IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received March 13, 2021, accepted April 4, 2021, date of publication April 15, 2021, date of current version April 29, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3073494

An Efficient Subgraph Isomorphism Solver for Large Graphs

ZUBAIR ALI ANSARI[1], JAHIRUDDIN[1], AND MUHAMMAD ABULAISH[2], (Senior Member, IEEE)
[1]Department of Computer Science, Jamia Millia Islamia (A Central University), New Delhi 110025, India
[2]Department of Computer Science, South Asian University, New Delhi 110021, India

Corresponding author: Zubair Ali Ansari (zuberaliansari@gmail.com)

This work was supported by the Visvesvaraya Ph.D. Scheme , Ministry of Electronics and Information Technology (MeitY), Government of India under Grant MEITY-PHD-1426.

ABSTRACT For a given pair of pattern and data graphs, the subgraph isomorphism finding problem locates all instances of the pattern graph into the data graph. For a given subgraph isomorphic image of the pattern graph in a data graph, the set of all ordered pairs of the pattern graph's vertices and their respective images data graph is called an embedding. Many solvers, such as $\text{Turbo}_{\text{ISO}}$, `Glasgow`, and `VF3` exist in the literature for subgraph isomorphism finding problem. Though each solver aims to minimize computing costs in its own way, computational efficiency is still a central issue for the subgraph isomorphism finding problem. In this paper, we present the development of an efficient solver, `SubGlw`, for subgraph isomorphism finding which first decomposes data graph into small-size candidate subgraphs using a ranking function and then searches the embeddings of the pattern graph in each of them separately. The ranking function is designed in such a way that it minimizes both number and size of the candidate subgraphs. The performance of `SubGlw` is empirically evaluated and compared with two state-of-the-art subgraph isomorphism solvers – `SubISO` and `Glasgow` over three benchmark datasets – `Yeast`, `Human`, and `Hprd`. The experimental findings reveal that `SubGlw` performs significantly better in terms of both *embedding count* and *execution time*. We have also presented an analysis for identifying *saddle point*, which is a timeout at which our solver achieves maximum embeddings in least execution time. This analysis provides a better understanding for parameter settings. The source codes of `SubGlw` can be downloaded from https://github.com/ZubairAliIgraph/SubGlw-master.

INDEX TERMS Graph mining, subgraph isomorphism, subgraph isomorphism solver, eccentricity, embedding, graph decomposition, saddle point.

I. INTRODUCTION

A graph is a well-known non-linear data structure which is used in many application areas, such as chemistry, biology, network science, and pattern recognition. Even though the process of finding subgraph isomorphism is computationally expensive, querying subgraph isomorphism becomes an inevitable task. Subgraph isomorphism problem has applications in many growing research areas, such as graph databases [1], program similarity comparison [2], compiler design [3], bioinformatics [4], intelligence analysis [5], and pattern recognition [6]. Theoretically subgraph isomorphism finding problem is to identify all instances of a pattern graph in a given data graph, wherein generally the size (number of vertices) of the pattern graph is very small in comparison to the size of the data graph. There are many variants of the subgraph isomorphism finding problem, in which identifying induced and non-induced form of subgraph isomorphism is more popular. The induced subgraph isomorphism solver aims to find the exact isomorphic image of the pattern graph in the data graph. On the other hand, non-induced subgraph isomorphism solver aims to find the isomorphic image of the pattern graph that can have some extra edges. In this paper, we consider both forms of the subgraph isomorphism problem.

SUBGRAPH ISOMORPHISM SOLVER AND DEALING WITH LARGE GRAPHS

Many subgraph isomorphism solvers exist in the literature that use different graph theory concepts to solve

The associate editor coordinating the review of this manuscript and approving it for publication was Moussa Ayyash.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

subgraph isomorphism problem. VF2, QuickSI, RI, VF3, and Glasgow are the examples of some state-of-the-art subgraph isomorphism solvers. However, most of them are inefficient when dealing with large data graphs. Moreover, these solvers also show exponential behavior for some set of pattern graph and data graph pairs [1]. Therefore, we have designed an algorithm based on the broader sense of the divide-and-conquer problem-solving paradigm which divides data graph into several small-size candidate subgraphs, which may contain one or more embeddings of the pattern graph. The graph decomposition is done in such a way that the size and count of the candidate subgraphs are optimal. After decomposition, the subgraph isomorphism solver identifies all embeddings of the pattern graph into the candidate subgraphs and combines them together to form the solution of the subgraph isomorphism finding problem.

A. OUR CONTRIBUTIONS

In this paper, we have proposed a subgraph isomorphism finding solver SubGlw. In line with [7], the proposed solver decomposes a data graph into several candidate subgraphs using a ranking function, which optimizes the size and count of the candidate subgraphs. To decompose data graph into candidate subgraphs, we first determine the pivot vertex of the pattern graph. Thereafter, we decompose data graph into small-size candidate subgraphs. A pivot vertex is a vertex of the pattern graph which optimizes the ranking function. After decomposing data graph into candidate subgraphs, the list of candidate subgraphs is sorted in ascending order of their size. Thereafter, in line to Glasgow, the proposed SubGlw locates all embeddings of the pattern graph in the candidate subgraphs. We have also presented an analysis to determine *saddle point* for the proposed subgraph isomorphism solver which helps in parameter settings.

In short, the contributions of this paper can be summarized as follows:

- Development of a subgraph isomorphism solver, SubGlw, to find at most *n* embeddings of a given pattern graph into a data graph within a specified timeout period.
- Introducing a ranking function to decompose a data graph into optimal-size candidate subgraphs.
- Imposing ordering on candidate subgraphs (increasing order of their size) to make SubGlw more efficient.
- An analysis to determine *saddle point* for the proposed solver which helps in parameter settings.

The rest of the paper is structured as follows. Section II presents a detailed review of the research works on subgraph isomorphism finding problem. Section III presents some preliminaries, including definitions, problem statement, and a lemma. Section IV presents the procedural details and pseudo-codes of the proposed solver. Section V presents the experimental setup and results. It also presents a comparative analysis of SubGlw with SubISO and Glasgow solvers. Section V-D presents an analysis of the results to determine saddle points. Finally, section VI concludes the paper with future directions of research.

II. RELATED WORKS

This section presents a brief review of the existing literature on subgraph isomorphism problem. Many subgraph isomorphism solvers have been proposed in literature to find embeddings of a given pattern graph into a data graph. Ullmann [8] introduced the first algorithm to address the subgraph isomorphism problem which was followed by a number of state-of-the-art subgraph isomorphism solvers viz. VF2 [9], LAD [10], RI [4], Glasgow [11], L2G [12], and VF3 [13]. Out of these, VF2, L2G, and VF3 are designed for induced subgraph isomorphism, whereas LAD, RI, and Glasgow work for both induced and non-induced subgraph isomorphisms. The VF2 uses state-space representation (SSR) and five sets of rules to prune the search space tree [9], and it uses a special data structure to reduce memory requirements at the time of exploring search space. In [4], authors proposed subgraph isomorphism search algorithm RI and reported that it finds all isomorphic images of pattern graph in the target graph, and they also compared it with some state-of-the-art subgraph isomorphism solvers on biochemical graph datasets. The Glasgow subgraph isomorphism solver supports both induced and non-induced variants of subgraph isomorphism problem [11], wherein *solution biased search* was introduced as a choice of backtracking search. Glasgow uses the degree of a vertex to determine the proportion of effort for solving a subgraph isomorphism problem. Glasgow is sequential, but it can easily be converted into a parallel approach to speedup the matching process of the subgraph isomorphism problem. The VF3 solver is an improved version of VF2 which is compared with RI, VF2, LAD, and L2G, and shows best performance on dense graphs [13]. The Sum$_{\mathrm{ISO}}$ [14] is another approach which uses modular decomposition for compressing pattern and data graphs. Similarly, BoostIso [15] compresses pattern and data graphs by exploiting vertex-based relationships and then applies subgraph isomorphism solver to get an improved performance.

Many researchers proposed indexing-based new framework to solve the subgraph isomorphism finding problem. Some of the indexing-based existing solvers are GADDI [16], GraphQL [17], SPath [18], STW [19], CFL-Match [20], QuickSI [21], and Turbo$_{\mathrm{ISO}}$ [22]. Out of these, GraphQL, STW, CFL-Match, and Turbo$_{\mathrm{ISO}}$ solvers work for non-induced subgraph isomorphism. The SND (Scoring-based Neighborhood Dominance) is a generalized filtering technique that may be used in subgraph isomorphism solvers to improve their performance [23]. It uses two parameters – *score* and *neighborhood* functions. The LAD is a particular case of SND where filtering operation is based on local alldifferent constraints [10]. In this work, the authors modeled the subgraph isomorphism problem into a constraint satisfaction framework and reported that their filtering technique improves the performance of the subgraph isomorphism solvers. In [1], the authors proposed a graph generating method that generates "really hard" instances of pattern and data graphs for induced and non-induced subgraph isomorphism solvers. The pattern and data graphs have

a fixed number of vertices, and the pattern graphs of these instances have tens of edges, while the data graph contains a couple of hundred edges. They presented the results of `VF2`, `LAD`, and `Glasgow` on the generated instances of the pattern and data graph pairs. Authors of [24] extended the work of [1] and compared subgraph isomorphism solvers `RI`, `VF3`, `Glasgow`, and `LAD` on broader datasets.

III. PRELIMINARIES

This section presents a formal definition of some basic concepts and a lemma related to subgraph isomorphism. It also explains the problem statement and presents a list of notations and their brief descriptions as given in table 1.

TABLE 1. Notations and their descriptions.

