G-Finder: Approximate Attributed Subgraph Matching

Lihui Liu*, Boxin Du*, Jiejun Xu† and Hanghang Tong*
*Department of Computer Science, University of Illinois at Urbana Champaign
†HRL Laboratories, LLC.
Email: *lihuil2@illinois.edu, *boxindu2@illinois.edu, †jxu@hrl.com, *htong@illinois.edu

***Abstract*—Subgraph matching is a core primitive across a number of disciplines, ranging from data mining, databases, information retrieval, computer vision to natural language processing. Despite decades of efforts, it is still highly challenging to balance between the matching accuracy and the computational efficiency, especially when the query graph and/or the data graph are large. In this paper, we propose an index-based algorithm (G-FINDER) to find the *top-k* approximate matching subgraphs. At the heart of the proposed algorithm are two techniques, including (1) a novel auxiliary data structure (LOOKUP-TABLE) in conjunction with a neighborhood expansion method to effectively and efficiently index candidate vertices, and (2) a dynamic filtering and refinement strategy to prune the false candidates at an early stage. The proposed G-FINDER bears some distinctive features, including (1) *generality*, being able to handle different types of inexact matching (e.g., missing nodes, missing edges, intermediate vertices) on node attributed and/or edge attributed graphs or multigraphs; (2) *effectiveness*, achieving up to 30% F1-Score improvement over the best known competitor; and (3) *efficiency*, scaling near-linearly w.r.t. the size of the data graph as well as the query graph.**

***Index Terms*—subgraph matching, approximate matching, subgraph index**

I. INTRODUCTION

(Approximate) Subgraph matching is a core primitive across a number of disciplines, ranging from data mining, databases, information retrieval, computer vision to natural language processing. In data mining, frequent patterns can be found by approximate subgraph isomorphism [1]; in databases and information retrieval, a query to search for a set of inter-correlated research papers can be treated as a subgraph matching problem [2]; in computer vision, the symbol and object recognition can be formulated as an approximate subgraph matching problem with a tolerance of node merging and splitting, due to object overlapping or segmentation error [3]; in natural language processing, words in a lexical resource can be viewed as nodes in a graph, their relations can be viewed as edges between nodes, and paraphrases can then be seen as matching graphs [4].

Generally speaking, the existing work can be grouped into two categories, including the exact matching methods and the inexact matching methods. Each of these two types of methods has its own pros and cons, which we elaborate below.

On one hand, extensive research effort has been devoted to exact matching, e.g., [5] [6] [7] [8] . The key ideas behind these methods are either pruning false candidates at an early stage and/or building indexes for promising candidates to accelerate the search. Through some effective pruning conditions, the search space can often be greatly reduced. However, many real graphs are noisy and incomplete. Consequently, the observed data graph might not contain any exact matching subgraph at all. In this case (i.e., whether or not exact matching exists is unknown), existing index-based exact matching methods (e.g., [6]) might iterate over *all* possible candidates in the index, which has an exponential time complexity before returning nothing. Moreover, in progressive search [9] [10], the user might not know what exactly s/he is looking for. In this case, exact matching, even if it exists, might not be desirable.

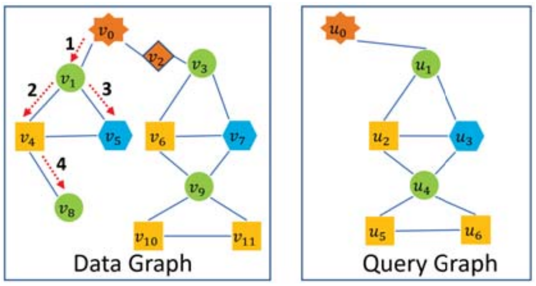


Fig. 1. An illustrative example. Left is the data graph and right is the query graph. Different node shapes and colors represent node attributes. Red dash arrows show the matching steps of TALE [12] which selects one important node at the first step. The proposed G-FINDER finds a matching subgraph induced by (v_0, v_2 v_3 ,v_6, v_7, v_9, v_{10}, v_{11}).

On the other hand, existing inexact subgraph matching methods often rely on certain heuristics to identify important seed nodes, and then progressively expand to neighbors of seeds [11] [12] [13]. Usually, these methods can return an approximate matching subgraph in a relatively short time. However, the greedy strategy of expanding the partial matching subgraph step by step could easily be diverted to a sub-optimal search path. For example, in Figure 1, assuming that the seed node is v_0. Many algorithms (e.g. [12], [13]) will choose v_1 as the next candidate node because v_1 can be perfectly mapped to u_1. After that, it will expand to v_4 and v_5, and finally, return the subgraph induced by (v_0, v_1 ,v_4, v_5, v_8). But obviously, (v_0, v_2 v_3 ,v_6, v_7, v_9, v_{10}, v_{11}) is a better result because it only contains one intermediate vertex v_2.

Due to the noise and incompleteness of the data graph, and the uncertainty of the query graph, exact matching subgraphs may not exist or may not be desirable. Therefore, in this paper, we mainly focus on approximate subgraph matching although our proposed algorithm is also capable to find exact matching. In particular, we propose a new algorithm called G-FINDER. The main idea of G-FINDER is to improve the matching accuracy by building index on data graph according to the query graph structure. Compared with existing inexact matching algorithms, the proposed G-FINDER embraces a novel auxiliary data structure called LOOKUP-TABLE (LTB), in conjunction with a neighborhood expansion method to index the candidate vertices of the query graph, and in the meanwhile it maintains the topology structure of the query graph. Furthermore, the proposed LTB can effectively prune false-positive candidates of query nodes for the purpose of computing an effective matching order.

The main contributions of the paper are

(1) **Data Structure**. We propose a novel auxiliary data structure (LOOKUP-TABLE), in conjunction with a neighborhood expansion method to effectively and efficiently index candidate vertices.
(2) **Algorithm**. We propose an algorithm (G-FINDER) which can handle a variety of different types of inexact matching on node attributed and/or edge attributed graphs. By further integrating LTB with a dynamic filtering and refinement strategy, G-FINDER can effectively prune the false candidates at an early stage to reduce the search space.
(3) **Empirical Evaluations**. We conduct extensive experiments on various real datasets, and demonstrate that G-FINDER (1) is up to 30% better than the best baseline in F1-Score; and (2) scales near-linearly w.r.t. the size of the data graph as well as the query graph, for the data graph with rich attribute information.

II. PROBLEM DEFINITION

Table I summarizes the main symbols and notations used in the paper. For clarity, we refer to nodes in $\mathcal{Q}$ as nodes and that in $\mathcal{G}$ as vertices. Likewise, we call edges in $\mathcal{Q}$ as edges and that in $\mathcal{G}$ as links. We use u and $\hat{u}$ to refer to nodes of the query graph $\mathcal{Q}$; and use v, $\hat{v}$ to refer to vertices of the data graph $\mathcal{G}$. A graph is denoted as a tuple $\{V, E, L\}$, where V is the node/vertex set, E is the edge/link set and L is an attribute function which maps a node/vertex or an edge/link to an attribute value. Given a query graph $\mathcal{Q}$ and a data graph $\mathcal{G}$, we say $\mathcal{Q}$ is the subgraph of $\mathcal{G}$ if there exists an exact subgraph matching from $\mathcal{Q}$ to $\mathcal{G}$ which satisfies the following condition.

Definition 1: Exact Subgraph Matching [14]. An exact subgraph matching (subgraph isomorphism) is an injective function $m()$: $V(\mathcal{Q}) \longrightarrow V(\mathcal{G})$, which satisfies: (1) $\forall\ u \in V(\mathcal{Q})$, $m(u) \in V(\mathcal{G})$ and $L(u) = L(m(u))$; (2) $\forall\ (u_a, u_b) \in E(\mathcal{Q})$, $(m(u_a), m(u_b)) \in E(\mathcal{G})$ and $L(u_a, u_b) = L(m(u_a), m(u_b))$, where L is the attribute function.

