is straightforward, querying u^1 requires the linkage between the trie-nodes that represent E_3, in order to correctly output the superset of sub-signatures $\{v_3^1, v_3^2\}$. In this way, we build an ordered trie with inverted list (OTIL) for each data vertex and call it as a trie index $\mathcal{T}$.

It is to be noted that both the edge dimension index $\mathcal{I}$ and trie index $\mathcal{T}$ are constructed offline and hence we can afford to invest time on it. Once constructed, both these indexes can be used for subgraph query processing for any type of query graph.

u	$\mathcal{P}(u)$	$C(u)$
u_1	$\{v_2, v_3, v_4, v_5, v_6\}$	$\{v_3, v_4\}$
u_2	$\{v_2, v_3, v_4, v_5, v_6\}$	$\{v_3, v_5\}$
u_3	$\{v_3, v_5\}$	$\{v_3, v_5\}$
u_4	$\{v_1, v_2, v_3, v_5, v_6, v_7\}$	$\{v_1, v_3, v_6\}$

TABLE III: Candidate set of vertexes during indexing

B. Query Processing for Candidate Selection

For each query vertex, the query processing procedure exploits the index $\mathcal{I}$ to retrieve the set of candidates data vertices. Query processing for the selection of candidates is achieved in two stages. Initially, we perform an *approximate selection* of candidates using the edge index $\mathcal{I}$, thereby partially discarding invalid candidates. Then, we exploit the trie index $\mathcal{T}$ to perform the *exact selection* of the valid candidate vertices, by pruning all the invalid candidate vertices.

1. Approximate selection: During this step, we retrieve at least all the valid candidate vertices from the data graph by exploiting the edge dimension index $\mathcal{I}$. Thus, we might have the candidate vertices that can be pruned further.

Lemma 1: Querying the edge index $\mathcal{I}$ constructed on 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 neighbourhood 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, 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. ■

To find candidate vertices for u_1, we build the synopses for u_1 and find the matchable vertices in g using the index $\mathcal{I}$. As we can recall, every synopses representation of the data vertices span a rectangle in the d-dimensional space. Thus, it remains to check, if the rectangle spanned by u_1 is contained in any of rectangles spanned by the synopses of the data vertices, with the help of R-tree built on data vertices. This results in the candidate set $\mathcal{P}(u_1) = \{v_2, v_3, v_4, v_5, v_6\}$, as shown in Table III. It should be noted that our goal is to find only the valid candidate matches for the query vertex u. We underline that $\mathcal{P}(u)$ contains the entire set of candidate vertices that matches u but, it may still contain false positive matches. Thus we need to prune all the invalid candidate vertices that do not match with u in order to obtain the exact candidate set $C(u)$.

2. Exact selection: In this step, we obtain the exact validate candidate set $C(u)$ for the query vertex u, by pruning all the

invalid candidates. To perform exact candidate selection, we use the partial candidate set $\mathcal{P}$.

As we recall from Table I, each data vertex is associated with the vertex neighbourhood signature. To exploit this information, we propose an approach to efficiently find the possible candidate set $C(u) = \{v \in g | \sigma(u) \subseteq \sigma(v)\}$. Since we are looking for a superset of a multi-set, the setting can get quite tricky.

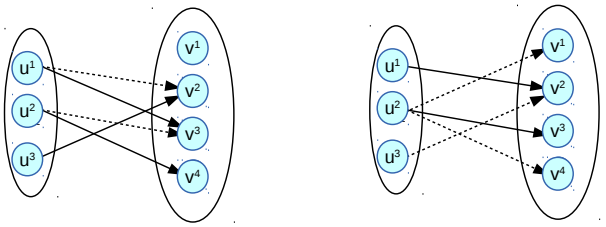
Let us consider a query vertex u represented by the vertex neighbourhood signature $\sigma(u) = \{u^1, u^2, \ldots, u^m\}$ and a possible candidate data vertex v represented by the signature $\sigma(v) = \{v^1, v^2, \ldots, v^n\}$, where u^i and v^i are the sub-signatures of $\sigma(u)$ and $\sigma(v)$ respectively. To check if the data vertex v is a valid candidate of query vertex u, we have to verify if $\sigma(u) \subseteq \sigma(v)$. In order to find the superset of the multiset (vertex neighbourhood signature) in an efficient manner, we now exploit the trie index $\mathcal{T}$.

Whenever a query vertex u is to be evaluated for its candidate vertex v, for each of the query sub-signature, we perform prefix search on the index $\mathcal{T}$ that we have built. After this evaluation, we have the knowledge about the existence of a sub-signature superset in v for all the sub-signatures in u. If there is at least one query sub-signature, that has not found a superset, then the possible candidate vertex v can safely be pruned. However, even in the case when all the query sub-signatures have found a superset, we cannot yet claim that v is a candidate vertex, and hence we offer a further treatment to find the valid candidates.