Notation	Description
G_p	Pattern graph
G_d	Data graph
$\#V$	Size of a graph with vertex set V
$\Delta(p)$	Degree of vertex p
$l(p)$	Label of vertex p
$\tilde{\Delta}_{\mathcal{N}}(p)$	Highest neighborhood degree
$\delta(p,p')$	Distance between vertex p and p'
$\epsilon(p)$	Eccentricity of vertex p
$\mathcal{N}_\epsilon(p)$	ϵ-neighborhood of vertex p
$\mathcal{C}(p)$	Set of candidate matches of vertex p in G_d
$\hat{p}$	Vertex p as a pivot vertex
$\mathcal{S}$	Candidate subgraph of G_q in G_d
$\mathcal{M}$	An embedding of G_p in G_d
$\mathcal{M}_{pd}$	Set of all embeddings of G_p in G_d
$\mathcal{L}_{\mathcal{S}}$	Ordered list of candidate subgraphs
$\mathcal{L}_{supp}$	Set of pairs of supplemental graphs
$\bar{\Delta}(G,p)$	Degree sequence of the neighboring vertices of p in G
$\mathcal{F}$	Set of nogoods

Definition 1 (`Labeled Graph`)*:* A labeled graph $G = \ < V, E, \Omega, l >$ is a four-tuple, where V is a non-empty set of vertices representing the entities or objects, $E \subseteq V \times V$ is the set of edges representing links between vertex pairs, Ω is the set of vertex labels, and $l : V \rightarrow \Omega$ is a *surjective* function which assigns each vertex of V a unique label from Ω.

Definition 2 (`Size of a Graph`)*:* The size of a graph $G = \ < V, E >$ is represented by $\#V$, and it is defined as the number of vertices in G.

Definition 3 (`Labeled Subgraph`)*:* A labeled graph $G_p = \ < V_p, E_p, \Omega_p, l_p >$ is a subgraph of another labeled graph $G_d = \ < V_d, E_d, \Omega_d, l_d >$ if $V_p \subseteq V_d$, $E_p \subseteq E_d$, $\Omega_p \subseteq \Omega_d$, and $l_p(p) = l_d(p), \forall p \in V_p$.

Definition 4 (`Neighbor Set`)*:* The neighbor set of a vertex p of a graph G is denoted by $\mathcal{N}(p)$, and it is the set of all adjacent vertices of p in G.

Definition 5 (`Degree`)*:* The degree of a vertex p of a graph G is denoted by $\Delta(p)$, and it is the number of vertices in the neighbor set of p in G.

Definition 6 (`Degree Sequence of a Vertex`)*:* The degree sequence of a vertex p of a graph G is denoted by $\bar{\Delta}(G, p)$, and it is the ordered sequence of the degree of its neighboring vertices in G.

Definition 7 (`Highest Neighborhood Degree`)*:* The highest neighborhood degree of a vertex p of a graph G is denoted by $\tilde{\Delta}_{\mathcal{N}}(p)$, and it is the highest degree of a vertex in the neighbor set of p.

Definition 8 (`Distance`)*:* The distance between two vertices p and p' of a graph G is denoted by $\delta(p, p')$, and it is the length of the shortest path between p and p'.

Definition 9 (`ϵ-Neighborhood`)*:* The ϵ-neighborhood of a vertex p of graph G is represented by $\mathcal{N}_\epsilon(p)$, and it is the set of all vertices of G whose distance from p is less than or equal to ϵ.

Definition 10 (`Eccentricity`)*:* The eccentricity of a vertex p of graph G is represented by $\epsilon(p)$, and it is the maximum distance between p and all other vertices of G.

Definition 11 (`Supplemental Graph`)*:* The supplemental graph of a graph G is denoted by $G^{[c,l]}$, and it has the same vertex set as the graph G, and an edge exists between any two vertices in $G^{[c,l]}$ if there are at least c paths of length l between them in G.

Definition 12 (`Induced Subgraph Isomorphism`)*:* A graph $G_p = \ < V_p, E_p, \Omega_p, l_p >$ is induced subgraph isomorphic to another graph $G_d = \ < V_d, E_d, \Omega_d, l_d >$ if there exists an injective function $g : V_p \rightarrow V_d$, along with the following three conditions:

1) $\forall p \in V_p, l_p(p) = l_d(g(p))$
2) $\forall (p, p') \in E_p \Rightarrow \exists\, (g(p), g(p')) \in E_d$
3) $\forall (p, p') \notin E_p \Rightarrow \exists\, (g(p), g(p')) \notin E_d$.

Definition 13 (`Non-Induced Subgraph Isomorphism`)*:* A graph $G_p = \ < V_p, E_p, \Omega_p, l_p >$ is subgraph isomorphic to another graph $G_d = \ < V_d, E_d, \Omega_d, l_d >$ if there exists an injective function $f : V_p \rightarrow V_d$, along with the following two conditions:

1) $\forall p \in V_p, l_p(p) = l_d(f(p))$, and
2) $\forall (p, p') \in E_p \Rightarrow \exists\, (f(p), f(p')) \in E_d$.

It should be noted that non-induced subgraph isomorphic image of G_p can have additional edges. In this way, every induced subgraph isomorphic image is also a non-induced subgraph isomorphic image, but vice-versa is not always true.

Definition 14: `Embedding`: For a given pattern graph G_p and data graph G_d, an embedding $\mathcal{M}$ of G_p in G_d is the set of all ordered pairs $< p, f(p) >$, where $p \in V_p$ is a vertex in G_p and $f(p) \in V_d$ is an isomorphic image of p in G_d.

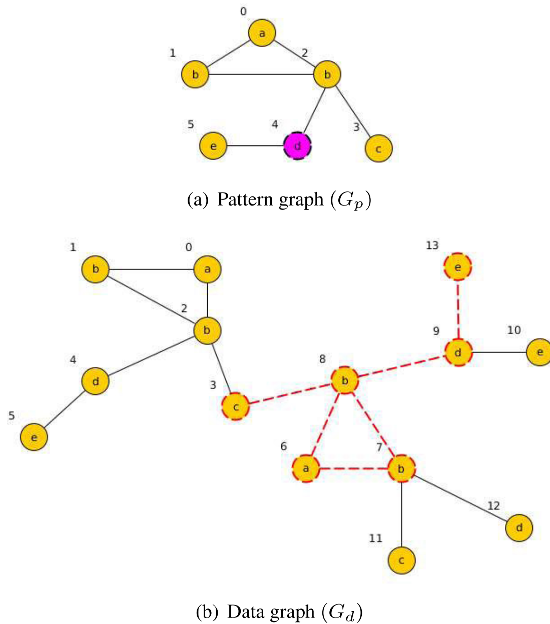
Definition 15: `Subgraph Isomorphism Problem`: For a pair of pattern and data graphs $< G_p, G_d >$, the subgraph isomorphism problem is to determine whether an embedding of G_p exists in G_d or not.

`Problem statement`: For a pair of pattern and data graphs, $< G_p, G_d >$, enumerate at most n embeddings of G_p in G_d within a given timeout t.

Our study considers pattern graphs of small size having tens of vertices, and data graph of large size consisting of thousands of vertices. We have taken into account simple, connected, undirected, and vertex labeled graphs. However,

we have limited the time and embedding count to deal with the *output crisis* problem flagged in [25].

Figures 1(a) and 1(b) present a pattern graph G_p and a data graph G_d in which the embedding $\{<0,6>, <1,7>, <2,8>, <3,3>, <4,9>, <5,13>\}$ is an induced as well as non-induced subgraph isomorphic image of G_p in G_d. If there exist an edge (3, 6) in G_d, then this embedding becomes non-induced isomorphic image, but it will not be an induced subgraph isomorphic image of G_p in G_d anymore.



(a) Pattern graph (G_p)

(b) Data graph (G_d)

FIGURE 1. An exemplar (a) pattern graph with pivot vertex 4, and (b) data graph.

As stated earlier, SubGlw first decomposes data graph G_d into candidate subgraphs and then searches the embeddings of the pattern graph in each of them separately. For each vertex p of the pattern graph G_p, there exists $|\mathcal{C}(p)|$ candidate subgraphs, where $\mathcal{C}(p)$ is the set of candidate matches of p in G_d. However, an optimal size of the subgrpahs is determined by the following lemma.

Lemma 1: For a given vertex p of the pattern graph G_p with eccentricity ϵ, if q is a match of p in the data graph G_d, then the maximum size of the respective candidate subgraph of G_d is at most the number of vertices in the ϵ-neighborhood of q.

Proof: It is given that ϵ is the eccentricity of vertex p of pattern graph G_p.
$\Rightarrow$ For all other vertices p' of G_p, $\delta(p, p') \leq \epsilon$.
$\because$ The distance between two vertices of the pattern graph G_p is less than or equal to the distance between respective isomorphic images in the data graph G_d.
$\therefore$ $\delta(q, q') \leq \epsilon$, where the vertices q and q' of G_d are matches of p and p' vertices of G_p, respectively.
$\Rightarrow q' \in \mathcal{N}_\epsilon(q)$
Thus, all candidate matches of the pattern graph's vertices lie in the corresponding $\mathcal{N}_\epsilon(q)$. In other words, maximum size of the candidate subgraph is at most the number of vertices in ϵ-neighborhood of vertex q of G_d. □

It should be noted that SubGlw aims to minimize both the number and size of the candidate subgraphs for efficiency purpose. To this end, we have defined a ranking function given in equation 1 that optimizes both the count and size of the candidate subgraphs.

$$\hat{p} = \arg\min_{p \in V_p}\{|\mathcal{C}(p)| \times \epsilon(p)\} \tag{1}$$

It is important to note that the correct choice of starting vertex or pivot vertex ($\hat{p}$) in the pattern graph is one of the critical steps in the subgraph isomorphism finding problem. In this study, a node which minimizes the ranking function defined in equation 1 is considered as the pivot vertex ($\hat{p}$).