Next, we introduce some basic concepts of approximate subgraph matching. Compared with exact subgraph matching, approximate subgraph matching can tolerate some missing query nodes, missing query edges and intermediate data vertices. We say a node u is a missing query node, if $u \in V(\mathcal{Q})$, but $m(u) \notin V(\mathcal{G})$. We say an edge (u_a, u_b) is a missing query edge, if $(u_a, u_b) \in E(\mathcal{Q})$, but $(m(u_a), m(u_b)) \notin E(\mathcal{G})$. We say a data vertex v_i returned by G-FINDER is an intermediate data vertex if it satisfies: (1) two nodes u_a, u_b in $V(\mathcal{Q})$ and $(u_a, u_b) \in E(\mathcal{Q})$, but $(m(u_a), m(u_b)) \notin E(\mathcal{G})$; (2) $(m(u_a), v_i) \in E(\mathcal{G})$ and $(v_i, m(u_b)) \in E(\mathcal{G})$. Last but not least, the remaining non-missing query nodes in $\mathcal{Q}$ and non-intermediate data vertices in the resulting subgraph returned by a matching algorithm which satisfy the mapping function $m()$ are referred to as matching pairs. For example, in Figure 1, assuming the subgraph return by a matching algorithm is $(v_0, v_2, v_3, v_6, v_7, v_9, v_{10}, v_{11})$, then v_2 is an intermediate data vertex. If the matching subgraph is $(v_0, v_1, v_4, v_5, v_8)$, then u_5 and u_6 are missing query nodes, and (u_3, u_4), (u_4, u_5), (u_4, u_6), (u_5, u_6) are missing query edges, and (u_0, v_0) is a matching pair.

Another subtle difference between exact subgraph matching and approximate subgraph matching is the methodology to measure the goodness of a matching. For approximate matching, an intuitive way to measure the quality of a resulting matching subgraph $\mathcal{H}$ (e.g., found by G-FINDER) is to use a loss function to quantify the difference between the matching subgraph and the query graph $\mathcal{Q}$. If the resulting matching subgraph has more intermediate data vertices, and the query graph $\mathcal{Q}$ has more missing query nodes and missing query edges, the loss function cost should be higher. In this paper, we use a linear loss function defined as below.

Definition 2: Linear Loss Function. Consider a query graph $\mathcal{Q}$ and a resulting matching subgraph $\mathcal{H}$. The linear loss function $f(\mathcal{Q}, \mathcal{H})$ is defined as $f(\mathcal{Q}, \mathcal{H}) = w_1 \times MN + w_2 \times ME + w_3 \times IN$, where MN is the number of missing query nodes in $\mathcal{Q}$, ME is the number of missing query edges in $\mathcal{Q}$, and IN is the number of intermediate data vertices in $\mathcal{H}$.

In the above loss function, w_1, w_2, w_3 are the weights for different types of inexact matching. By changing these weights, we can adjust the tolerance of the number of missing query nodes/edges and intermediate data vertices. For example, if w_1 is small, the results could tolerance more missing query nodes, and if $w_2 = \infty$, the results cannot tolerance any missing query edges. If an exact matching subgraph exists, the loss function cost should be 0.

In exact subgraph matching, a node u in $\mathcal{Q}$ can be mapped to a vertex v in $\mathcal{G}$ only if each of u's neighbors could be mapped to some of v's neighbors. However, this hard constraint can be relaxed in approximate subgraph matching. That is, a node u can be mapped to a vertex v even though their neighbors could not be perfectly matched with each other. For example, in Figure 1, u_1 could be mapped to v_3, even though u_0 cannot match to any of v_3's neighbors. In the proposed G-FINDER, we allow a node in $\mathcal{Q}$ and a vertex in $\mathcal{G}$ to be mapped to each other if they satisify the following conditions: (1) have the same attribute value, (2) bear a high node-vertex similarity, and (3) their neighbors also bear high node-vertex similarity, where the node-vertex similarity is defined as follows.

TABLE I
NOTATIONS AND DEFINITION

Symbols	Definition
$\mathcal{Q}$={V_Q, E_Q, L_Q}	an attributed query graph
$\mathcal{G}$={V_G, E_G, L_G}	an attributed data graph
$\mathcal{H}_i$	a partial matching with i nodes
$\mathcal{Q}_i$	a partial query graph with i nodes
u, $\hat{u}$	nodes in query graph
v, $\hat{v}$	vertices in data graph
m(u_i) = v_i	matching function
N(u)	neighborhood of query node u
C(u)	candidate set of query node u
LTB(u)	LOOKUP-TABLE of query node u
LTBG	LOOKUP-TABLE-GRAPH formed by all LTBs
f($\mathcal{Q}$, $\mathcal{H}$)	the loss function

Definition 3: *Node-Vertex Similarity* is defined as $sim(v_i, u_i) = \frac{|(m(N(u_i)) \cap N(v_i))|}{|N(u_i)|}$, where $N(v_i)$ denotes the neighbors of v_i, and $m(N(u_i)) \cap N(v_i)$ represents the nodes in $N(u_i)$ that are mapped to $N(v_i)$.

For example, in Figure 1, since u_2 can be mapped to v_6, and u_3 can be mapped to v_7, the node-vertex similarity between v_3 and u_1 is $sim(v_3, u_1) = 2/3$.

Remarks. For both loss function (Definition 2) and node-vertex similarity (Definition 3), there exist alternative choices. For example, there are a wealth of different graph similarity measures, e.g., graph edit distance [15], graph kernel [16] and embedding based graph similarity [17]. Likewise, there are also rich node similarity measures in the literature, e.g., matrix based method [10], graph neural networks based similarity [18]. In principle, the proposed G-FINDER algorithm can admit these alternative graph similarity as well as node similarity measures. In this paper, we use these two relatively simple forms due to the efficiency consideration.

Finally, the *top-k* approximate subgraph matching problem can be formally defined as follows.

Problem Definition 1: top-k Approximate Subgraph Matching: [1]

Given: (1) An attributed data graph $\mathcal{G}$, (2) an attributed query graph $\mathcal{Q}$, (3) the number of desired matching subgraphs k, and (4) a loss function $f()$;

Find: k approximate matching subgraphs that match the query graph $\mathcal{Q}$ as well as possible (have the smallest loss function cost).

III. PROPOSED G-FINDER: OVERVIEW

The overall framework of G-FINDER contains four major steps which are illustrated in Figure 2. Algorithm 1 presents the corresponding pseudo code, and Table II compares G-FINDER and existing algorithms, in terms of the input data graph, the ability for exact matching, the types of inexact matching, and computational efficiency.

First (Root Selection, Lines 3-4), G-FINDER selects a root node from the query graph. Second (LOOKUP-TABLE-GRAPH Construction, Line 7), a dynamic filtering and refinement strategy is used to construct LOOKUP-TABLE-GRAPH (short for LTBG) for the query graph. Third (Matching Identification, Line 8), a matching order is calculated based on the constructed LOOKUP-TABLE-GRAPH. Finally (Subgraph Enumeration, Line 9), G-FINDER searches LOOKUP-TABLE-GRAPH according to the matching order to return the *top-k* results which have the smallest loss function cost. In the remaining of this section, we highlight each of these four steps.

[1]The data graph and query graph can be multi-graph. For easier understanding, we mainly focus on graph with node attribute in this paper.