In order to have a lucid understanding of a valid candidate, let us consider a scenario where a data vertex v is being verified to be a valid candidate for u, as shown in Figure 4. It is not hard to observe that more than one query sub-signatures can compete for a unique data sub-signature. For example, in Figure 4a, query sub-signature u^1 and u^3 are the subsets of a unique data sub-signature v^2. Thus, in order to obtain a valid matching, we propose to build a bipartite graph that has an edge (u^i, v^j) if and only if $u^i \subseteq v^j$. For the ease of understanding, we introduce the notion a valid maximum matching.

Definition 4: Valid Maximum Matching. For a bipartite graph with m and n nodes, we define the valid maximum matching as a maximum matching where all the m nodes have been matched.

Now, for the considered bipartite graph, the sub-signatures of both u and v represent the vertices of the bipartite graph, with m nodes and n nodes respectively. Once all the possible edges have been added to the bipartite graph, we run the maximum matching algorithm to check if for every query sub-signature u^i there exists a unique superset data sub-signature v^j that it matches to. It is worthy to observe that Figure 4a has a valid matching (marked in solid lines) since, there exists a maximum matching that includes all the query vertices. Where as the maximum matching in Figure 4b is not a valid matching since the maximum matching can not include all the query vertices (in this case either u^1 or u^3 is excluded). To solve the maximum matching problem on the bipartite graph, we employ the *Hopcroft-Karp* [7] algorithm. This algorithm has a worst case time complexity of $O(|E|\sqrt{|V|})$, where E and V are the edges and nodes of the bipartite graph.



(a) A valid matching (b) An invalid matching

Fig. 4: A scenario of maximum bipartite matching with $u = \{u^1, u^2, u^3\}$ and $v = \{v^1, v^2, v^3, v^4\}$

At this stage, the candidate set $C(u)$ is available for all the vertices of the query graph, and $C(u)$ contains the exact possible candidate vertices that are returned by the SELECTCANDIDATES procedure in Algorithm 1.

VI. SUBGRAPH QUERY PROCESSING

Now that all the valid candidates for each of the query vertex have been found, we proceed further with subgraph query processing. In order to find the embeddings of a query graph, we not only need to find the valid candidates, but also find the structure of the query graph to be matched. In this section, we discuss in detail about the various procedures involved in Algorithm 1.

A. Choosing the Initial Vertex

It is argued that an effective way of choosing the initial query vertex improves the efficiency of subgraph querying [10]. Once we have the selected candidates $C(u)$ for all the query vertices, we choose the initial vertex u_{init} that has the least number of possible candidates. Following the strategy proposed in [15], we build a spanning tree for the query subgraph. For every edge in q we assign a weight, which is the minimum value of the size of the candidate set that corresponds to the pair of the vertices it belongs to (Algorithm 2). Thus we transform the unweighted query graph to a weighted query graph, and use MST algorithm (Prim's algorithm) to output the minimum spanning tree q_{mst}.

Algorithm 2: FINDMST$(q, C(u))$

```
1 W = ∅
2 for (u_i, u_j) ∈ E(q) do
3 |   w(u_i, u_j) = min(|C(u_i)|, |C(u_j)|)
4 |   W[(u_i, u_j)] = w(u_i, u_j)
5 q_mst = MST(W, q)  /* Apply Prim's algorithm */
6 return q_mst
```

The minimum spanning tree q_{mst} of the query graph q plays a vital role in maintaining the structure of the matchable data pattern that recursively grows during the subgraph search procedure. Also, the minimum spanning tree helps in reducing the search space for the solution tree while running the recursive subgraph search procedure.

Algorithm 3: SUBGRAPHSEARCH(R,M,$C(u)$,q,g,q_{mst})

```
1  INPUT: M = [M_q, M_g]: partial matching of q in g, q: query graph, g: data graph, C: candidate set for q, q_mst: spanning tree of q
2  u_n = NEXTQUERYVERTEX(M_q, C(u), q_mst)
3  M_C = FINDJOINABLE(C(u_n), M_q, M_d, q, d, u_n)
                /* Matchable candidate vertices */
4  if M_C ≠ ∅ then
5      for v_n ∈ M_C do
6          M_q = M_q ∪ u_n; M_g = M_g ∪ v_n
7          M = [M_q, M_g]          /* Partial matching grows */
8          SUBGRAPHSEARCH(R, M, C(u), q, g, q_mst)
9          if (|M| == |q_mst|) then
10             R = R ∪ M          /* Embedding found */
11 return R
```

B. Subgraph Searching

The process of SUBGRAPHSEARCH is described in Algorithm 3. Once an initial query vertex u_{init} is chosen and its corresponding matchable data vertex is chosen from the set of select candidates $C(u_{init})$, we have the partial solution pair of the pattern we want to grow. To proceed further with the pattern matching, we have to choose the next query vertex for which we run the NEXTQUERYVERTEX procedure as depicted in Algorithm 4. We choose the next query vertex dynamically, as and when the query pattern grows. During every iteration of the SUBGRAPHSEARCH procedure, we collect all the $Frontier$ query vertices from the minimum spanning tree q_{mst}, that have not been considered so far (Line 1), which are the possible next query vertices to be chosen. Now, among the possible $Frontier$ query vertices, we choose the one with minimal number of select candidates, as a next query vertex to be matched. With this approach we can guarantee the structural properties of the query graph to be matched.