IV. PROPOSED METHOD

This section presents a detailed description of our proposed subgraph isomorphism solver, SubGlw, to identify all embeddings of a pattern graph G_p into the data graph G_d. Algorithm 1 presents the pseudo-code of the SubGlw solver. The core functions of SubGlw are GetPivotVertex( ), CandidateSubgraphFinding( ), BuildSupplementalGraphList ( ), Initialize( ), CountAllDistinct( ), and EmbSearch( ). The GetPivotVertex( ) function is used to select a pivot vertex $\hat{p}$ of G_p, and a set of all candidate matches of $\hat{p}$ in G_d using the ranking function defined in equation 1. The CandidateSubgraphFinding( ) function returns the sorted list of candidate subgraphs. BuildSupplementalGraphList( ) builds seven pairs of supplemental graphs for each pair of pattern graph and candidate subgraph. The Initialize( ) function initializes candidate set for each vertex of the pattern graph G_p. CountAllDistinct( ) ensures different constraints, such as all vertices of pattern graph G_p must have at least one distinct match in the data graph G_d. The EmbSearch( ) function is used to get an embedding of G_p in the candidate subgraph $\mathcal{S}$. Procedural details of these functions are further described in the following sub-sections.

A. PIVOT VERTEX SELECTION

As described earlier, the pivot vertex of the pattern graph G_p is a vertex which optimizes the ranking function defined in equation (1). It is used to decompose the data graph G_d into candidate subgraphs. It is also used as the starting vertex to initiate the search process of locating subgraph isomorphisms of G_p in the candidate subgraphs of G_d. Algorithm 2 presents the functional details of this process formally. In this algorithm, steps 1 to 8 are used to create a candidate set for each vertex of G_p. We have used label, degree, and highest neighborhood degree of a vertex to generate its candidate set. Step 9 orders V_p based on the element counts of the respective candidate set. This ordering helps to select the first three vertices in $\mathcal{L}'$. Steps 11 to 20 are used to refine the candidate sets of the selected three vertices using a list of labels of the neighboring vertices. Step 21 identifies the pivot vertex $\hat{p}$. In figure 1(a), vertex 4 is the pivot vertex of the

Algorithm 1 SubGlw(G_p, G_d)

Input : G_p: pattern graph, G_d: data graph
Output: A set $\mathcal{M}_{pd}$ of all embeddings of G_p in G_d

```
1  G_p ← RemoveIsolatedVertices(G_p)
2  < p̂, C(p̂) > ← GetPivotVertex(G_p, G_d)
3  M_pd ← φ
4  ε ← Eccentricity(p̂)
5  L_S ← CandidateSubgraphFinding(G_p, G_d, C(p̂), ε)
6  foreach S ∈ L_S do
7      if #V_p > #V_S then
8          continue
9      end
10     L_supp ← BuildSupplementalGraphList(G_p, S)
11     while embedding of G_p exist in S do
12         C(G_p) ← Initialize(V_p, V_S, L_supp)
13         if ¬CountAllDistinct(C(G_p)) then
14             continue
15         end
16         M ← φ
17         M ← EmbSearch(L_supp, C(G_p), M)
18         M_pd ← M_rs ∪ {M}
19     end
20 end
21 return M_pd
```

pattern graph G_p determined by algorithm 2. If we choose any other vertex to start, then either size or number of candidate subgraphs may increase.

B. CANDIDATE SUBGRAPHS FINDING

The candidate subgraphs finding process is used to decompose data graph G_d into several small size subgraphs. For each matching vertex of the pivot vertex in G_d using lemma 1, it identifies a candidate subgraph. Algorithm 3 presents the candidate subgraphs finding process formally. It finds a candidate subgraph for each $q \in \mathcal{C}(\hat{p})$, and finally it returns the sorted list of candidate subgraphs $\mathcal{L}_{\mathcal{S}}$ based on their size.

In the exemplar graphs given in figure 1, the pivot vertex is 4, the eccentricity of the pivot vertex is 2, and $\mathcal{C}(\hat{p})$ has only two vertices 4, 9. Therefore by using lemma 1, only two candidate subgraphs ($\mathcal{S}_1$ and $\mathcal{S}_2$) exist that are shown in figure 2. The ordered $\mathcal{L}_{\mathcal{S}}$ contains $\mathcal{S}_1$ followed by $\mathcal{S}_2$ because the size of $\mathcal{S}_1$ is less than the size of $\mathcal{S}_2$.

C. SUPPLEMENTAL GRAPH BUILDING AND EMBEDDING SEARCH

In order to build supplemental graph and searching embeddings, we have used BuildSupplementalGraphList( ), Initialize( ), CountAllDistinct( ), EmbSearch( ) and Assign( ) functions proposed in [26]. In algorithm 4, $G_p^{[c,l]}$ denotes a supplemental graph having the same vertex set as G_p, and an edge exists between any two vertices in $G_p^{[c,l]}$ if there is at least c paths of length l between them in pattern graph G_p. Similarly, $\mathcal{S}^{[c,l]}$ is the supplemental graph of the candidate subgraph $\mathcal{S}$. In order to avoid computation overhead we have restricted the values of c and l to maximum 3. Algorithm 4 creates seven pairs of supplemental graphs for pattern and candidate subgraph pair, wherein the first supplemental graph pair is the graphs themself. The rest of the six pairs of supplemental graphs are shown in figures 3, 4, 5, and 6. These supplemental pairs of graphs help to find more appropriate matches for the specified pattern graph vertices.

Algorithm 2 GetPivotVertex(G_p, G_d)

Input : G_p: pattern graph, G_d: data graph
Output: The pivot vertex $\hat{p}$ and its candidate matches $\mathcal{C}(\hat{p})$

```
1  foreach p ∈ V_p do
2      C̃(p) ← φ
3      foreach q ∈ V_d do
4          if l(p) = l(q) & Δ(p) ≤ Δ(q) &
           Δ̃_N(p) ≤ Δ̃_N(q) then
5              C̃(p) ← C̃(p) ∪ {q}
6          end
7      end
8  end
9  L ← Ordered(V_p)// based on number of
       matches of vertices.
10 L' ← Select first three member of (L)
11 foreach p ∈ L' do
12     C(p) ← φ
13     l_p ← l(N(p))
14     foreach v ∈ C̃(p) do
15         l_q ← l(N(q))
16         if l_p ⊆ l_q then
17             C(p) ← C(p) ∪ {q}
18         end
19     end
20 end
21 p̂ ← arg min_{p∈L'} {|C(p)| × ε(p)}
   // provided #C(p) × ε(p) ≠ 0, ∀p ∈ L'
22 return < p̂, C(p̂) >
```

Algorithm 3 CandidateSubgraphFinding(G_p, G_d, $\mathcal{C}(\hat{p})$, ϵ)

Input : G_p: pattern graph, G_d: data graph, $\mathcal{C}(\hat{p})$: candidate match of $\hat{p}$, ϵ: eccentricity of $\hat{p}$
Output: $\mathcal{L}_{\mathcal{S}}$: Ordered list of candidate subgraphs.

```
1  L_S ← φ
2  foreach q ∈ C(p̂) do
3      S ← N_ε(q)
4      L_S ← L_S ∪ {S}
5  end
6  Ordered(L_S)// based on #V_S.
7  return L_S
```

Algorithm 4 BuildSupplementalGraphList(G_p, $\mathcal{S}$)

Input : G_p: pattern graph, $\mathcal{S}$: candidate subgraph
Output: Set of pairs of supplemental graphs: $\mathcal{L}_{supp}$

1 $\mathcal{L}_{supp} \leftarrow \{(G_p, \mathcal{S})\}$
2 $\mathcal{A}_p \leftarrow$ Adjacency matrix of G_p
3 $\mathcal{A}_\mathcal{S} \leftarrow$ Adjacency matrix of $\mathcal{S}$
4 Compute matrices $\mathcal{A}_p^2$, $\mathcal{A}_p^3$, $\mathcal{A}_\mathcal{S}^2$, $\mathcal{A}_\mathcal{S}^3$
5 **foreach** $c \in \{1, 2, 3\}$ **do**
6   **foreach** $l \in \{2, 3\}$ **do**
7     $G_p^{[c,l]}[i, j] \leftarrow 1$, **if** $\mathcal{A}_p^l[i, j] \geq c$, where $0 \leq i, j \leq rowCount(\mathcal{A}_p)$
8     $\mathcal{S}_p^{[c,l]}[i, j] \leftarrow 1$, **if** $\mathcal{A}_\mathcal{S}^l[i, j] \geq c$, where $0 \leq i, j \leq rowCount(\mathcal{A}_\mathcal{S})$
9     $\mathcal{L}_{supp} \leftarrow \mathcal{L}_{supp} \cup \{(G_p^{[c,l]}, \mathcal{S}_p^{[c,l]})\}$
10   **end**
11 **end**
12 **return** $\mathcal{L}_{supp}$

Algorithm 5 Initialize(V_p, $V_\mathcal{S}$, $\mathcal{L}_{supp}$)

Input : V_p: vertex set of pattern graph, $V_\mathcal{S}$: vertex set of candidate subgraph, $\mathcal{L}_{supp}$: set of supplemental graph pairs
Output: $\mathcal{C}(G_p)$: set of candidate sets.