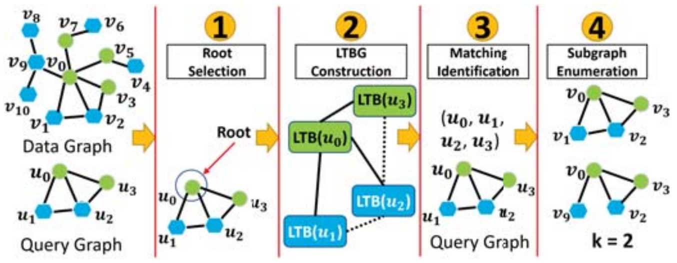


Fig. 2. The overall framework of G-FINDER.

```
Algorithm 1 G-FINDER
 1: Input: a query $\mathcal{Q}$, a data graph $\mathcal{G}$, the number of desired
    matching subgraphs $k$ and a loss function $f()$.
 2: Output: top-$k$ approximate matching subgraphs
 3: ($V_c$, $V_t$) $\leftarrow$ Core-Forest-Decompose($\mathcal{Q}$) // $V_c$ is the core
    structure, and $V_t$ is the forest structure.
 4: (Query root $u_r$, Data root vertices $S_{root}$) $\leftarrow$ Root-
    Selection
 5: Initialize top-$k$ heap
 6: for each $root_i \in S_{root}$ do
 7:     LTBG $\leftarrow$ LTBG-Builder($\mathcal{Q}$, $u_r$, $root_i$, $\mathcal{G}$)
 8:     M $\leftarrow$ calculating matching order by LTBG
 9:     Subgraph-Enumeration($k$, LTBG, $\mathcal{Q}$, $\mathcal{G}$, M, $f()$)
10: end for
11: Return top-$k$ approximate matching subgraphs
```

A. Step 1: Root Selection

Given a query graph $\mathcal{Q}$, we first select a root node to start the matching process. For example, in Figure 1, we can choose u_0 as the root. Many existing root selection methods can be used in the proposed G-FINDER algorithm. Basically, we would like to choose the root node which (1) has as few candidates as possible, and (2) is at the center of the query graph so that the diameter of the search space could be minimized [5]. With these two design objectives in mind, we first decompose the query graph into the core-forest structure [2]. After that, we choose the node $u_r \leftarrow \arg\min_u \frac{|C(u)|}{\deg(u)}$ from the core structure as the root, where $|C(u)|$ is the size of the candidate set of u in the data graph, and deg(u) is the degree of node u in the query graph. [3]

[2]The core structure of a graph $\mathcal{Q}$ is a maximum subgraph of $\mathcal{Q}$ where each node has at least 2 neighbors [6]. The remaining parts of the query graph is called the forest-structure of $\mathcal{Q}$. Figure 3(a-b) gives an example of the core-forst decomposition.

[3]For Figure 1 in practice, G-FINDER will choose u_4 as the root node. However in order to explain the details of G-FINDER, we assume u_0 is the root node for all examples in this paper. Moreover, the root node could not be a missing vertex.

TABLE II
COMPARISON OF SUBGRAPH MATCHING METHODS. ROWS ARE METHODS AND COLUMNS ARE DESIRED PROPERTIES.

	node attribute	edge attribute	exact matching	missing nodes	missing edges	intermediate nodes	index-based	early pruning	multi-graph
G-FINDER	✓	✓	✓	✓	✓	✓	✓	✓	✓
CFL [6]	✓	✗	✓	✗	✗	✗	✓	✓	✗
G-Ray [13]	✓	✗	✗	✓	✓	✓	✗	✗	✗
FIRST [10]	✓	✓	✗	✓	✓	✓	✗	✗	✗
MAGE [19]	✓	✓	✗	✓	✓	✓	✗	✗	✗
FilM [7]	✗	✗	✓	✗	✗	✗	✗	✓	✓
NeMa [20]	✓	✓	✓	✓	✓	✓	✓	✓	✗

B. *Step 2:* LOOKUP-TABLE-GRAPH *Construction*

Following the root selection, the next step is to traverse the query graph to build index node by node. The key of this step is a novel data structure called LOOKUP-TABLE (short for LTB), which stores the information of the candidate vertices. Each node u in the query graph has a corresponding LTB(u), and all the LTBs will form a graph which shares the same topology as the query graph. We call this new graph as LOOKUP-TABLE-GRAPH (short for LTBG). For example, in column 3 of Figure 2, LTB(u_0), LTB(u_1), LTB(u_2), and LTB(u_3) form a LOOKUP-TABLE-GRAPH for the input query.

The LTBG is built according to the traversing order of the query graph. Figure 3(c-d) gives an example of two traversal strategies, including BFS Tree and Dynamic-Tree. BFS is a common traversal strategy used by several existing exact subgraph matching methods [5] [6]. There are two kinds of edges in this strategy, including tree edges (TE) and non-tree edges (NTE). The edges that exist in both the BFS tree and the query graph are called tree edges. The edges that only exist in the query graph but not in the BFS tree are called non-tree edges. For example, in Figure 3(c), (u_2, u_4) is a tree edge, and (u_3, u_4) is a non-tree edge.

Although BFS is a common method for exact subgraph matching to traverse the query graph, it may not be suitable for approximate matching. For example, suppose we use BFS traversal strategy for the data graph in Figure 1 and the query graph in Figure 3(a). Since u_7 has no candidates in the data graph, LTB(u_7) is empty, and consequently, both u_5 and u_6 will be ignored. Finally, the result graphs found by G-FINDER will be bad [4].

To tackle this issue for approximate subgraph matching, we propose a heap based dynamic traversal strategy to build LOOKUP-TABLE-GRAPH. We maintain a heap to store the current processing LTB and each time we pop the LTB with the smallest $\frac{|C(u)|}{\deg(u)}$ from the heap. We recursively build LOOKUP-TABLE for its un-visited neighbors and push them into the heap. We repeat this procedure until all nodes in the query graph are processed. More details will be further presented in Section 4.1.

[4]See the performance of G-FINDER in `Human` dataset for details (section V-B).

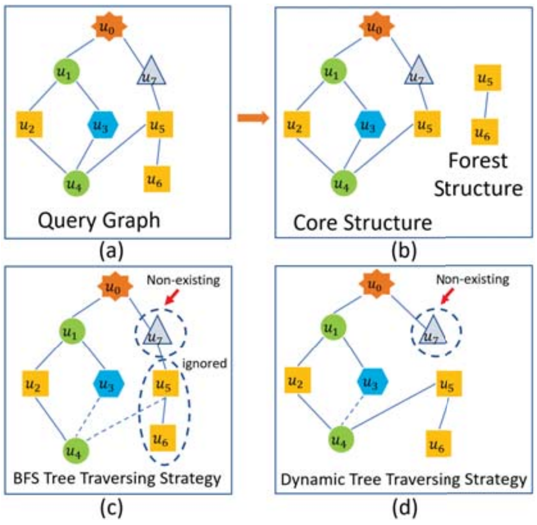


Fig. 3. The core-forest decomposition (a-b) and two traversal strategies (c-d). The dash lines represent non-tree edges.

C. *Step 3: Matching Identification*

A matching order is the node sequence of the query graph, based on which we match the candidates in LTBG to the query nodes. For example, in Figure 2, the matching order is (u_0, u_1, u_2, u_3). G-FINDER will first choose a good candidate v_0 in LTB(u_0), and then choose a good candidate v_1 in LTB(u_1) which connects to v_0. Next, G-FINDER will choose a good candidate v_2 in LTB(u_2) which connects to both v_0 and v_1. Finally, G-FINDER will choose a good candidate v_3 in LTB(u_3) which connects to both v_0 and v_2.

There are several methods for choosing matching order, such as the least frequent path first [21], greedy approach to order paths [6], and the least frequent node first [22]. The general idea behind these methods is to select the node or path with the smallest number of candidate vertices first, which will make the search space compact. In this paper, we found that these methods lead to similar performance in terms of matching accuracy for G-FINDER, and we use the greedy approach to order the paths for its computational efficiency.