Algorithm 4: NEXTQUERYVERTEX($M_q, C(u), q_{mst}$)

```
1  Frontier := {u_i ∈ (V(q_mst) \ M_q) | ∃u_j (u_i, u_j) ∈ E(q_mst) ∧ u_j ∈ M_q}
2  u_n := argmin_{u_i ∈ Frontier} |C(u_i)|
3  return u_n
```

The crucial step in the subgraph search algorithm is the FINDJOINABLE procedure (Line 3). For the chosen next query vertex u_n to be matched, we have a set of possible candidate matches $C(u_n)$. Since $C(u_n)$ has all the candidates that can be matched with u_n, we have to guarantee that any $v \in C(u_n)$, which is the partial solution, maintains the structure of the query graph. To find the solutions with the valid structure, we call the FINDJOINABLE procedure, as described by Algorithm 5. The FINDJOINABLE procedure is meant to perform two vital operations: (i) it maintains the structural properties of the solution, (ii) it verifies the correctness of edge dimensions.

To maintain the structural property of the solution, we find the set A_q that contains all the matched query vertices which are adjacent to the next vertex u_n to obtain the corresponding set A_d that contains the matched data vertices that correspond to A_q (Line 1-2). Now for each $v_n \in C(u_n)$ and $v \in A_d$, we check if there exists an edge in the data graph (Line 5). This ensures us that the possible data vertex v_n is a valid match, since there exists an edge that maintains the structural connectivity with the previously matched data vertices. It is to be observed that, since we are growing the pattern from a single vertex, we only need to check the structural connectivity with only the matched vertices that are adjacent to the next matchable vertex. Now, in order to verify the correctness of the edge dimensions, we check if the edge label of the data edge $L(v, v_n)$ is a superset of the edge label of the query edge $L(u, u_n)$ (Line 6). If such is the case, then the candidate vertex v_n is added to the set of matchable candidate vertices M_C.

Algorithm 5: FINDJOINABLE($C(u_n), M_q, M_d, q, d, u_n$)

```
1  A_q := M_q ∩ adj(u_n)
2  A_d ⊆ M_d | [A_q, A_d] ⊆ [M_q, M_d]
3  for v_n ∈ C(u_n) do
4      for v ∈ A_d do
5          if ∃(v, v_n) ∈ E(d) then
6              if L(v, v_n) ⊇ L(u, u_n), where u ∈ [u, v] then
7                  add v_n to M_C
8  return M_C
```

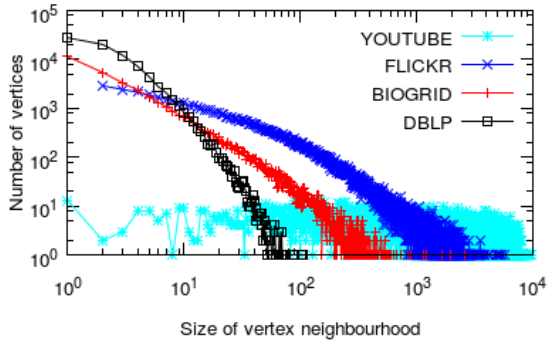
FINDJOINABLE returns all the possible data vertices that can contribute towards the partial solution. As the partial solution is updated for each $v_n \in M_C$ (Line 7), the SUBGRAPHSEARCH procedure is called recursively (Line 8), to further grow the partial solution. Once the entire embedding has been found, it is updated to the set of resultant solutions R (Line 9-10).

Lemma 2: The procedure FINDJOINABLE guarantees to retain the structure of the embeddings.

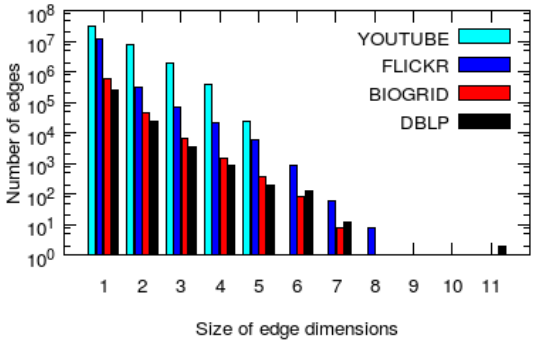
Proof: Consider a query q of size $|U|$. For $n = 1$, let the first matching M_d^1 corresponding to the initial query vertex M_q^1. Now, A_q and A_d contain all the adjacent vertices of the previously matched vertices M_q^1 and M_d^1 respectively, thus maintaining the connectivity with the partially matched solution M. Hence for $n > 1$, by induction, the structure of entire embedding (that corresponds to the query graph) is retained. ■

VII. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of SUMGRA for real and synthetic multigraphs. Since this work is supposedly the first work that addresses multigraphs, comparison with any other approaches is hardly possible. Hence, we deploy various benchmarks to portray the efficiency and suitability of our algorithm. All the experiments were run on a 64-bit Intel Core i7-4900MQ @ 2.80GHz, with 32GB memory, running Linux OS - ubuntu 14.04 LTS. Our methods have been implemented using C++.