1 $\mathcal{C}(G_p) \leftarrow \phi$
2 **foreach** $p \in V_p$ **do**
3   $\mathcal{C}(p) \leftarrow \{q \in V_\mathcal{S} : l(p) = l(q)$ *and* $\forall (g_1, g_2) \in \mathcal{L}_{supp}, \bar{\Delta}(g_1, p) \preceq \bar{\Delta}(g_2, q)\}$
4 **end**
5 $\mathcal{C}(G_p) \leftarrow \cup_{p \in V_p}\{\mathcal{C}(p)\}$
6 **return** $\mathcal{C}(G_p)$

Algorithm 6 EmbSearch($\mathcal{L}_{supp}$, $\mathcal{C}(G_p)$, $\mathcal{M}$)

Input : $\mathcal{L}_{supp}$: set of supplemental graph pairs, $\mathcal{C}(G_p)$: set of candidate sets, $\mathcal{M}$: an embedding
Output: Fail $\mathcal{F}$ or Success $\mathcal{M}$

1 **if** $\mathcal{C}(G_p) = \phi$ **then**
2   **return** *Success* $\mathcal{M}$
3 **end**
4 $\mathcal{C}(p) \leftarrow$ Select a least sized candidate set from $\mathcal{C}(G_p)$.
5 $\mathcal{F} \leftarrow \{p\}$
6 **foreach** $q \in Ordered(\mathcal{C}(p))$ **do**
7   $\mathcal{C}'(G_p) \leftarrow Copy(\mathcal{C}(G_p))$
8   **switch** $Assign(\mathcal{L}_{supp}, \mathcal{C}'(G_p), p, q)$ **do**
9   **case** *Fail* $\mathcal{F}'$
10       $\mathcal{F} \leftarrow \mathcal{F} \cup \mathcal{F}'$
11   **case** *Success*
12       $\mathcal{M} \leftarrow \mathcal{M} \cup \{< p, q >\}$
13   **switch** $EmbSearch(\mathcal{L}_{supp}, \mathcal{C}'(G_p) - \mathcal{C}(p), \mathcal{M})$ **do**
14   **case** *Success*
15   **return** *Success*
16   **case** *Fail* $\mathcal{F}'$
17   **if** $\nexists p' \in \mathcal{F}'$ *such that* $\mathcal{C}(p') \neq \mathcal{C}'(p')$ **then**
18   Remove $\{< p, q >\}$ from $\mathcal{M}$
19   **return** *Fail* $\mathcal{F}'$
20   **end**
21           $\mathcal{F} \leftarrow \mathcal{F} \cup \mathcal{F}'$
22   **end**
23   **endsw**
24   **end**
25   **endsw**
26 **end**
27 **return** *Fail* $\mathcal{F}$

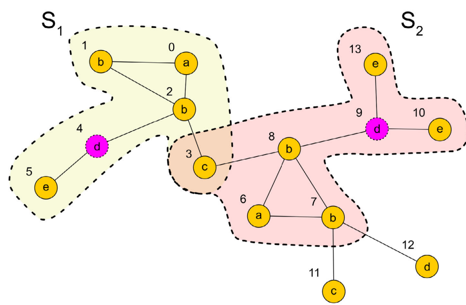
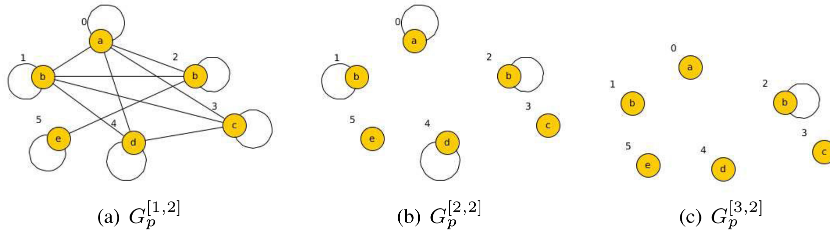


FIGURE 2. Identified candidate subgraphs of pattern graph G_p in data graph G_d.

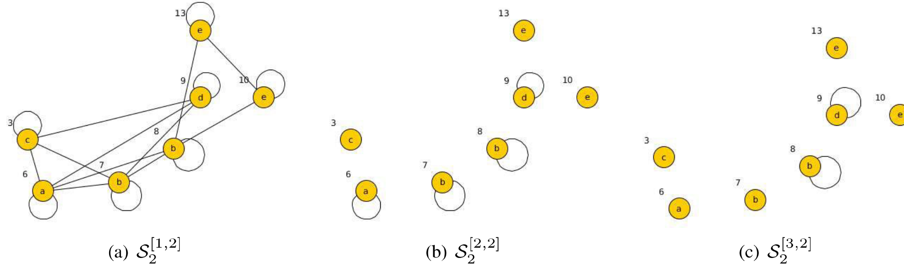
In algorithm 5, we have created and initialized the candidate set for each pattern graph's vertex. For every vertex $p \in V_p$, a vertex $q \in V_\mathcal{S}$ can be a possible match if the labels of p and q are same, and $\bar{\Delta}(g_1, p) \preceq \bar{\Delta}(g_2, q)$ for every pair of supplemental graphs created in algorithm 4, where $\bar{\Delta}(g_1, p)$ is a degree sequence of vertex p in graph g_1. Similarly, $\bar{\Delta}(g_2, q)$ is the degree sequence of vertex q in graph g_2. $\bar{\Delta}(g_1, p)$ precedes $\bar{\Delta}(g_2, q)$ is represented by $\bar{\Delta}(g_1, p) \preceq \bar{\Delta}(g_2, q)$ if there exists a subsequence of $\bar{\Delta}(g_2, q)$ of the same length as $\bar{\Delta}(g_1, p)$ such that each element of this subsequence is greater than or equal to corresponding element of $\bar{\Delta}(g_1, p)$. Finally, $\mathcal{C}(G_p)$ is obtained by taking a union of candidate sets of each pattern graph's vertex. Table 2 presents the degree sequences of the vertex 1 in supplemental graphs of the pattern graph G_p. Similarly, tables 3 and 4 present the degree sequences of vertices 8 and 7 of the supplemental graphs of the candidate subgraph $\mathcal{S}_2$. The vertex 8 does not belong in $\mathcal{C}(1)$ because $\bar{\Delta}(G_p^{[1,2]}, 1) \npreceq \bar{\Delta}(\mathcal{S}_2^{[1,2]}, 8)$ (first column of tables 2 and 3). On the other hand, vertex 7 belongs in $\mathcal{C}(1)$ (tables 2 and 4); therefore, $\mathcal{C}(1) = \{7\}$. Similarly, candidate sets for other pattern graph vertices are created. The elements belonging to candidate sets are as follows: $\mathcal{C}(0) = \{6\}$, $\mathcal{C}(1) = \{7\}$, $\mathcal{C}(2) = \{8\}$, $\mathcal{C}(3) = \{3\}$, $\mathcal{C}(4) = \{9\}$, $\mathcal{C}(5) = \{10, 13\}$.

Algorithm 6 presents the steps to find an embedding of the pattern graph in candidate subgraphs. This algorithm is recursive, and the terminating condition is given in step 1. It first chooses a least sized candidate set $\mathcal{C}(p)$ from $\mathcal{C}(G_p)$; if



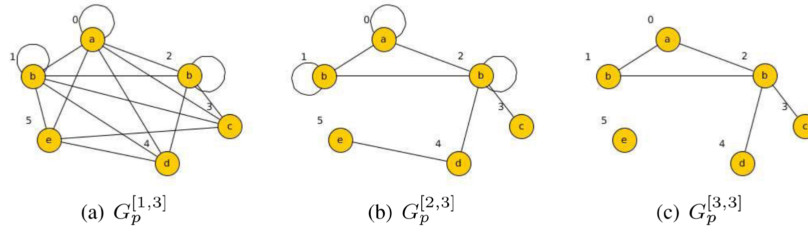
(a) $G_p^{[1,2]}$ (b) $G_p^{[2,2]}$ (c) $G_p^{[3,2]}$

FIGURE 3. Supplemental graphs of path-length 2 corresponding to the exemplar pattern graph G_p.



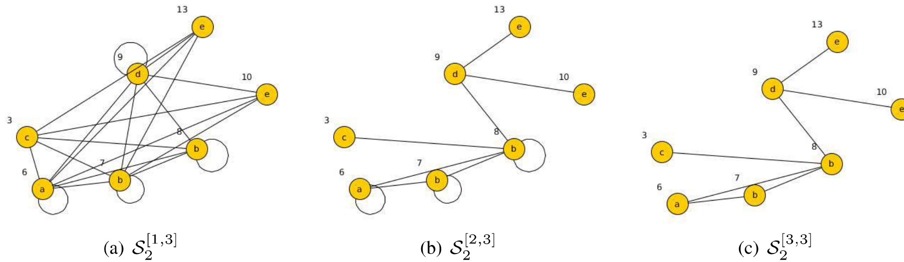
(a) $\mathcal{S}_2^{[1,2]}$ (b) $\mathcal{S}_2^{[2,2]}$ (c) $\mathcal{S}_2^{[3,2]}$

FIGURE 4. Supplemental graphs of path-length 2 corresponding to the candidate subgraph $\mathcal{S}_2$.



(a) $G_p^{[1,3]}$ (b) $G_p^{[2,3]}$ (c) $G_p^{[3,3]}$

FIGURE 5. Supplemental graphs of path-length 3 corresponding to the exemplar pattern graph G_p.



(a) $\mathcal{S}_2^{[1,3]}$ (b) $\mathcal{S}_2^{[2,3]}$ (c) $\mathcal{S}_2^{[3,3]}$

FIGURE 6. Supplemental graphs of path-length 3 corresponding to the candidate subgraph $\mathcal{S}_2$.

TABLE 2. Degree sequences of the pattern graph's vertex 1 in its supplemental graphs.

$\bar{\Delta}(G_p, 1)$	$\bar{\Delta}(G_p^{[1,2]}, 1)$	$\bar{\Delta}(G_p^{[2,2]}, 1)$	$\bar{\Delta}(G_p^{[3,2]}, 1)$	$\bar{\Delta}(G_p^{[1,3]}, 1)$	$\bar{\Delta}(G_p^{[2,3]}, 1)$	$\bar{\Delta}(G_p^{[3,3]}, 1)$
2, 4	5, 5, 5, 6, 6	2	Non	4, 4, 4, 6, 7, 7	4, 4, 6	2, 4

TABLE 3. Degree sequences of the candidate subgraph's vertex 8 in its supplemental graphs.