D. *Step 4: Subgraph-Enumeration*

In this step, we search candidates in the LTBG according to the matching order from Step 3, and find the *top-k* approximate matching subgraphs which have the smallest loss function cost. During the matching process, we maintain another heap to store the k best matching results so far, and update the heap when we find a better result. The details of subgraph enumeration will be introduced in Section 4.2.

IV. PROPOSED G-FINDER: DETAILS

In this section, we presents the details of LOOKUP-TABLE-GRAPH construction and subgraph enumeration, which are the two most complicate steps in the proposed G-FINDER(Figure 2). At the end of this section, we analyze the time and space complexity of G-FINDER.

A. LOOKUP-TABLE-GRAPH *Construction*

A - Lookup Table Structure. We design a novel data structure called LOOKUP-TABLE to store the information of the candidate vertices, including a candidate set, parent-child relationship, the number of intermediate vertices and the node-vertex similarity. Figure 4 gives an example, which illustrates the LTBs of $(u_0, u_1, u_2, u_3, u_4, u_7)$ for the query graph in Figure 3 w.r.t. the data graph in Figure 1, where each LTB is visualized as a table. The black solid arrows represent the parent-child relationship between different LTBs. For example, LTB(u_1) is the parent of both LTB(u_2) and LTB(u_3). The black dash arrows represent the prior-visited neighborhood relationship. For example, there is a non-tree edge between u_3 and u_4, and LTB(u_3) is built before LTB(u_4). Therefore, there is a prior-visited neighborhood relationship between LTB(u_3) and LTB(u_4). The red solid arrows represent the parent-child relationship of candidate vertices in the data graph. For example, v_0 is the parent of both v_1 and v_3. In the LOOKUP-TABLE, the node-vertex similarity (`SIM`) and intermediate vertex number (`IVN`) are recorded. For example, there is an intermediate vertex between v_0 and v_3 in the data graph (Figure 1). Therefore, the entry at the third row and the third column of LTB(u_1) is 1. When building the LTB, the candidate vertices will be sorted by their intermediate vertex number (`IVN`) and node-vertex similarity (`SIM`). During the matching process, we first choose the vertex with the high node-vertex similarity (`SIM`) and small intermediate vertex number (`IVN`) so that the corresponding linear loss function cost would be small.

B - LTB Construction. Following the root node selection (Step 1 of Figure 2), we dynamically explore the data graph to select the candidates for each query node. When building LOOKUP-TABLE, we design several refinement and filtering strategies to prune unpromising candidates so as to minimize the size of candidates. These refinement strategies include node-vertex similarity, intermediate vertex number and missing node/edge tolerance. To make it easier to understand, we first introduce how to construct a single LTB, followed by how to build the entire LTBG dynamically.

The key of building a single LOOKUP-TABLE is to effectively find candidate vertices. We use Figure 1 as an example to explain this process. Suppose we have a partial matching $\mathcal{H}_i$, e.g., (v_0, v_1, v_4) in Figure 1, where i means that there are i vertices in the partial matching (i.e., $i = 3$ in this case). The corresponding query graph is $\mathcal{Q}_i$, e.g., (u_0, u_1, u_2) in Figure 1, and we want to find the matching vertex in the data graph for the next node u_j (e.g., u_3). Intuitively, if v_j is a good candidate of u_j, for each node $u_n \in \mathcal{Q}_i \cap N(u_j)$, v_j must be adjacent to u_n's candidate v_n, where $v_n \in \mathcal{H}_i$. For example, v_5 is a good candidate of u_3 because it is adjacent to both v_1 and v_4. However, for approximate matching, v_j does not need to satisfy such a hard constraint if we are willing to tolerate some missing query nodes/edges. For example, in Step 4 of Figure 2, (v_9, v_0, v_2, v_3) is also a good approximate matching of query graph (u_0, u_1, u_2, u_3), even though there is no direct link between v_9 and v_2 (missing edge).

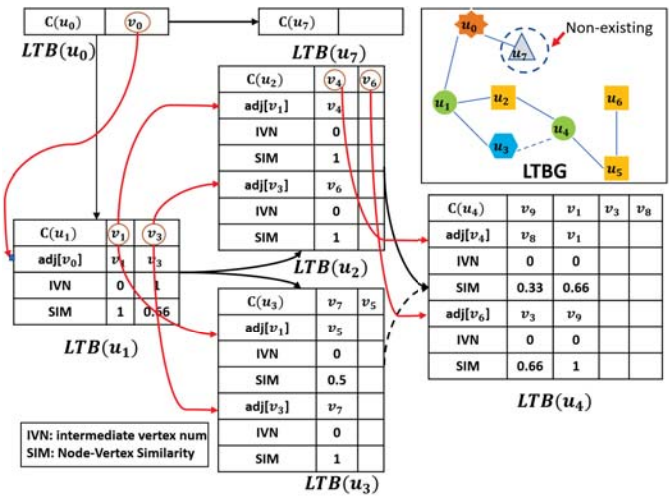


Fig. 4. An example of LOOKUP-TABLE-GRAPH for the data graph in Figure 1 w.r.t. the query graph in Figure 3. We omit the LTBs of u_5 and u_6 for brevity. An empty LTB means it is absent.

Algorithm 2 LTB-Construction

```
1:  Input: a data graph $\mathcal{G}$, a query graph $\mathcal{Q}$, a query node $u_i$
    and its parent $u_p$.
2:  Output: the LOOKUP-TABLE of $u_i$ over $\mathcal{G}$
3:  Mark $u_i$ as visited;
4:  for each visited neighboring query vertex $\hat{u}$ of $u_i$ do
5:      for each vertex $\hat{v} \in$ C( $\hat{u}$ ) do
6:          $S \leftarrow$ Neighbor-Expander($\hat{v}$, $L(u_i)$, $\mathcal{G}$)
7:      end for
8:  end for
9:  Set LTB$(u_i)$.parent = $u_p$
10: for each vertex $v_n \in C(u_p)$ do
11:     for each vertex $n \in$ (Neighbor-Expander($v_n$, $L(u_i)$, $\mathcal{G}$)
        $\cap$ $S$) and $n$.connection $\geq$ ($|N(u_i)|$-$d$) do
12:         // $d$ is the the number of missing edges the algorithm
            is willing to tolerate.
13:         Add $n$ to LTB$(u_i)$.adj[$v_n$] and record IVN and SIM.
14:     end for
15: end for
16: Sort $C(u_i)$ by IVN and SIM.
17: Return LOOKUP-TABLE of $u_i$
```

Based on the ideas above, we introduce a Neighbor-Expander strategy as follows. Assuming u_p is the parent of u_j and its corresponding data vertex is v_p. Neighbor-Expander first selects all of v_p's t-hop neighbors[5] with attribute $L(u_j)$. Then, it adds these vertices into set M. For each vertex $v_m \in M$, we calculate the node-vertex similarity between v_m and u_j. If the similarity is at least s, which is a pre-defined threshold, v_m is treated as a good candidate. For example, in Figure 1, v_3 is a 2-hop neighbor of v_0 and the node-vertex similarity $sim(v_3, u_1)$ is $2/3$. We treat it as a good candidate if the threshold $s = 0.5$. Finally, for each v_m, we calculate how many vertices in $\mathcal{H}_i$ it connects to. If the number is bigger

[5] $t = 2$ in this paper for approximate subgraph matching, and users can choose a different t.

than $|N(u_j)| - d$, where d is the number of missing edges the algorithm is willing to tolerate, we consider it is good.

Likewise, we build a LOOKUP-TABLE by exploiting its prior-visited neighbors' LTB information. For example, assume u_i is a node in $\mathcal{Q}$, and u_p is u_i's parent. When building LTB for u_i, let $N_{\text{visited}}(u_i)$ denote the set of all visited neighbors of $u_i \in \mathcal{Q}$. Notice that all visited nodes have already built the corresponding LTBs. The candidate vertices of u_i are therefore generated from the sets of candidates in $N_{\text{visited}}(u_i)$.