(a) Number of vertices Vs size of vertex neighbourhood



(b) Number of edges Vs size of edge dimensions

Fig. 5: Statistics on distribution of real data sets

A. Description of Datasets

To validate the correctness, efficiency and versatility of SUMGRA, we consider four real world datasets that span over biological and social network data. Further, to test the scalability of our approach, we consider a synthetic data set. All the multigraphs considered in this work are undirected and they do not contain any attribute on the vertex. Table IV offers a quick description of all the characteristics of the data sets.

1) Real Datasets: For our experimental analysis, we consider four real world data sets: *DBLP* [2] data set is built by following the procedure adopted in [1]. In this graph the vertices correspond to different authors and the dimensions represent the 50 conferences in Computer Science having the most number publications. Two authors are connected over a dimension if they co-authored more than one paper together in that conference. *BIOGRID* [3] dataset [2] is a protein-protein interactions network, where nodes represent proteins and the edges represent interactions between the proteins. The data set has 7 unique edge dimensions which correspond to the 7 different types of interactions between a pair of proteins. *FLICKR* [4] dataset has been crawled from Flickr, which is an image and video hosting website, web services suite, and an online community. In this data set, the users are represented by nodes, and the blogger's friends are represented using edges (since edge network is the friendship network among the bloggers). In addition, the data set represents the friendship network based on the group memberships (195 in number), which we represent as edge dimensions. Thus multiple edges exist between two users if they have common multiple memberships. *YOUTUBE* dataset [18] treats users as the nodes and the various connections among them as milti-edges. The edge information includes the contacts, mutual-contact, co-subscription network, co-subscribed network: two users are connected if they are both subscribed by the same user and favorite network (two users are connected if they share favourite videos).

To analyse the results, for all the real graphs, we provide the vertex neighbourhood distribution as well as the edge dimension distribution as seen in Figure 5. The logarithmic distribution of the number of vertices with the increasing size of vertex neighbourhood is plotted in Figure 5a, while the logarithmic distribution of the number of edges with increasing size of edge dimensions is plotted in Figure 5b.

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
SYNTH	500,000	25,000,000	20	2.0e-4	50	1.15

TABLE IV: Statistics of datasets; Dim = number of dimensions; A_{deg}=Average vertex degree; A_{dim} = Average no. of dimensions/ edge

Referring to Figure 5a and Table IV, we can 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* and *DBLP* 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 *DBLP*, the distribution shifts towards the smaller neighbourhood size.

Referring to Figure 5b (where the number of edges are on a logarithmic scale), we observe that most of the edges, of all the dataset, lie only over one dimension ($d = 1$). *YOUTUBE* has a fairly good distribution of the edges over all the five edge dimensions, and hence has a fairly large average dimension A_{dim} = 1.321. *DBLP* data set has the maximum size of edge dimensions (some of the edges have eleven dimensions).

2) Synthetic Dataset: We generate the synthetic graph using Erdos Renyi (ER) model, which is a classical random graph generator model. We generate 20 random graphs each with 500 000 nodes. For each node, we set the average degree to 50. Each random graph represent a different edge dimension. The resulting synthetic multigraph has 25 million multi-edges. We name this multigraph as *SYNTH*.

[2]http://www.dblp.org/db/
[3]http://thebiogrid.org/
[4]http://socialcomputing.asu.edu/pages/datasets

B. Description of Query Graphs

To test the behavior of our approach, we generate *path*, *subgraph* and *clique* queries, as already done for standard subgraph querying [6], [15]. For real data sets, although we could generate path and subgraph queries of size 3,5,7,9 and 11, we could only find clique queries of size 3,5,7 and 9. Since it is hard to inject cliques into the synthetic graphs [13], we generate only path and subgraph queries. All the generated queries contain an edge with at least two dimensions.

For our experiments, we generate 1 000 samples for each of the aforementioned queries. Following the methodology previously proposed for subgraph query matching algorithms [5], [13], we report the average time values over 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 [24], [6], [13]). In order to have a clear picture of the different component of our approach, we output both the time required to obtain the select candidate set $C(u)$ as well as the time required to perform SUBGRAPHSEARCH procedure.

C. Baseline Approaches

To perform a valid comparison with SUMGRA, we derive the following base line approaches.

No Synopses: To test the performance of edge indexing, we conduct experiments without constructing the edge index. In this approach (NoSyn), approximate candidate set $\mathcal{P}(u)$ is not computed, and hence no data vertex is pruned. Thus, all the data vertices are eligible to be the possible candidates, and thence used for exact matching.

Bit vector approach: In order to perform exact matching, we need to perform a superset query on the vertex signature $\sigma(v)$, so that we can obtain only the valid candidate vertices $C(u)$. To compete against the OTIL approach, we propose a naive but efficiently implementable bit vector (BIT) approach, to perform superset query on the multiset.

In BIT, we represent the set of edge dimensions in the vertex neighbourhood signature σ by a bit vector. To do this, we predefine the size of the bit vector and set the positional values to 1, if the edge dimension exists. For example, in Table I, we can set the size of the bit vector to 4 and hence for v_3, $\sigma^b(v_3) = \{\{1,0,1,0\},\{1,1,1,0\},\{1,0,0,0\},\{1,1,0,0\}\}$.