$\bar{\Delta}(\mathcal{S}_2, 8)$	$\bar{\Delta}(\mathcal{S}_2^{[1,2]}, 8)$	$\bar{\Delta}(\mathcal{S}_2^{[2,2]}, 8)$	$\bar{\Delta}(\mathcal{S}_2^{[3,2]}, 8)$	$\bar{\Delta}(\mathcal{S}_2^{[1,3]}, 8)$	$\bar{\Delta}(\mathcal{S}_2^{[2,3]}, 8)$	$\bar{\Delta}(\mathcal{S}_2^{[3,3]}, 8)$
1, 2, 2, 3	4, 4, 6, 6, 6	2	2	4, 5, 6, 8, 8	1, 3, 4, 4, 6	1, 2, 2, 3

a tie occurs, then it chooses a candidate set of vertex $p \in V_p$ of least degree. Next, it sorts the elements of $\mathcal{C}(p)$ based on their degrees in G_d. Step 6 selects every time a vertex q from the ordered $\mathcal{C}(p)$. Step 8 assign q to p; if *Success* occurres, then it adds $< p, q >$ to embedding $\mathcal{M}$ and recurs to find a match for another pattern graph vertices. The function Assign( ) and recursive call either gets *Success* or returns a set of *nogoods* $\mathcal{F}$. The $\mathcal{F}$ (set of nogoods) is a set of vertices of

TABLE 4. Degree sequences of the candidate subgraph's vertex 7 in its supplemental graphs.

$\bar{\Delta}(\mathcal{S}_2, 7)$	$\bar{\Delta}(\mathcal{S}_2^{[1,2]}, 7)$	$\bar{\Delta}(\mathcal{S}_2^{[2,2]}, 7)$	$\bar{\Delta}(\mathcal{S}_2^{[3,2]}, 7)$	$\bar{\Delta}(\mathcal{S}_2^{[1,3]}, 7)$	$\bar{\Delta}(\mathcal{S}_2^{[2,3]}, 7)$	$\bar{\Delta}(\mathcal{S}_2^{[3,3]}, 7)$
2, 4	5, 5, 6, 6, 6	2	Non	4, 4, 5, 5, 6, 8, 8	4, 4, 6	2, 4

the pattern graph returned by the CountAllDistinct( ) function when all-different constraints are failed. It is used to reduce the search space. If the next recursive call fails to find any match, the current match $< p, q >$ cannot reduce the size of candidate sets for the pattern graph vertices in $\mathcal{F}'$. In this case, we ignore the current assignment and remove $< p, q >$ from embedding $\mathcal{M}$ and backtrack. Finally, if matching is not possible, then we return a set of nogoods combined with nogoods of unsuccessful assignments or nogoods of the next recursive call. Instead of simple backward jumps, the search uses a set of nogoods and restarts [27], [28]. After succeeding or failing of finding the embedding, the search process restarts from the beginning, ensuring not to repeat any portion of the search space that has already been visited.

Algorithm 7 is used to map a candidate set vertex q to pattern graph p. In step 1, it performs the assignment operation. Thereafter, steps 2 to 14 are used to update candidate sets of unmatched vertices of G_p. In the update process (step 3), matched vertex q is removed from the candidate sets of all remaining unmatched vertices of the pattern graph G_p. For each pair of supplemental graphs $(G_1, G_2) \in \mathcal{L}_{supp}$, if any vertex p' is a neighbor of vertex p in G_1, then candidate set $\mathcal{C}(p')$ is updated by taking the common elements of the neighbor of vertex q in G_2 (i.e., $\mathcal{N}(G_2, q)$) with $\mathcal{C}(p')$. If any candidate set gives a wipe-out, then the assignment process fails and returns the nogoods set. Step 13 collects all candidate sets and invokes CountAllDistinct( ) function.

Algorithm 7 Assign($\mathcal{L}_{supp}, \mathcal{C}(G_p), p, q$)

Input : $\mathcal{L}_{supp}$: Set of supplemental graph pairs, $\mathcal{C}(G_p)$: Set of candidate sets, Vertex p, Vertex q
Output: Fail $\mathcal{F}$ or Success

1 $\mathcal{C}(p) \leftarrow \{q\}$
2 $\mathcal{I} \leftarrow \phi$
3 **foreach** $\mathcal{C}(p') \in \mathcal{C}(G_p) - \mathcal{C}(p)$ **do**
4 $\quad \mathcal{C}(p') \leftarrow \mathcal{C}(p') - q$
5 $\quad$ **foreach** $(G_1, G_2) \in \mathcal{L}_{supp}$ **do**
6 $\quad\quad$ **if** $(p, p') \in E(G_1)$ **then**
7 $\quad\quad\quad \mathcal{C}(p') \leftarrow \mathcal{C}(p') \cap \mathcal{N}(G_2, q)$
8 $\quad\quad$ **end**
9 $\quad$ **end**
10 $\quad$ **if** $\mathcal{C}(p') = \phi$ **then**
11 $\quad\quad$ **return** *Fail*$\{p'\}$
12 $\quad$ **end**
13 $\quad \mathcal{I} \leftarrow \mathcal{I} \cup \{p'\}$
14 **end**
15 $\mathcal{C}(G_p) \leftarrow \cup_{p \in \mathcal{I}} \{\mathcal{C}(p)\}$
16 **return** *CountAllDistinct*($\mathcal{C}(G_p)$)

Algorithm 8 presents the steps of the CountAllDistinct( ) function formally. This function ensures all-different constraint; i.e., all vertices of pattern graph G_p must have at least one distinct match in the candidate subgraph $\mathcal{S}$. First, it initializes the variables $\mathcal{F}, \mathcal{H}, \mathcal{A}$, and n, where $\mathcal{F}$ is the set of nogoods, $\mathcal{H}$ is a `Hall set`, i.e., the set of all vertices to be excluded from other candidate sets, and $\mathcal{A}$ is a set that accumulates all elements of processed candidate sets, and n represents the number of candidate sets contributing to set $\mathcal{A}$. After initialization, $\mathcal{C}(G_p)$ is ordered in ascending order based on the size of its members. Thereafter, for each $\mathcal{C}(p)$ in Ordered($\mathcal{C}(G_p)$), the following steps are performed: remove any element present in existing Hall set $\mathcal{H}$ from $\mathcal{C}(p)$ (step 3); update sets $\mathcal{A}$ and $\mathcal{F}$ and make increment in variable n (step 4). If candidate set $\mathcal{C}(p)$ is wiped out or the cardinality of set $\mathcal{A}$ is less than n, then return set $\mathcal{F}$ (steps 5 and 6). If cardinality of set $\mathcal{A}$ is equal to n, then update Hall set $\mathcal{H}$ and reset $\mathcal{A}$ and n (steps 8, 9). If for any subgraph isomorphism problem candidate sets are $\mathcal{C}(p_1) = \{q_1\}, \mathcal{C}(p_2) = \mathcal{C}(p_3) = \{q_1, q_2, q_3\}, \mathcal{C}(p_4) = \{q_1, q_2, q_3, q_4\}$ and $\mathcal{C}(G_p) = \{\mathcal{C}(p_1), \mathcal{C}(p_2), \mathcal{C}(p_3), \mathcal{C}(p_4)\}$, then the CountAllDistinct( ) function updates these candidate sets to ensure the all-different constraint, and returns the updated candidate sets as $\mathcal{C}(p_1) = \{q_1\}, \mathcal{C}(p_2) = \mathcal{C}(p_3) = \{q_2, q_3\}, \mathcal{C}(p_4) = \{q_4\}$.

Applying these algorithms over the exemplar pattern and data graphs identify three embeddings that are shown in table 5. In order to deal with induced subgraph isomorphism, we modified `SubGlw` in line to `Glasgow` solver, and its technical details can be seen in [11], [29].

Algorithm 8 CountAllDistinct($\mathcal{C}(G_p)$)

Input : $\mathcal{C}(G_p)$: Set of candidate sets
Output: Fail $\mathcal{F}$ or Success

1 $(\mathcal{H}, \mathcal{A}, \mathcal{F}, n) \leftarrow (\phi, \phi, \phi, 0)$
2 **foreach** $\mathcal{C}(p) \in$ *Ordered*($\mathcal{C}(G_p)$) **do**
3 $\quad \mathcal{C}(p) \leftarrow \mathcal{C}(p) \setminus \mathcal{H}$
4 $\quad (\mathcal{A}, \mathcal{F}, n) \leftarrow (\mathcal{A} \cup \mathcal{C}(p), \mathcal{F} \cup \{p\}, n + 1)$
5 $\quad$ **if** $\mathcal{C}(p) = \phi$ *or* $\#\mathcal{A} < n$ **then**
6 $\quad\quad$ **return** *Fail* $\mathcal{F}$
7 $\quad$ **end**
8 $\quad$ **if** $\#\mathcal{A} = n$ **then**
9 $\quad\quad (\mathcal{H}, \mathcal{A}, n) \leftarrow (\mathcal{H} \cup \mathcal{A}, \phi, 0)$
10 $\quad$ **end**
11 **end**
12 **return** *Success*

V. EXPERIMENTAL SETUP AND RESULTS

This section presents an experimental evaluation of our proposed `SubGlw` solver over various pattern and data graph pairs. All experiments are carried out on a desktop with an Intel Core i5-6600 processor and 4GB of RAM. `SubGlw` is implemented using the *igraph* library [30] in the C++ programming language and it uses some functions

TABLE 5. Exemplar pattern graph embeddings found in the data graph.

S.No.	Embedding
1	$\lhd 0, 0 \rhd, \lhd 1, 1 \rhd, \lhd 2, 2 \rhd, \lhd 3, 3 \rhd, \lhd 4, 4 \rhd, \lhd 5, 5 \rhd$
2	$\lhd 0, 6 \rhd, \lhd 1, 7 \rhd, \lhd 2, 8 \rhd, \lhd 3, 3 \rhd, \lhd 4, 9 \rhd, \lhd 5, 10 \rhd$
3	$\lhd 0, 6 \rhd, \lhd 1, 7 \rhd, \lhd 2, 8 \rhd, \lhd 3, 3 \rhd, \lhd 4, 9 \rhd, \lhd 5, 13 \rhd$

of `Glasgow`[1]. The *igraph* library has various functions to carry out basic graph operations. We have used the g++ compiler in the Ubuntu 18.04.2 LTS environment to compile the codes of `SubGlw`. The source codes of `SubGlw` can be downloaded from https://github.com/ZubairAliIgraph/SubGlw-master. This section also present an analysis for identifying saddle point, which is a timeout at which `SubGlw` finds maximum embeddings with least execution time. This analysis provides a better understanding of parameter settings.