The full algorithm for building a single LOOKUP-TABLE is summarized in Algorithm 2. First, for each vertex u_j in $N_{\text{visited}}(u_i)$, we iterate u_j's candidates. For each vertex $v_j \in C(u_j)$, we use Neighbor-Expander to select the good candidates v with high node-vertex similarity with respect to u_i (i.e., $sim(v, u_i) \geq s$) and insert them into vertex set S (Lines 4-6). For the example in Figure 4, LTB(u_4) is built according to LTB(u_2) and LTB(u_3). The candidate sets of LTB(u_2) and LTB(u_3) are (v_4, v_6) and (v_5, v_7), respectively. Their 2-hop neighbors with attribute $L(u_4)$ are $(v_1, v_8), (v_3, v_9), (v_1)$ and (v_3, v_9), respectively. Assume the threshold $s = 1/4$. Then, all of them are good candidates for node u_4, i.e., $S = (v_1, v_8, v_3, v_9)$. Second, we iterate each candidate vertex in $C(u_p)$, and record its parent-child relationship. For each node, we select all its t-hop neighbors in S and store these vertices in LTB(u_i).adj[v_n], which means that v_n is these vertices' common parent in the data graph. Finally, we record the number of intermediate vertices (Line 12). For the example in Figure 4, the parent node of u_4 is u_2 and $C(u_2) = (v_4, v_6)$. Therefore, we have that LTB(u_4).adj[v_4]=(v_8, v_1) and LTB(u_4).adj[v_6]=(v_3, v_9).

C - LTBG Construction. Besides LTB construction, another key issue of building LTBG is to select the LTBG building order for the query graph $\mathcal{Q}$. The pseudo code for building LOOKUP-TABLE-GRAPH is shown in Algorithm 3. We dynamically build LTBG with a min heap to store the LTBs of all current processing nodes. First, we build LTB for the root node (Line 4), and push its LTB into the min heap (Line 5). Each time we pop a LTB from the min heap, we build LTB for its un-visited children (Lines 7, 9). Then, we push its children LTB into the min heap (Line 11). We repeat this procedure until all nodes in the query graph are processed. During this process, a subtle issue is how to pop the next LTB from the min heap. We select the LTB with the minimum $\frac{|C(u)|}{\deg(u)}$ for the following reason. If a LTB has smaller $\frac{|C(u)|}{\deg(u)}$, its children will be more likely to have less candidate vertices.

Algorithm 3 LTBG-Builder

```
1: Input: a query $\mathcal{Q}$, its root node $u_r$ and the candidate vertex $v_r$, and a data graph $\mathcal{G}$.
2: Output: the LOOKUP-TABLE-GRAPH of $\mathcal{Q}$ over $\mathcal{G}$
3: Mark $u_r$ as visited.
4: Initialize LTB($u_r$) by adding $v_r$ to its candidate set.
5: Push LTB($u_r$) into the min heap
6: while min heap is not empty do
7:     currentLTB $\leftarrow$ Pop a LTB from the min heap
8:     curQueryId $\leftarrow$ currentLTB.Id
9:     for each un-visited neighboring query node $u$ of curQueryId do
10:        nextLTB $\leftarrow$ LTB-Construction($\mathcal{G}$, $\mathcal{Q}$, $u$, curQueryId)
11:        LTB-heap.push(nextLTB)
12:    end for
13: end while
14: Return LOOKUP-TABLE-GRAPH.
```

Algorithm 4 Subgraph-Enumeration

```
1: Input: a parameter $k$, the LTBG, a query graph $\mathcal{Q}$, a data graph $\mathcal{G}$, the matching order $M$, and the linear loss function $f()$.
2: Output: the top-k approximate subgraphs
3: while $\mathcal{H}_i \leftarrow$ search subgraph by $M$ do
4:     cost = f($\mathcal{Q}_i$, $\mathcal{H}_i$)
5:     if cost > max loss function cost in the max heap then
6:         backtrace, stop further expanding $\mathcal{H}_i$.
7:     end if
8:     if $\mathcal{H}_i$ can not be further expanded then
9:         if cost < max loss function cost in the max heap then
10:            Add $\mathcal{H}_i$ to top-k heap
11:        end if
12:    end if
13: end while
14: Return the top-k approximate subgraphs
```

B. Subgraph-Enumeration

In this subsection, we describe the *top-k* search algorithm (i.e., Step 4 of Figure 2). First, a new *top-k* max heap is maintained to store the best k answers that have been seen so far. During the search process, we calculate the current linear loss function cost between the partial query graph $\mathcal{Q}_i$ (i means that there are i nodes in the partial query graph $\mathcal{Q}_i$) and the partial matching graph $\mathcal{H}_i$. Then, we compare the linear loss function cost with the maximum loss function cost in the heap. If the current loss function cost is higher than the maximum loss function cost in the max heap, we stop further expanding the current partial matching to allow earlier termination.

Let us take Figure 5 as an example. Suppose the matching order is $< u_0, u_1, u_7, u_2, u_3, u_4, u_5, u_6 >$ and we have gotten a partial matching $\mathcal{H}_5 = (v_0, v_2, v_3, v_6, v_7)$. The corresponding partial query graph is $\mathcal{Q}_5 = (u_0, u_1, u_2, u_3, u_7)$. If $w_1 = w_2 = w_3 = 1$, the loss function cost is 2 with one intermediate data vertex v_2 and one missing query node u_7. If this is higher than the maximum loss function cost in the max heap, we stop further expanding the current partial matching. Otherwise we continue to expand the current partial matching subgraph. The full algorithm of Subgraph-Enumeration is shown in Algorithm 4.

C. Complexity Analysis

The complexity of G-FINDER mainly come from two parts: LTBG-Builder (Step 2) and Subgraph-Enumeration (Step 4). The worst-case space complexity of LTBG-Builder for a query $\mathcal{Q}$ over a data graph $\mathcal{G}$ is $O(|V(\mathcal{Q})| \times |E(\mathcal{G})|)$

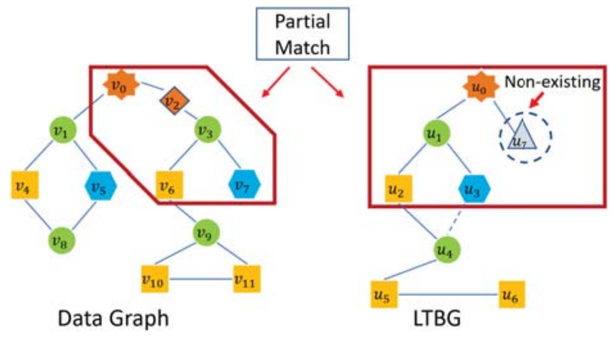


Fig. 5. An example of partial matching.

and its time complexity is $O(|E(\mathcal{Q})| \times |E(\mathcal{G})| \times |V(\mathcal{G})|)$. For Subgraph-Enumeration, its time complexity in the worst case is $O(\prod_{i=0}^{|V(\mathcal{Q})|-1} |V(\mathcal{G}) - i|)$. However, the worst case only happens when the data graph is a complete graph, which is extremely unlikely for any real graph. The best case time complexity is $O(|V(\mathcal{Q})|)$, which happens when all nodes have distinct attributes. On average, for a data graph $\mathcal{G}$ with D attributes, its worst time complexity is $O(\prod_{i=0}^{D} \prod_{j=0}^{\frac{|V(\mathcal{Q})|}{D}-1} |\frac{|V(\mathcal{G})|}{D} - j|)$. Empirically, we find the total running time of G-FINDER scales near-linearly with respect to the number of nodes of the data graph and that of the query graph, for the data graph with rich attribute information.