Let us now consider a query vertex u represented by bitset signature $\sigma^b(u) = \{u^1, u^2, \ldots, u^m\}$ and a possible data candidate vertex v represented by bitset signature $\sigma^b(v) = \{v^1, v^2, \ldots, v^n\}$, where u^i and v^i are the bitset representations of edge dimensions whose elements are set to 1 if the edge dimension exists, else set to 0, as described before.

For each pair of u and v we maintain a subset bipartite graph (SBG), whose nodes are $u^1, u^2, \ldots, u^m$ and $v^1, v^2, \ldots, v^n$ respectively. We perform maximum matching on this SBG, as already explained in Section V-B, to obtain the valid candidate set $C(u)$.

At this point we propose three baseline approaches to compare with SUMGRA, in order to demonstrate the performance of our proposed method. In particular we coupled synopses based (Syn) and no synopses based ($NoSyn$) strategies with the two different ways of performing the exact matching procedure ($OTIL$ and BIT):

A. Syn+OTIL : SUMGRA (using edge indexing for approximate matching and OTIL for exact matching)

B. Syn+BIT: Using edge indexing for approximate matching and bit vector for exact matching

C. NoSyn+OTIL: Only exact matching with OTIL is used

D. NoSyn+BIT: Only exact matching with bit vector is used

D. Time performance of SUMGRA

In Section V, we gave emphasis on constructing the indexes for edge dimensions in order to extract the approximate candidate set, and then introduced the *OTIL* approach to extract the exact candidate set $C(u)$. We recall that SUMGRA constructs both $\mathcal{I}$ and $\mathcal{T}$ offline. These indexes are explored during the query processing in order to retrieve the valid candidate set $C(u)$. In Table V we report the indexing construction time of SUMGRA for each of the employed dataset. We can observe that for sparse graphs as *DBLP* and *BIOGRID* the most costly operation is the construction of the $\mathcal{I}$ index while for the other datasets (*FLICKR* and *YOUTUBE*) the most costly offline step is the construction of the $\mathcal{T}$ index. This behavior is reasonable owing to the high density in both edges and dimensions that has a bigger impact on the *OTIL* construction. To conclude, we can highlight that the offline step is not so time consuming as in the worst case, for the *SYNTH* multigraph, we need less than 2 minutes to index the 500 000 nodes and 25 million edges.

Data set	Edge Dimension Index $\mathcal{I}$ Time (seconds)	Trie Index $\mathcal{T}$ Time (seconds)
DBLP	4.67	0.3076
BIOGRID	1.86	0.5433
FLICKR	8.93	10.7068
YOUTUBE	12.15	41.176
SYNTH	43.78	49.2236

TABLE V: Time taken to construct the offline index

1) Query Processing Time: Figure 6 summarises the time performance of SUMGRA. Every bar in the histogram represents two values: the time taken to select valid candidates and the time taken to perform sub-multigraph search, which together constitute the time required to output the embeddings.

It is interesting to observe from Figure 6 that, for *DBLP* and *BIOGRID*, the time required to output the candidate set $C(u)$ is the least when we adopt indexing, irrespective of *BIT Vector* or *OTIL*. Also, owing to the relatively smaller size and lesser density of the graph, the subgraph matching time requires more time when compared to the select candidate time, except for the 'NoIndex+OTIL' setting. Also, for these two data sets, the baseline approach of *BIT Vector* performs reasonably well compared to the *OTIL* approach that we introduce to manage the exact match procedure.

On the other hand, *FLICKR* and *YOUTUBE* data sets behave quite differently, when compared to the other two datasets. For these two datasets, the time required to obtain $C(u)$ is larger than the time required to find the embeddings. Since these datasets are large and comparatively denser than

the previous multigraphs, the time required to obtain $C(u)$ is quite noticeable. But the subgraph matching time is relatively lesser when compared to the time required to obtain $C(u)$. We can observe that, for these kinds of data, SUMGRA ($Syn + OTIL$) clearly outperforms all the other baseline approaches. This is particularly evident on the *YOUTUBE* dataset where we obtain the maximum gain in terms of time performance.

As a general conclusion, we can state that SUMGRA obtains remarkable improvements w.r.t. the base line approaches for huge and very dense multigraphs (considering both edges and dimensions) while it achieves comparable time performance with the baseline approaches on *DBLP* and *BIOGRID*.

E. Performance of SUMGRA with varying dimensions

In this section we analyse the time performance of SUMGRA by varying the number of edge dimensions in the query graph. 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 $d = 4$.

Due to the difficulty in finding multigraph queries with $d = 4$ we can only use *BIOGRID* and *YOUTUBE* datasets as we are able to obtain 1 000 queries for each kind of query (path, subgraph and clique) with one or more edges that lie over four edge dimensions ($d = 4$). In spite of the limited number of queries for *DBLP* and *FLICKR*, we witness the similar behavior for all the datasets. Hence, we report the results only for *BIOGRID* and *YOUTUBE* datasets as shown in Figure 7. From almost all the plots, we observe that queries with more dimensions ($d = 4$) are, generally, faster to process since they are richer with information and, because of that, they can get quickly selective.