A. PATTERN AND DATA GRAPHS

In order to assess `SubGlw`'s performance, we have used three benchmark data graphs namely `Yeast`, `Human`, and `Hprd` whose statistics are presented in table 6. We have constructed four pattern sets for each data graph containing 100 simple, undirected, labeled, and connected graphs of a specific size. We kept the size of pattern graph as 10, 15, 20 and 25. Table 7 provides the details of the constructed pattern sets. The pattern set Yn contains 100 pattern graphs of size n (where $n \in \{10, 15, 20, 25\}$) is listed in this table for the `Yeast` data graph. Accordingly, other notations from this table can be followed. To construct a pattern graph of said size, say n, we randomly took one vertex from the data graph and visited $n - 1$ vertices of the data graph in breadth-first-search order. The resulting pattern graph is the labeled subgraph induced by n visited data graph vertices. In order to create a pattern set of 100 such pattern graphs of a particular data graph, we have repeated these steps 100 times.

TABLE 6. Statistics of the data graphs.

Data graphs	#Vertices	#Edges	#Distinct labels	Avg. degrees
Yeast	3112	12519	71	8.10
Human	4675	86282	44	36.82
Hprd	9460	34998	307	7.80

B. EVALUATION RESULTS

This section presents the evaluation results of `SubGlw` using all pattern sets and data graphs discussed in the previous section. The performance of `SubGlw` is measured in terms of *embedding count* and *execution time*. The *embedding count* for a pattern set is the sum of each pattern graph's embedding counts found in the corresponding data graph. On the other hand, *execution time* of a pattern set is the sum of the execution times to find at most n embeddings for each pattern graph in the corresponding data graph. While executing `SubGlw`, we set 1000 as the embedding count limit for each data graph; 500 milliseconds as the timeout for the `Yeast` and `Human` data graphs, and 1000 milliseconds as the timeout for the `Hprd` data graph. Table 8 presents the evaluation results of `SubGlw` in terms of *embedding count* and *execution time* over all data graphs.

[1] https://github.com/ciaranm/glasgow-subgraph-solver

TABLE 7. Constructed pattern sets for `Yeast`, `Human`, and `Hprd` data graphs.

Data graph	Pattern set	Description
Yeast	Yn	pattern set Yn contains 100 pattern graphs of size n, where $n \in \{10, 15, 20, 25\}$
Human	Hn	pattern set Hn contains 100 pattern graphs of size n, where $n \in \{10, 15, 20, 25\}$
Hprd	Pn	pattern set Pn contains 100 pattern graphs of size n, where $n \in \{10, 15, 20, 25\}$

TABLE 8. Performance evaluation results of `SubGlw` over `Yeast`, `Human`, and `Hprd` data graphs.

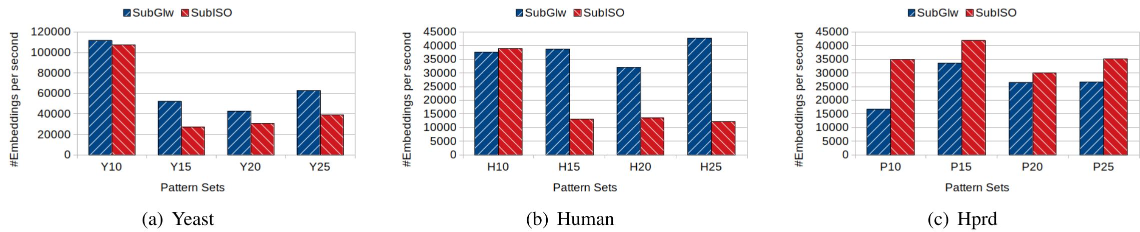
Data graph	P. set	Embedding limit	Timeout (ms)	#Embeddings	Exe. Time (ms)
Yeast	Y10	1000	500	56914	4689
	Y15	1000	500	66838	6566
	Y20	1000	500	67718	8586
	Y25	1000	500	69633	7769
Human	H10	1000	500	86908	10748
	H15	1000	500	90734	13260
	H20	1000	500	90954	17472
	H25	1000	500	95104	14494
Hprd	P10	1000	1000	5812	11839
	P15	1000	1000	9229	12386
	P20	1000	1000	11524	18093
	P25	1000	1000	19255	16655

C. COMPARATIVE ANALYSIS

In this section, we present a comparative analysis of `SubGlw` with two state-of-the-art subgraph isomorphism finding solvers – `SubISO` [7] and `Glasgow` [11] for both induced and non-induced subgraph isomorphism problems. It should be noted that `SubISO` finds only non-induced subgraph isomorphism, whereas `Glasgow` finds both induced and non-induced subgraph isomorphisms. Therefore, we have considered both `SubISO` and `Glasgow` for the comparative analysis of `SubGlw` for non-induced subgraph isomorphism, and only `Glasgow` for the comparative analysis of `SubGlw` for induced subgraph isomorphism. The experimental findings reveal that `SubGlw` performs significantly better in terms of both *embedding count* and *execution time*. Further details about the comparative analysis are presented in the following sub-sections.

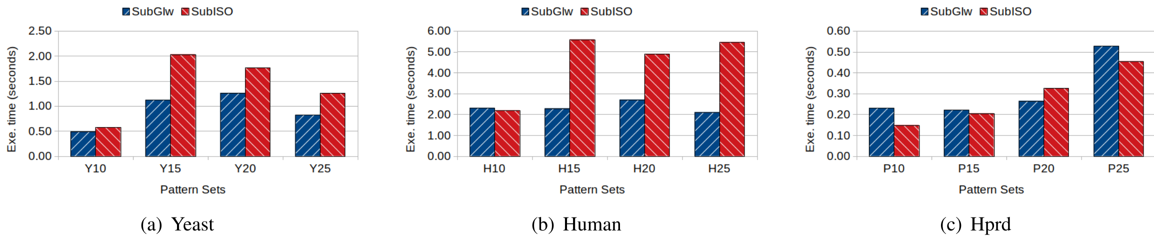
1) COMPARATIVE ANALYSIS OF `SubGlw` FOR NON-INDUCED SUBGRAPH ISOMORPHISM

In order to compare the performance of `SubGlw` for non-induced subgraph isomorphism problem, we have considered `SubISO` solver [7] with the default parameter settings, i.e., maximum recursive call as 1000 and embedding count limit as 1000 for each pattern graph. For `SubGlw`, we have used two parameters (i) *embedding count limit*, which is set to 1000, and (ii) *timeout*, which ranges



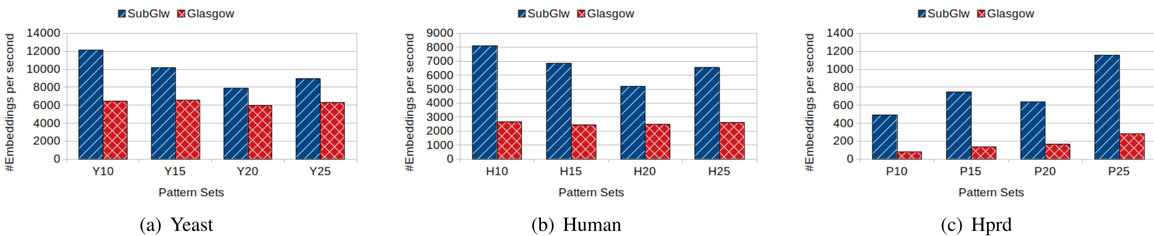
(a) Yeast (b) Human (c) Hprd

FIGURE 7. **Comparative analysis results of `SubGlw` and `SubISO` in terms of *number of embeddings per second* for non-induced subgraph isomorphism.**



(a) Yeast (b) Human (c) Hprd

FIGURE 8. **Comparative analysis results of `SubGlw` and `SubISO` in terms of *execution time* for non-induced subgraph isomorphism.**



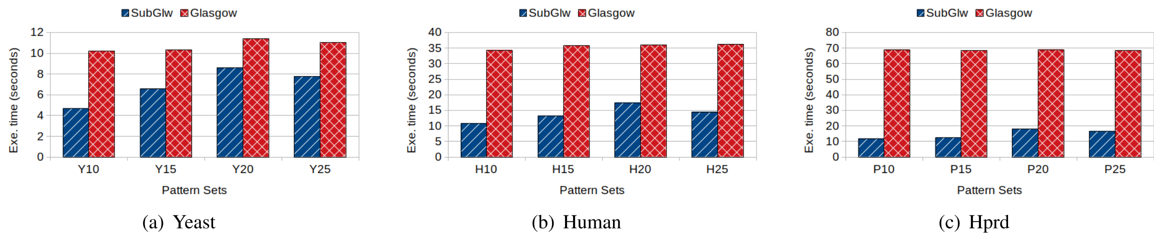
(a) Yeast (b) Human (c) Hprd

FIGURE 9. **Comparative analysis results of `SubGlw` and `Glasgow` in terms of *number of embeddings per second* for non-induced subgraph isomorphism.**

from 15 to 125 milliseconds. Figures 7 and 8 presents the comparative analysis results of `SubGlw` with `SubISO`. In these figures, pattern sets are along the x-axis, and *number of embeddings per unit time* and *average execution time per embedding* are along the y-axis. Figures 7(a) and 8(a) present the performance of `SubGlw` and `SubISO` over the `Yeast` data graph. It can be observed from these figures that except Y10 pattern set, `SubGlw`'s embedding count is greater than that of the `SubISO`, and the execution time of `SubGlw` is lesser than the execution time of `SubISO` for each pattern set. Similarly, figures 7(b) and 8(b) present the performance of `SubGlw` and `SubISO` over the `Human` data graph. It can be observed from these figures that `SubGlw`'s embedding count is greater than `SubISO`'s embedding count for all pattern sets, and the execution time of `SubGlw` is lesser than that of the `SubISO` for all pattern sets except H10. Finally, figures 7(c) and 8(c) present the performance of `SubGlw` and `SubISO` over the `Hprd` data graph. It can be observed from these figures that here `SubISO` performs better in comparison to `SubGlw` for all pattern sets except P20. `SubISO` has better performance on the `Hprd` data graph because it has too many distinct vertex labels, and the set of candidate matches for each vertex of the pattern graph is very small. However, `SubISO` uses vertex labels as well as the labels of the neighboring vertices to determine the set of candidate matches. On the other hand, `SubGlw` takes more time to identify fewer embeddings because it uses only vertex labels (not the labels of the neighboring vertices as SubISO does) to determine the set of candidate matches for a pattern graph's vertex.