V. EXPERIMENTS

In this section, we conduct empirical studies to evaluate the effectiveness and efficiency of the proposed algorithm. The experiments are designed to answer the following questions:

- **Q1. Effectiveness:** How accurate is the proposed G-FINDER algorithm for subgraph matching?
- **Q2. Efficiency:** How fast and scalable is the proposed G-FINDER algorithm?

A. Experimental Setup

Datasets. We use 8 real-world datasets in our experiments, which are summarized in Table III. `Human` and `HPRD` are two protein-protein interaction networks used by [6]. `DBLP` is a dataset which represents the relationship between authors and papers. `Flickr` shows the friendship of users on Flickr. `LastFm` contains the following relationships of users on LastFm. `AMiner` represents the academic social network, where undirected edges represent co-authorship and the node attribute vector is extracted from the number of published papers. `PNNL-V4` is created by Pacific Northwest National Laboratory (PNNL). `IMDB` is the same movie dataset used in NeMa [20].

Baseline Methods. Five algorithms are used as the baselines, including G-Ray [13], MAGE [19], FIRST [10], FilM [7] and NeMa [20]. Source codes of the baseline methods are provided by the original authors. For the proposed G-FINDER algorithm, we have two variants based on different traversing strategies outlined in Section 3.2, including G-FINDER-BFS and G-FINDER-Dynamic.

TABLE III
SUMMARY OF DATASETS

Name	# of Nodes	# of Edges	Attribute	# of Attribute
`Human`	4,674	86,282	Node only	44
`HPRD`	9,460	37,081	Node only	307
`DBLP`	9,143	16,338	Node only	29
`Flickr`	12,974	16,149	Node only	3
`LastFm`	136,421	1,685,524	Node only	3
`AMiner`	1,274,360	4,756,194	Node only	300
`PNNL-V4`	22,154	460,196	Edge only	259,917
`IMDB`	2,932,657	11,040,263	Node & Edge	# of Nodes + # of Edges

Evaluation Metrics. We use the F1-Score to evaluate the matching accuracy and total running time [6] to evaluate the efficiency of different algorithms. Other alternative accuracy metrics include (1) the linear loss function defined in Section 2; (2) % Extra nodes in MAGE and FIRST; (3) % Exact Matching Nodes in FIRST. The proposed G-FINDER outperforms the baseline methods on these alternative metrics as well. Due to the space limitation, we only show the results of F1-Score.

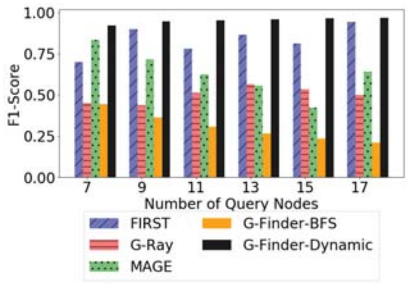
Reproducibility. G-FINDER is implemented in C++. Experiments are conducted on a machine with an Intel Core-i7 3.20GHz CPU and 32GB memory. We will release the source code after the paper is published.

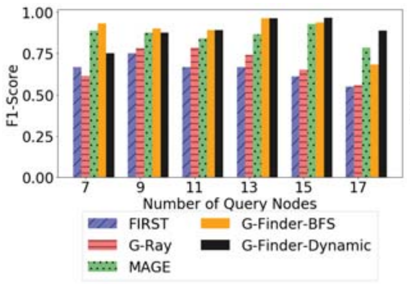
B. Effectiveness Results

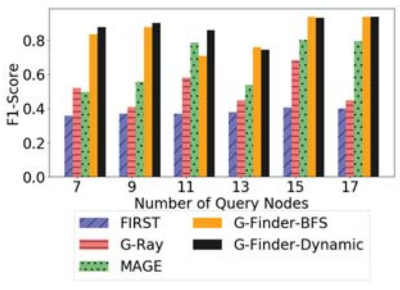
Here, we compare G-FINDER-Dynamic and G-FINDER-BFS against three baseline algorithms, including G-Ray, MAGE and FIRST. For the query graph, it is generated as follows. We select a connected subgraph of the data graph, and then inject a variety of different types of noise, e.g., node addition, node attribute alteration, edge deletion. More specifically, we find a induced subgraph in the data graph, and treat it as the query graph. This means that there is at least one exact matching for the query graph. Then, we will add some noises into the query graph. For example, we randomly delete some edges in the query graph, change some nodes' attributes or add some extra nodes into the query graph. These extra nodes don't exist in the data graph. The ground truth is the induced subgraph. The main parameters are set as $k = 10, w_1 = w_2 = w_3 = 1$ [7]. Figure 6 shows the F1-Score of these 5 algorithms on 6 datasets, including `Human`, `HPRD`, `DBLP`, `Flickr`, `Lastfm` and `AMiner`. The X-axis is the number of query nodes, and the Y-axis shows the F1-Scores. We can see that the proposed G-FINDER-Dynamic consistently outperforms all the three baseline methods in all cases. For example, on `DBLP`, G-FINDER-Dynamic is 30% better than the best baseline method (MAGE) with 7, 9 and 13 query nodes. Between the two variants of the proposed G-FINDER, G-FINDER-Dynamic wins or achieves the similar high score in almost all cases, with the only three exception on (1) `HPRD` with 7 query nodes, (2) `Lastfm` with

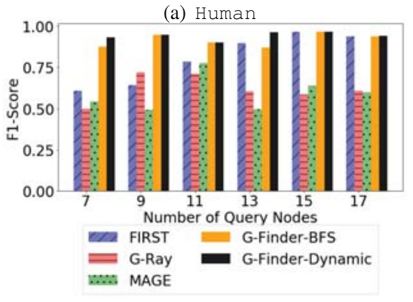
[6]The index construction time is smaller, compared to the query runtime.

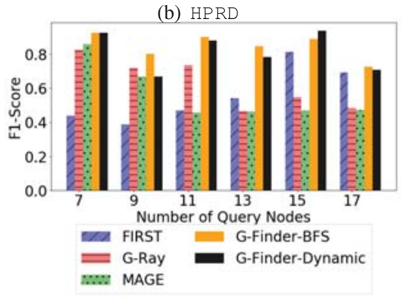
[7]This means that G-FINDER has no bias. For example, if w_1 is larger, G-FINDER perfers to find more result nodes, but the result graph may contain a lot of intermediate nodes.


(a) Human


(b) HPRD


(c) DBLP


(d) Flickr


(e) Lastfm

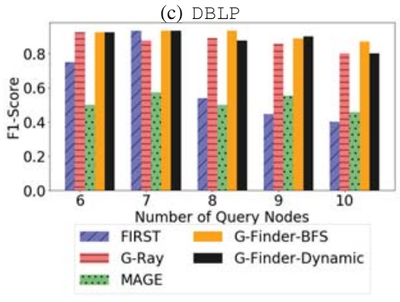
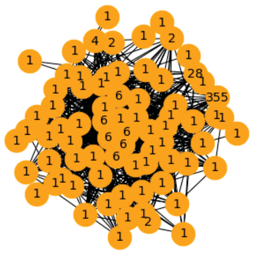

(f) Aminer

Fig. 6. Matching accuracy (F1-Score) comparison. Higher is better. The proposed G-FINDER-Dynamic is the rightmost bar.


(a) FilM

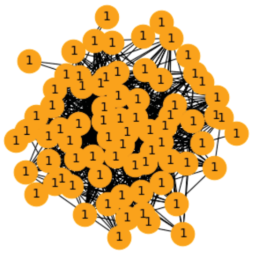

(b) G-FINDER

Fig. 7. Comparison between G-FINDER-Dynamic and FilM. Numbers are the candidate numbers for the corresponding query nodes.