F. Scalability of SUMGRA

To verify the scalability of SUMGRA, we conduct experiments on the synthetic graph previously introduced. *SYNTH* has 500 000 nodes, 25 000 000 edges span over 20 dimensions. In Figure 8, we report the time required to output the first 1 000 embeddings for path and subgraph queries with d=2 and d=4. We can observe that, also on this huge graph, SUMGRA is able to answer to such queries in a reasonable amount of time and this fact underlines the scalability and the robustness of the proposed approach.

VIII. DISCUSSION AND POSSIBLE EXTENSIONS

We now briefly summarize the experimental findings and highlight the possible extension of our work.

A. Synopsis vs No Synopsis

With the help of the experiments conducted, we observe that the synopsis based representation, introduced in Section V, is capable of improving the performance of the multigraph query procedure. This is observed for all the real datasets we employed. This evaluation underlines the effectiveness of our proposed vertex feature representation to speed up the discovery of the valid candidate set.

B. OTIL vs BIT vector

The SELECTCANDIDATES procedure, for some multigraphs, takes most of the total amount of processing time. This phenomena is particularly true for *FLICKR* and *YOUTUBE* data. One crucial point of this operation being the exact selection step. In our experiments, we evaluated two different strategies (*OTIL* and *BIT vector*) and we observe that no single strategy is always superior to the other. What we can note is that this step is influenced by the multigraph characteristics. However, we can observe that for multigraphs with higher average degree (A_{deg}) and higher average edge dimensions (A_{dim}) the *OTIL* approach clearly outperforms the *BIT vector* method. For more sparse data sets, in terms of both graph topology and edge dimensions, we can note that both *OTIL* and *BIT vector* have almost equal performance and the difference is less pronounced.

C. Attributed vs Non-Attributed Multigraphs

Attributed Multigraphs are multigraphs where one or more attributes (or items) are associated with each node. In this work we concentrate our effort on Non-Attributed multigraphs as we focused our work to the problem of how to manage different kinds of edges (dimensions) between nodes that, from the best of our knowledge, was not previously addressed by any of work in literature. For the simple graphs, previous works already deal with node attributes [22] and it will be straightforward to re-use those ideas directly in order to manage the attributed multigraphs with our framework. In particular, the previous techniques can be incorporated at the level of select candidates procedure in order to reduce the possible select candidate set. Thus, the data vertices that meet the requirements of both the vertex attributes and edge dimensions will only be eligible as possible select candidate set. The additional information in the form of vertex attributes might be helpful in reducing the search space for the subgraph matching procedure. Addressing attributed multigraphs can be a possible extension of our work.

IX. CONCLUSION

We proposed an efficient algorithm SUMGRA that can perform subgraph matching on multigraphs. The main contributions included the construction of indexing for the edge dimensions to prune the possible candidates, followed by building an ordered tree with inverted lists to retain only the valid candidates. Then we proposed an efficient subgraph search procedure that can very well work on the multigraphs. Various experiments were conducted to demonstrate the efficiency, versatility, and scalability of our approach.

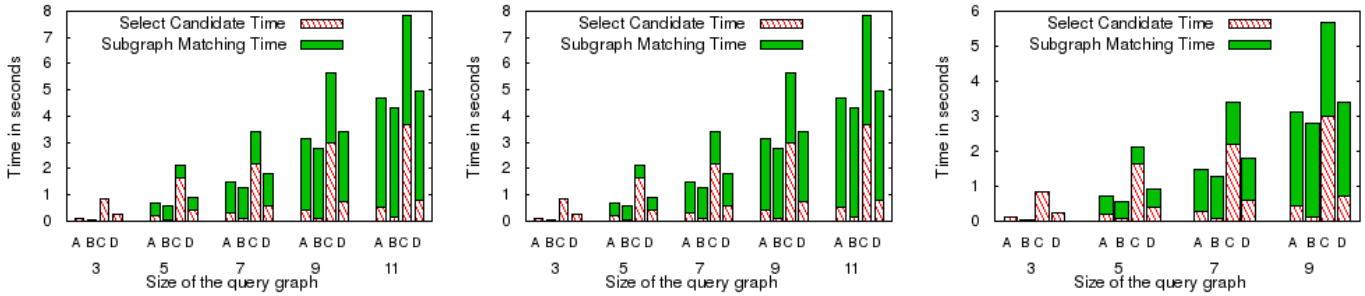
REFERENCES

[1] B. Boden, S. Günnemann, H. Hoffmann, and T. Seidl. Mining coherent subgraphs in multi-layer graphs with edge labels. In *KDD*, pages 1258–1266, 2012.