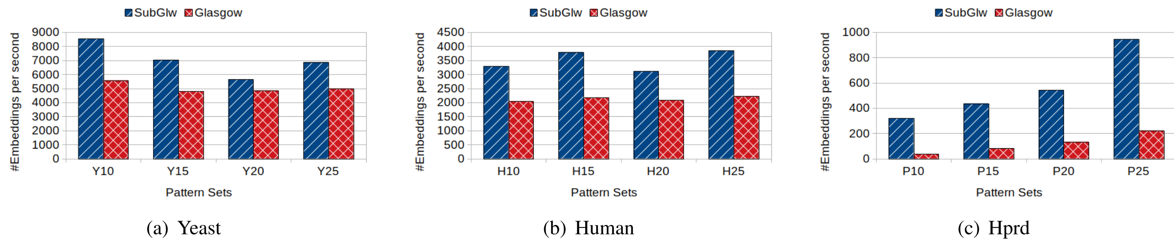
We have also compared the performance of `SubGlw` with `Glasgow` [11] for non-induced subgraph isomorphism. For both solvers, we used `embedding limit` value as 1000 and the `timeout` value ranging from 500 to 1000 milliseconds.

Figures 9(a) and 10(a) present the performance of `SubGlw` and `Glasgow` over the `Yeast` data graph. It can be observed from these figures that except the pattern set Y10, `SubGlw`'s embedding count is almost same as `Glasgow`'s embedding count, but the execution time of `SubGlw` is lesser than the execution time of `Glasgow` for all pattern sets. Similarly, figures 9(b) and 10(b) present the performance of both solvers over the `Human` data graph. It can be observed from these figures that except the pattern set H10 `SubGlw`'s embedding count is greater than `Glasgow`'s embedding count, and execution time of `SubGlw` is lesser than the execution time of `Glasgow` for all pattern sets. Finally, figures 9(c) and 10(c) present the performance of both solvers over the `Hprd` data



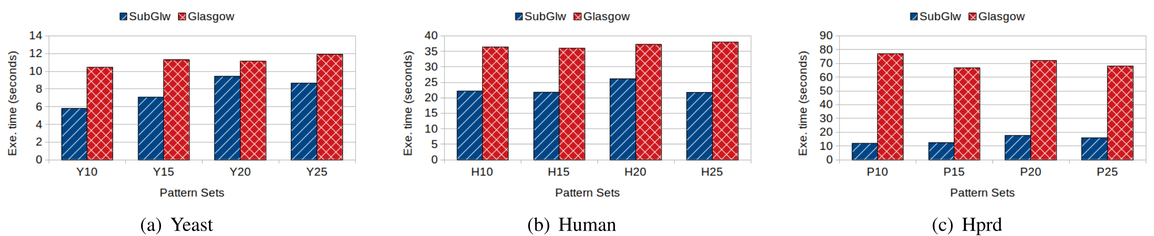
(a) Yeast (b) Human (c) Hprd

FIGURE 10. Comparative analysis results of `SubGlw` and `Glasgow` in terms of *execution time* for non-induced subgraph isomorphism.



(a) Yeast (b) Human (c) Hprd

FIGURE 11. Comparative analysis results of `SubGlw` and `Glasgow` in terms of *number of embeddings per second* for induced subgraph isomorphism.



(a) Yeast (b) Human (c) Hprd

FIGURE 12. Comparative analysis results of `SubGlw` and `Glasgow` in terms of *execution time* for induced subgraph isomorphism.

graph. It can be observed from these figures that `SubGlw`'s embedding count is almost same as `Glasgow`'s embedding count; however, the execution time of `SubGlw` is far lesser than the execution time of `Glasgow` for all pattern sets.

2) COMPARATIVE ANALYSIS OF `SubGlw` FOR INDUCED SUBGRAPH ISOMORPHISM

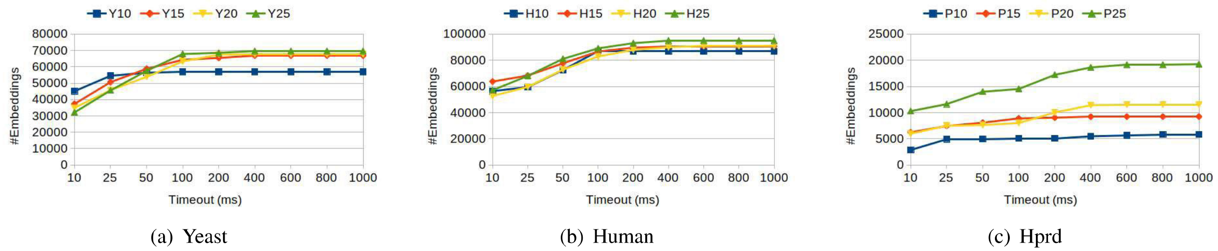
This section presents comparative analysis of `SubGlw` with `Glasgow` for the induced subgraph isomorphism. Since `SubISO` solver is not applicable for induced subgraph isomorphism, we have not considered it for the comparative analysis. The parameter settings of `SubGlw` and `Glasgow` are the same as explained in the previous section.

Figures 11(a) and 12(a) present the comparative analysis results of `SubGlw` and `Glasgow` over the `Yeast` data graph. It can be observed from these figures that `SubGlw`'s embedding count is lesser than `Glasgow`'s embedding count for the pattern sets Y10 and Y15; and embedding count of both solvers is almost same for the pattern sets Y20 and Y25. However, the execution time of `SubGlw` is lesser than `Glasgow`'s execution time for all pattern sets. Similarly, figures 11(b) and 12(b) present the comparative analysis results of both solvers over the `Human` data graph. It can be observed from these figures that `SubGlw`'s embedding count is lesser than `Glasgow`'s embedding count for the pattern sets H10 and H25, but it is larger for the pattern sets H15 and H20. However, execution time of `SubGlw` is lesser than `Glasgow`'s execution time for all pattern sets. Finally, figures 11(c) and 12(c) present the comparative analysis results of both solvers over the `Hprd` data graph. It can be observed from these figures that `SubGlw`'s embedding count is either same or greater than the embedding count of `Glasgow`. Moreover, the execution time of `SubGlw` is far lesser than the execution time of `Glasgow` for all pattern sets.

Based on the above experimental results, it can be concluded that `SubGlw` significantly performs better than `Glasgow` for induced subgraph isomorphism. The embedding count of `SubGlw` is comparable with `Glasgow`, but its execution time is much lesser than the execution time of `Glasgow`.

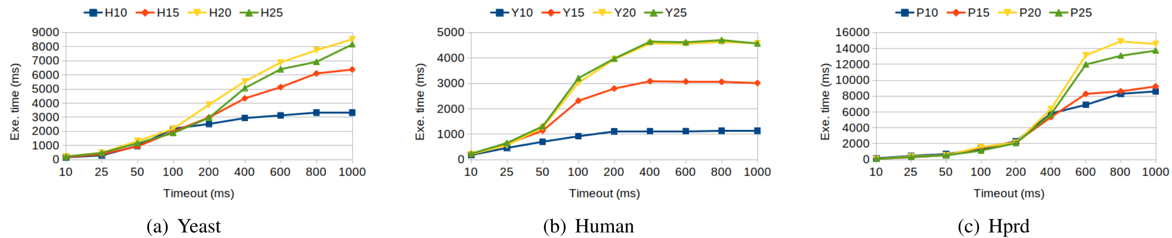
D. DISCUSSION ON THE SADDLE POINT

This section presents a discussion on the analysis of the *saddle point* for each pattern set which could be helpful for a better understanding of the parameter settings. In geometry, *saddle point* is achieved when maxima of one variable coincides with the minima of another variable. The concept of *saddle point* is also used in the field of game theory. For example, in a two-person-zero-sum game, *saddle point* occurs when both players optimize their worst-case payoff.



(a) Yeast (b) Human (c) Hprd

FIGURE 13. **Timeout vs. embedding count for different pattern sets.**



(a) Yeast (b) Human (c) Hprd

FIGURE 14. **Timeout vs. execution time for different pattern sets.**

In this game, player1 wants to maximize its goal, whereas player2 wishes to minimize its goal, and the equilibrium is achieved at the saddle point. In this study, the definition of the saddle point given in the following paragraph is not exactly same as it has been defined in game theory; instead, it is an analogous definition. For a specific pattern set, we have considered `SubGlw`'s embedding count and execution time as player1 and player2, respectively.

Definition 16: `Saddle point`: A saddle point is a timeout value at which a solver achieves maximum embeddings of a pattern graph in the data graph in least execution time.

Figures 13 and 14 present a visualization of the variation in the embedding count and execution time with increasing timeout values. In figures13(a) and 14(a), saddle points occur when timeout value is 100, 400, 200, and 400 milliseconds for the pattern sets Y10, Y15, Y20, and Y25, respectively. In figures 13(b), and 14(b), saddle points occur when timeout value is 100, 400, 600, and 200 milliseconds for the pattern sets H10, H15, H20, and H25, respectively. In figures 13(c), and 14(c), saddle point occurs when timeout value is 800, 800, 800, and 600 milliseconds for the pattern sets P10, P15, P20, and P25, respectively.