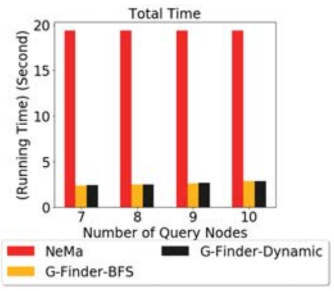


Fig. 8. Running time comparison: G-FINDER-Dynamic vs. NeMa.

9 nodes and (3) AMiner with 10 query nodes. The relatively lower F1-Score of G-FINDER-BFS in Human is due to the following reason. **The noise nodes are close to the root node**. After transforming the query graph into a BFS tree, these nodes of the BFS tree do not have any candidate in the data graph. Consequently, their children nodes are ignored. This is consistent with our analysis in Section 3.2. That is, despite its wide usage in exact subgraph matching, BFS-tree traversing strategy might be sub-optimal for approximate subgraph matching. Rather, the proposed dynamic-tree is a much more effective traversing strategy.

Although G-FINDER is primarily designed for approximate subgraph matching, by (1) setting $t = 1$ (i.e., only considering the direct neighbors in LOOKUP-TABLE-GRAPH) and (2) only retaining the resulting matching subgraphs with a 0 linear loss function value, it can also be used for exact subgraph matching. Here, we compare G-FINDER-Dynamic with FilM [7]. It is worth pointing out that for each query node, FilM returns a candidate set which contains all the possible candidates, including potential false positives. For example, Figure 7(a) is the result generated by FilM on PNNL-V4 dataset. The number inside each query node is the number of candidate vertices for it (e.g., a query node with 6 means that it has 6 candidates). The total number of resulting matching subgraphs by FilM are more than 100 (i.e., the multiplication of the candidate numbers for all query nodes). In this case, there is actually only one exact matching subgraph in the underlying data graph. Therefore, the F1-Score of FilM is close to 0. On the other hand, as we can see from Figure 7(b), the proposed G-FINDER-Dynamic generates exactly one candidate for each query node, precisely finding the only exact matching subgraph[8].

We also compare G-FINDER-Dynamic with another algorithm NeMa [20]. NeMa is for approximate matching, which aims at finding *top-k* matching subgraphs. Different from other baseline methods, NeMa treats each node attribute value as a set of words and uses Jaccard similarity to measure the similarity between attribute values. When comparing G-FINDER-Dynamic with NeMa, to make a fair comparison, we use the same dataset (IMDB) and the same query graphs with node addition noise as NeMa. For each attribute value

[8]We also found that G-FINDER-Dynamic is faster than FilM (134s vs. 199 s, total running time). There is only one query graph in the PNNL dataset.

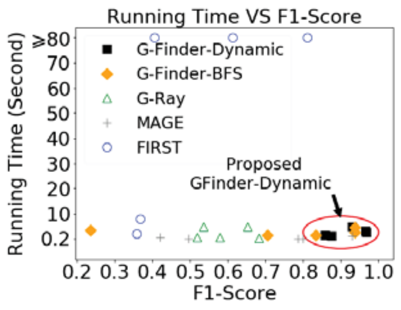
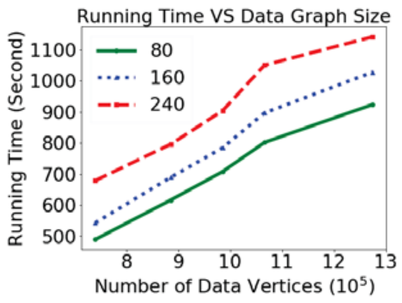
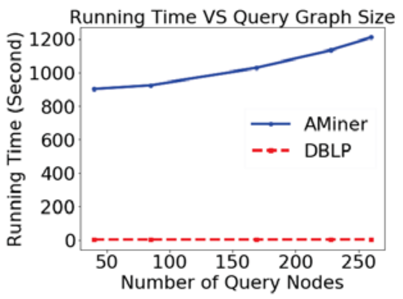


Fig. 9. Matching accuracy and running time trade-off.



(a) G-FINDER-Dynamic Running time vs. data graph size. 80, 160 and 240 mean the size of the query graph.



(b) Running time vs. query graph size.

Fig. 10. Scalability of G-FINDER-Dynamic.

in IMDB, we map it to an integer value when running G-FINDER-Dynamic.

Since both the data graph and query graph have distinct attribute values, these two algorithms find the same matching subgraphs, so they have the same F1-Score (the same effectiveness). Figure 8 compares the total running time of NeMa and G-FINDER-Dynamic. We can see that G-FINDER-Dynamic is $6 \sim 8$ times faster than NeMa.

C. *Efficiency Results*

Figure 9 shows the trade-off between the total running time and matching accuracy (measured by F1-Score) of different methods. Each point in the figure represents a pair of F1-Score and running time, and we present 5 pairs (i.e., points) for each method. As we can see, G-Ray, MAGE and G-FINDER are fastest, all of which can find matching results in a quite short time. However, the matching accuracy (X-axis) of the proposed G-FINDER-Dynamic is much higher than other methods. In most cases, F1-Scores of G-FINDER-Dynamic are greater than 0.8.

Figure 10(a) and Figure 10(b) show the scalability of G-FINDER-Dynamic. We can see that the running time of G-FINDER-Dynamic scales near-linearly w.r.t. the number of nodes of the data graph as well as that of the query graph.

VI. RELATED WORK

In this section, we review the related work, which can be categorized into two groups.

A - Exact subgraph matching. Depending on the specific problem to solve, the exact subgraph matching algorithms can be further divided into two sub-groups. The algorithms in the first sub-group aim to find subgraphs in a database with many data graphs. Representative algorithms include gIndex [14], FG-Index [23], Tree+delta [24] and so on. They typically use graph mining techniques to find small frequent subgraphs (e.g. path, tree) from the database, and then use a filter-and-refine strategy to prune false data graphs. In this way, the search space can often be greatly reduced. The algorithms in the second sub-group mainly focus on finding an exact subgraph from a very large data graph. Representative algorithms includes Ullmann [25], CFL [6] and so on. The main idea behind these algorithms is iteratively trying to map nodes one by one from a query graph to a data graph, and backtracing if the attempt fails.

B - Inexact subgraph matching. Inexact subgraph matching focuses on finding approximate subgraphs on a large data graph. There have been extensive studies on this problem. To name a few, Tong et al. [13] propose the best-effort pattern matching, which aims to maintain the topology of the query. Tian et al. [12] propose an approximate subgraph matching tool TALE with efficient indexing. Khan et al. [20] propose a heuristic approach NeMa based on a new definition of matching cost metric. Pientar et al. [19] propose an algorithm called MAGE which is an improved version of G-Ray. Different from G-Ray, it supports graphs with both node and edge attributes. Zhang et al. propose an inexact subgraph matching algorithm SAPPER [26] which utilizes the hybrid neighborhood unit structures in the index. Tian et al. [11] propose an approximate graph matching algorithm SAGA which employs a flexible graph distance model to measure similarities between graphs. He et al. [27] propose an index based algorithm called Closure-Tree to support both subgraph queries and similarity queries.

VII. CONCLUSION

In this paper, we study the approximate attributed subgraph matching problem and develop an effective and scalable algorithm (G-FINDER). First, we propose a data structure called LOOKUP-TABLE (LTB) to efficiently index the candidates of the query graph. Second, we propose an effective algorithm to build and search LOOKUP-TABLE-GRAPH (LTBG) in order to find the *top-k* approximate subgraphs. Finally, we conduct extensive experiments on real world data, which demonstrate that the proposed G-FINDER (1) achieves an up to 30% F1-Score improvement over the best baseline method, and (1) scales near-linearly to large data graphs with $1M+$ vertices and large query graphs with hundreds of nodes. Future work includes generalizing the proposed G-FINDER to (a) interactive subgraph matching (b) knowledge graph matching.