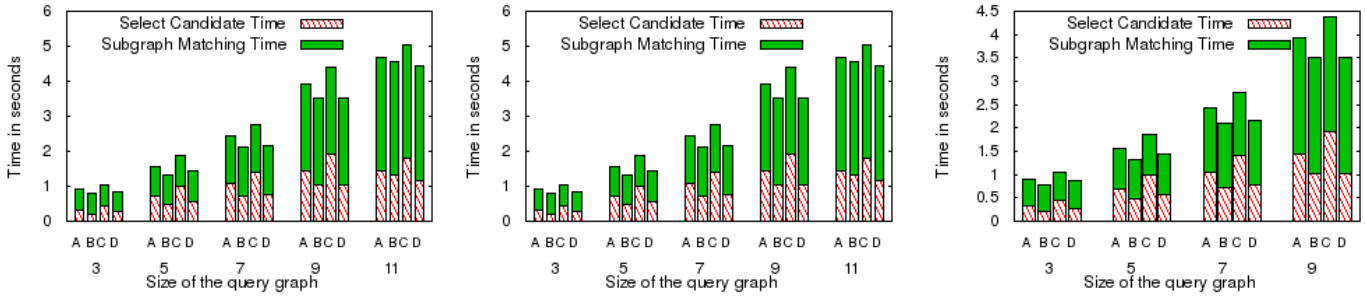
[2] F. Bonchi, A. Gionis, F. Gullo, and A. Ukkonen. Distance oracles in edge-labeled graphs. In *EDBT*, pages 547–558, 2014.

[3] J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: towards verification-free query processing on graph databases. In *SIGMOD*, pages 857–872. ACM, 2007.

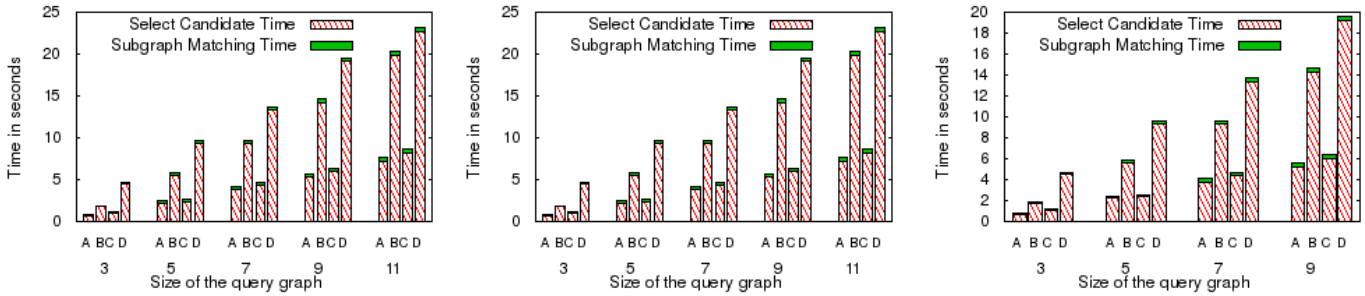
[4] L. P Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE TPAMI*, 26(10):1367–1372, 2004.



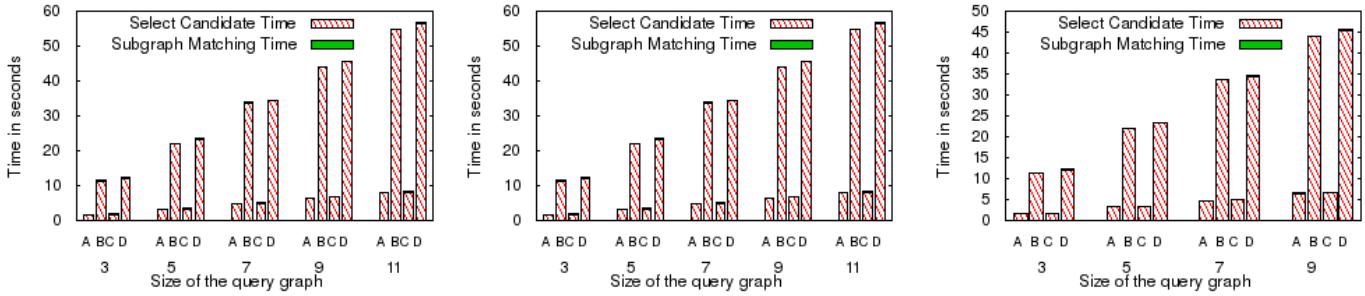
(a) *DBLP*: Subgraph Query (b) *DBLP*: Path Query (c) *DBLP*: Clique Query



(d) *BIOGRID*: Subgraph Query (e) *BIOGRID*: Path Query (f) *BIOGRID*: Clique Query



(g) *FLICKR*: Subgraph Query (h) *FLICKR*: Path Query (i) *FLICKR*: Clique Query



(j) *YOUTUBE*: Subgraph Query (k) *YOUTUBE*: Path Query (l) *YOUTUBE*: Clique Query

Fig. 6: Time performance of the proposed method compared to the base lines approaches: A = $Syn + OTIL$ (SUMGRA), B = $Syn + BIT$, C = $NoSyn + OTIL$, D = $NoSyn + BIT$

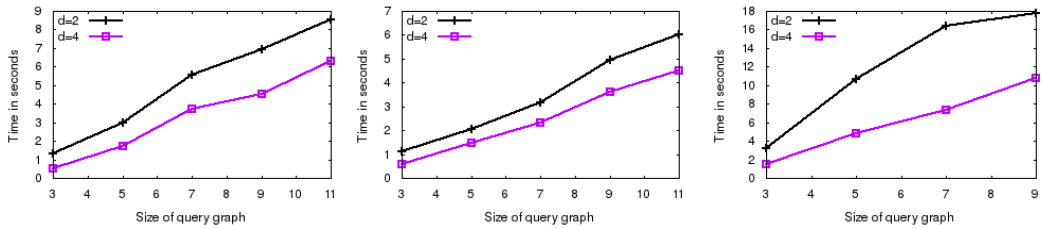
[5] W.-S. Han, J. Lee, and J.-H. Lee. Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD*, pages 337–348. ACM, 2013.