Since for any subgraph isomorphism solver both embedding count and execution time parameters matter, identification of saddle point for a pair of pattern and data graph is an impotent issue. In [24], the authors compared many state-of-the-art solvers using both embedding count and execution time parameters. Saddle point identification for subgraph isomorphism solvers is beneficial when we are concerned about both parameters while solving a subgraph isomorphism problem. We are always interested in maximizing the embeddings count and minimizing the execution time for any pair of pattern and data graphs. At the saddle point, we get the maximum value of embedding count with least execution time. Our above-mentioned analysis determines one saddle point for every pair of pattern and data graphs.

VI. CONCLUSION AND FUTURE WORK

In this paper, we have proposed the development of an efficient solver for subgraph isomorphism finding in large graphs. The proposed solver is based on `SubISO` and `Glasgow`, but it first decomposes a given data graph into small-size candidate subgraphs based on a ranking function, and then search the embeddings of the pattern graph in each of them in line to the `Glasgow` solver. The ranking function is designed to minimize both count and size of the candidate subgraphs. The proposed `SubGlw` solver is compared with two state-of-the-art subgraph isomorphism solvers for both induced and non-induced subgraph isomorphisms in terms of *embedding count* and `execution time`. The comparative analysis results over three different data graphs reveal that `SubGlw` performs significantly better in comparison to the state-of-the-art solvers in terms of both *embedding count* and *execution time*. We have also presented an analysis to determine the `saddle point` for each pattern set. The empirical analysis demonstrates that when parameter timeout meets the saddle point for a specific pattern set, `SubGlw` achieves optimal values of both *embedding count* and `execution time`. Potential future directions of research include (i) using the proposed lemma in other variants of the subgraph isomorphism problem to decompose data graph into small-size subgraphs, (ii) redesigning `SubGlw` for its parallel implementation using state-of-the-art tools, (iii) substituting supplemental graph by more light and effective isomorphic invariants or other suitable data structures.

REFERENCES

[1] C. McCreesh, P. Prosser, C. Solnon, and J. Trimble, "When subgraph isomorphism is really hard, and why this matters for graph databases," *J. Artif. Intell. Res.*, vol. 61, pp. 723–759, Mar. 2018.

[2] M. D. Preda and V. Vidali, "Abstract similarity analysis," *Electron. Notes Theor. Comput. Sci.*, vol. 331, pp. 87–99, Mar. 2017.

[3] A. Murray and B. Franke, "Compiling for automatically generated instruction set extensions," in *Proc. 10th Int. Symp. Code Gener. Optim. (CHO)*, San Jose, CA, USA, 2012, pp. 13–22.

[4] V. Bonnici, R. Giugno, A. Pulvirenti, D. Shasha, and A. Ferro, "A subgraph isomorphism algorithm and its application to biochemical data," *BMC Bioinf.*, vol. 14, no. 1, p. S13, 2013.

[5] T. Coffman, S. Greenblatt, and S. Marcus, "Graph-based technologies for intelligence analysis," *Commun. ACM*, vol. 47, no. 3, pp. 45–47, Mar. 2004.

[6] D. Conte, P. Foggia, C. Sansone, and M. Vento, "Thirty years of graph matching in pattern recognition," *Int. J. Pattern Recognit. Artif. Intell.*, vol. 18, no. 3, pp. 265–298, May 2004.

[7] M. Abulaish, Z. A. Ansari, and Jahiruddin, "SubISO: A scalable and novel approach for subgraph isomorphism search in large graph," in *Proc. 11th Int. Conf. Commun. Syst. Netw. (COMSNETS)*, Bengaluru, India, Jan. 2019, pp. 102–109.

[8] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, no. 1, pp. 31–42, Jan. 1976.

[9] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub)graph isomorphism algorithm for matching large graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 26, no. 10, pp. 1367–1372, Oct. 2004.

[10] C. Solnon, "Alldifferent-based filtering for subgraph isomorphism," *Artif. Intell.*, vol. 174, nos. 12–13, pp. 850–864, Aug. 2010.

[11] B. Archibald, F. Dunlop, R. Hoffmann, C. McCreesh, P. Prosser, and J. Trimble, "Sequential and parallel solution-biased search for subgraph algorithms," in *Proc. 16th Int. Conf. Integr. Constraint Program., Artif. Intell., Oper. Res.* Thessaloniki, Greece: Springer, 2019, pp. 20–38.

[12] I. Almasri, X. Gao, and N. Fedoroff, "Quick mining of isomorphic exact large patterns from large graphs," in *Proc. IEEE Int. Conf. Data Mining Workshop*, Shenzhen, China, Dec. 2014, pp. 517–524.

[13] V. Carletti, P. Foggia, A. Saggese, and M. Vento, "Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with VF3," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 40, no. 4, pp. 804–818, Apr. 2018.

[14] C. Nabti and H. Seba, "Querying massive graph data: A compress and search approach," *Future Gener. Comput. Syst.*, vol. 74, pp. 63–75, Sep. 2017.

[15] X. Ren and J. Wang, "Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs," *Proc. VLDB Endowment*, vol. 8, no. 5, pp. 617–628, Jan. 2015.

[16] S. Zhang, S. Li, and J. Yang, "GADDI: Distance index based subgraph matching in biological networks," in *Proc. 12th Int. Conf. Extending Database Technol. Adv. Database Technol. (EDBT)*. New York, NY, USA: ACM, 2009, pp. 192–203.

[17] H. He and A. K. Singh, "Graphs-at-a-time: Query language and access methods for graph databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*. New York, NY, USA: ACM, 2008, pp. 405–418.

[18] P. Zhao and J. Han, "On graph query optimization in large networks," *Proc. VLDB Endowment*, vol. 3, nos. 1–2, pp. 340–351, Sep. 2010.

[19] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *Proc. VLDB Endowment*, vol. 5, no. 9, pp. 788–799, May 2012.

[20] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, "Efficient subgraph matching by postponing Cartesian products," in *Proc. Int. Conf. Manage. Data*. New York, NY, USA: ACM, Jun. 2016, pp. 1199–1214.

[21] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, "Taming verification hardness: An efficient algorithm for testing subgraph isomorphism," *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 364–375, Aug. 2008.

[22] W.-S. Han, J. Lee, and J.-H. Lee, "Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases," in *Proc. Int. Conf. Manage. Data (SIGMOD)*. New York, NY, USA: ACM, 2013, pp. 337–348.

[23] G. Audemard, C. Lecoutre, M. Samy-Modeliar, G. Goncalves, and D. Porumbel, "Scoring-based neighborhood dominance for the subgraph isomorphism problem," in *Proc. 25th Int. Conf. Princ. Pract. Constraint Program.* Lyon, France: Springer, 2014, pp. 125–141.

[24] C. Solnon, "Experimental evaluation of subgraph isomorphism solvers," in *Proc. 12th Int. Workshop Graph-Based Represent. Pattern Recognit.* Tours, France: Springer, 2019, pp. 1–13.

[25] M. Qiao, H. Zhang, and H. Cheng, "Subgraph matching: On compression and computation," *Proc. VLDB Endowment*, vol. 11, no. 2, pp. 176–188, Oct. 2017.

[26] C. McCreesh and P. Prosser, "A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs," in *Proc. 21st Int. Conf. Princ. Pract. Constraint Program. (CP)*. Cork, Ireland: Springer, 2015, pp. 295–312.

[27] C. Lecoutre, L. Sais, S. Tabary, and V. Vidal, "Nogood recording from restarts," in *Proc. 20th Int. Joint Conf. Artif. Intell.* Hyderabad, India: Morgan Kaufmann, 2007, pp. 131–136.

[28] J. H. Lee, C. Schulte, and Z. Zhu, "Increasing nogoods in restart-based search," in *Proc. 30th AAAI Conf. Artif. Intell.* Phoenix, AZ, USA: AAAI Press, 2016, pp. 3426–3433.

[29] R. Hoffmann, C. McCreesh, and C. Reilly, "Between subgraph isomorphism and maximum common subgraph," in *Proc. 31st AAAI Conf. Intell. (AAAI)*. Palo Alto, CA, USA: AAAI Press, 2017, pp. 3907–3914.

[30] G. Csardi and T. Nepusz, "The igraph software package for complex network research," *Inter J. Complex Syst.*, vol. 1695, no. 5, pp. 1–9, 2006.

ZUBAIR ALI ANSARI received the M.Sc. degree in mathematics from Banaras Hindu University (BHU), India, in 2007, and the M.Tech. degree in computer applications from the Indian Institute of Technology (IIT) Delhi, in 2010. He is currently a Research Scholar with the Department of Computer Science, Jamia Millia Islamia (A Central University), New Delhi. In 2014, he has qualified two most prestigious Indian exams GATE and UGC-NET in computer science and engineering. His research interests include graph mining, data analysis, and graph theory.

JAHIRUDDIN received the Ph.D. degree in computer science from Jamia Millia Islamia (A Central University), New Delhi, India, in 2012. He is currently an Associate Professor with the Department of Computer Science, Jamia Millia Islamia. He has published over 19 research articles in various journals and conference proceedings. His research interests include text mining, machine learning, computational biology, and social networks analysis.

MUHAMMAD ABULAISH (Senior Member, IEEE) received the Ph.D. degree in computer science from the Indian Institute of Technology (IIT) Delhi, in 2007. He is currently a Full Professor of computer science with Jamia Millia Islamia (A Central University), New Delhi, India. He is also on deputation and working with South Asian University, New Delhi. He has published over 108 research articles in reputed journals and conference proceedings, including five papers in IEEE/ACM Transactions. His research interests include data analytics and mining, social computing, machine learning, and data-driven cyber security. He is also a TPC Member, and a Senior Member of ACM and CSI. He served for various reputed conferences, including CIKM, SDM, PAKDD, BIOKDD, IJCAI, WI, and ASONAM.