VIII. ACKNOWLEDGEMENT

This material is supported by the National Science Foundation under Grant No. IIS-1651203, IIS-1715385, IIS-1743040, and CNS-1629888, by DTRA under the grant number HDTRA1-16-0017, by the United States Air Force and DARPA under contract number FA8750-17-C-0153[9], by Army

[9]Distribution Statement ”A” (Approved for Public Release, Distribution Unlimited)

Research Office under the contract number W911NF-16-1-0168, and by the U.S. Department of Homeland Security under Grant Award Number 2017-ST-061-QA0001. The content of the information in this document does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

REFERENCES

[1] P. Anchuri, M. J. Zaki, O. Barkol, S. Golan, and M. Shamy, "Approximate graph mining with label costs," in *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '13. New York, NY, USA: ACM, 2013, pp. 518–526. [Online]. Available: http://doi.acm.org/10.1145/2487575.2487602

[2] L. Hong, L. Zou, X. Lian, and P. S. Yu, "Subgraph matching with set similarity in a large graph database," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 9, pp. 2507–2521, Sep. 2015.

[3] J. Lladós, E. Martí, and J. J. Villanueva, "Symbol recognition by error-tolerant subgraph matching between region adjacency graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 23, no. 10, pp. 1137–1143, 2001.

[4] V. Nastase, R. Mihalcea, and D. R. Radev, "A survey of graphs in natural language processing," *Natural Language Engineering*, vol. 21, pp. 665–698, 2015.

[5] B. Bhattarai, H. Liu, and H. Howie Huang, "Ceci: Compact embedding cluster index for scalable subgraph matching," 01 2019.

[6] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, "Efficient subgraph matching by postponing cartesian products," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: ACM, 2016, pp. 1199–1214. [Online]. Available: http://doi.acm.org/10.1145/2882903.2915236

[7] J. D. Moorman, Q. Chen, T. K. Tu, Z. M. Boyd, and A. Bertozzi, "Filtering methods for subgraph matching on multiplex networks," 12 2018, pp. 3980–3985.

[8] X. Ren and J. Wang, "Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs," *Proc. VLDB Endow.*, vol. 8, no. 5, pp. 617–628, Jan. 2015. [Online]. Available: http://dx.doi.org/10.14778/2735479.2735493

[9] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, and J. Cheng, "G-miner: An efficient task-oriented graph mining system," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: ACM, 2018, pp. 32:1–32:12. [Online]. Available: http://doi.acm.org/10.1145/3190508.3190545

[10] B. Du, S. Zhang, N. Cao, and H. Tong, "First: Fast interactive attributed subgraph matching," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '17. New York, NY, USA: ACM, 2017, pp. 1447–1456. [Online]. Available: http://doi.acm.org/10.1145/3097983.3098040

[11] Y. Tian, R. C. Mceachin, C. Santos, D. J. States, and J. M. Patel, "Saga: A subgraph matching tool for biological graphs," *Bioinformatics*, vol. 23, no. 2, pp. 232–239, Jan. 2007. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btl571

[12] Y. Tian and J. M. Patel, "Tale: A tool for approximate large graph matching," in *2008 IEEE 24th International Conference on Data Engineering*, April 2008, pp. 963–972.

[13] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad, "Fast best-effort pattern matching in large attributed graphs," in *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '07. New York, NY, USA: ACM, 2007, pp. 737–746. [Online]. Available: http://doi.acm.org/10.1145/1281192.1281271

[14] X. Yan, P. S. Yu, and J. Han, "Graph indexing: A frequent structure-based approach," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '04. New York, NY, USA: ACM, 2004, pp. 335–346. [Online]. Available: http://doi.acm.org/10.1145/1007568.1007607

[15] A. Fischer, C. Y. Suen, V. Frinken, K. Riesen, and H. Bunke, "Approximation of graph edit distance based on hausdorff matching," *Pattern Recogn.*, vol. 48, no. 2, pp. 331–343, Feb. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.patcog.2014.07.015

[16] N. Shervashidze, P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, and K. M. Borgwardt, "Weisfeiler-lehman graph kernels," *J. Mach. Learn. Res.*, vol. 12, pp. 2539–2561, Nov. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1953048.2078187

[17] M. Ou, P. Cui, J. Pei, Z. Zhang, and W. Zhu, "Asymmetric transitivity preserving graph embedding," in *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: ACM, 2016, pp. 1105–1114. [Online]. Available: http://doi.acm.org/10.1145/2939672.2939751

[18] A. Grover and J. Leskovec, "Node2vec: Scalable feature learning for networks," in *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: ACM, 2016, pp. 855–864. [Online]. Available: http://doi.acm.org/10.1145/2939672.2939754

[19] R. Pienta, A. Tamersoy, H. Tong, and D. H. Chau, "Mage: Matching approximate patterns in richly-attributed graphs," *2014 IEEE International Conference on Big Data (Big Data)*, pp. 585–590, 2014.

[20] A. Khan, Y. Wu, C. C. Aggarwal, and X. Yan, "Nema: fast graph search with label similarity," in *Proceedings of the 39th international conference on Very Large Data Bases*, ser. PVLDB'13. VLDB Endowment, 2013, pp. 181–192. [Online]. Available: http://dl.acm.org/citation.cfm?id=2448948.2448952

[21] P. Zhao and J. Han, "On graph query optimization in large networks," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 340–351, Sep. 2010. [Online]. Available: http://dx.doi.org/10.14778/1920841.1920887

[22] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, "Taming verification hardness: An efficient algorithm for testing subgraph isomorphism," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 364–375, Aug. 2008. [Online]. Available: http://dx.doi.org/10.14778/1453856.1453899

[23] J. Cheng, Y. Ke, W. Ng, and A. Lu, "Fg-index: Towards verification-free query processing on graph databases," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '07. New York, NY, USA: ACM, 2007, pp. 857–872. [Online]. Available: http://doi.acm.org/10.1145/1247480.1247574

[24] P. Zhao, J. X. Yu, and P. S. Yu, "Graph indexing: Tree + delta ¿ graph," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, ser. VLDB '07. VLDB Endowment, 2007, pp. 938–949. [Online]. Available: http://dl.acm.org/citation.cfm?id=1325851.1325957

[25] D. G. Corneil and C. C. Gotlieb, "An efficient algorithm for graph isomorphism," *J. ACM*, vol. 17, no. 1, pp. 51–64, Jan. 1970. [Online]. Available: http://doi.acm.org/10.1145/321556.321562

[26] S. Zhang, J. Yang, and W. Jin, "Sapper: subgraph indexing and approximate matching in large graphs," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 1185–1194, 2010.

[27] Huahai He and A. K. Singh, "Closure-tree: An index structure for graph queries," in *22nd International Conference on Data Engineering (ICDE'06)*, April 2006, pp. 38–38.

IX. APPENDIX

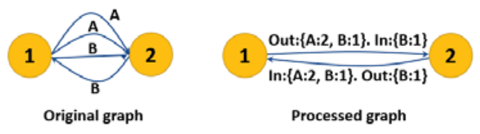


Fig. 11. Graph transformation for G-FINDER.

When we run G-FINDER on multigraph, we first convert the original graph into a directed graph. Figure 11 gives a simple example. In the original graph, there are 3 edges from $node_1$ to $node_2$ whose labels are A, A and B, and 1 edge from $node_2$ to $node_1$ whose label is B. After the transformation, there will be only two edges between $node_1$ and $node_2$. For example, the directed edge from $node_1$ to $node_2$ has attribute $Out : \{A : 2, B : 1\}.In : \{B : 1\}$. This means that there are two edges labeled "A" and one edge labeled "B" from $node_1$ to $node_2$. And one edge labeled "B" from $node_2$ to $node_1$.