[6] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*, pages 405–418. ACM, 2008.

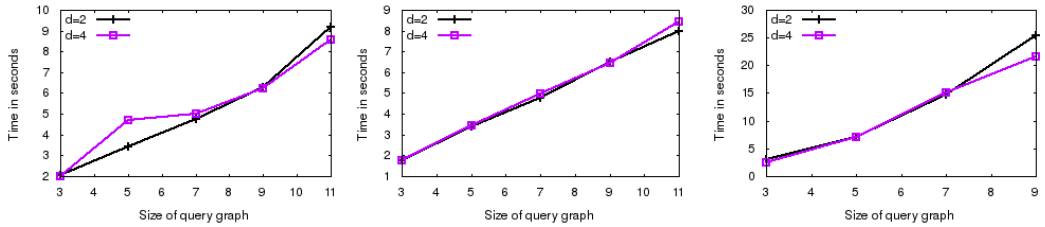
[7] J. E. Hopcroft and R. M. Karp. An n^5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–231, 1973.

[8] R. Jin, H. Hong, H. Wang, N. Ruan, and Y. Xiang. Computing label-constraint reachability in graph databases. In *SIGMOD*, pages 123–134, 2010.

[9] S. Kim, I. Song, and Y. J. Lee. An edge-based framework for fast

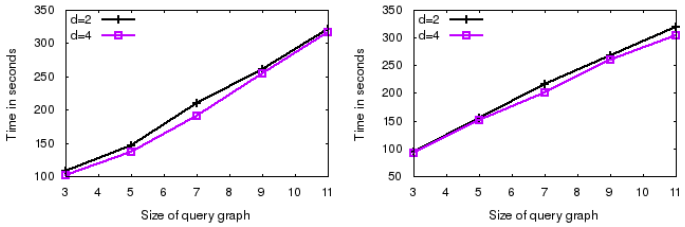


(a) *BIOGRID*: Subgraph Query (b) *BIOGRID*: Path Query (c) *BIOGRID*: Clique Query



(d) *YOUTUBE*: Subgraph Query (e) *YOUTUBE*: Path Query (f) *YOUTUBE*: Clique Query

Fig. 7: Performance of SUMGRA with increasing dimensions: $d = 2$ and $d = 4$



(a) *SYNTH*:Subgraph Query (b) *SYNTH*:Path Query

Fig. 8: Performance of SUMGRA for *SYNTH* data set with $d = 2$ and $d = 4$

subgraph matching in a large graph. In *Database Systems for Advanced Applications*, pages 404–417. Springer, 2011.

[10] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In *PVLDB*, pages 133–144. VLDB Endowment, 2012.

[11] J. Leskovec, A. Singh, and J. Kleinberg. Patterns of influence in a recommendation network. In *PAKDD*, 2006.

[12] L. Libkin, J. Reutter, and D. Vrgoč. Trial for rdf: adapting graph query languages for rdf data. In *PODS*, pages 201–212. ACM, 2013.

[13] Z. Lin and Y. Bei. Graph indexing for large networks: A neighborhood tree-based approach. *Knowledge-Based Systems*, 2014.

[14] J. J. McAuley and J. Leskovec. Learning to discover social circles in ego networks. In *NIPS*, pages 548–556, 2012.

[15] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 1(1):364–375, 2008.

[16] D. Shasha, J. TL Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, pages 39–52. ACM, 2002.

[17] L. Tang, X. Wang, and H. Liu. Community detection via heterogeneous interaction analysis. *Data Min. Knowl. Discov.*, 25:1–33, 2012.

[18] L. Tang, X. Wang, and H. Liu. Community detection via heterogeneous interaction analysis. *Data Min. Knowl. Discov.*, 25(1):1–33, 2012.

[19] M. Terrovitis, S. Passas, P. Vassiliadis, and T. Sellis. A combination of trie-trees and inverted files for the indexing of set-valued attributes. In *CIKM*, pages 728–737. ACM, 2006.

[20] Julian R Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.

[21] X. Yan, P. S Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *SIGMOD*, pages 335–346. ACM, 2004.

[22] J. Yang, S. Zhang, and W. Jin. Delta: indexing and querying multi-labeled graphs. In *CIKM*, pages 1765–1774. ACM, 2011.

[23] A. Zhang. Protein interaction networks: Computational analysis, 2009.

[24] P. Zhao and J. Han. On graph query optimization in large networks. *PVLDB*, 3(1-2):340–351, 2010.

[25] P. Zhao, J. X. Yu, and Philip S Yu. Graph indexing: tree+ delta¡= graph. In *PVLDB*, pages 938–949. VLDB Endowment, 2007.