Detecting dynamic patterns in dynamic graphs using subgraph isomorphism

Kamaldeep Singh Oberoi, Géraldine Del Mondo, Benoit Gaüzère, Yohan Dupuis, Pascal Vasseur

Detecting Dynamic Patterns In Dynamic Graphs Using Subgraph Isomorphism

Kamaldeep Singh Oberoi[1,2], Géraldine Del Mondo[1]*, Benoît Gaüzère[1], Yohan Dupuis[3], Pascal Vasseur[4,5]

[1]Normandie Univ, INSA Rouen, LITIS, 76000 Rouen, France
[2]IRIT, University of Toulouse, CNRS, UT3, Toulouse, France
[3]LINEACT CESI, CESI, 92074 Paris La Défense, France
[4]Normandie Univ, UNIROUEN, LITIS, 76000 Rouen, France
[5]Laboratoire MIS - Université de Picardie Jules Verne - Amiens, France

Abstract

Graphs have been used in different fields of research for performing structural analysis of various systems. In order to compare the structure of two systems, the correspondence between their graphs has to be verified. The problem of graph matching, especially subgraph isomorphism (SI), has been well studied in case of static graphs. However, many applications require incorporating temporal information, making the corresponding graphs dynamic. In this paper, we apply SI to detect dynamic patterns in dynamic graphs. We propose an algorithm for induced SI to detect all the matchings for a given pattern graph while considering snapshot-based representation of dynamic graphs and taking into account the chronological order of these snapshots. This is the novelty of the proposed approach since the existing state-of-the-art algorithms model dynamic graphs using an aggregated model with time-stamped edges. To the best of our knowledge, there does not exist another approach which considers snapshot-based representation of dynamic pattern and dynamic target graphs for this problem. We discussed the time complexity of our algorithm and tested its performance while comparing it with two existing algorithms using the real-world datasets. It was found that our algorithm is the second best overall in terms of the execution time. The results are promising given the fact that the choice of dynamic graph model affects the algorithmic design for solving the problem of SI. For the applications where aggregated model of dynamic graphs is not applicable and snapshot-based representation is indispensable, our algorithm can be directly applied as opposed to the existing ones.

*Corresponding author

1 Introduction

Graphs have been in use in various fields of research such as bioinformatics, telecommunication, social networks etc. The main idea of using a graph is to model the structure of the underlying system in terms of its constituents (represented using nodes) and their interactions (represented using edges). To verify if two given systems, modeled using graphs, are similar or exactly the same, or if one system is a sub-system of another, their corresponding graphs have to be examined. The problem of determining the exact correspondence between graphs falls under the realm of *exact graph matching* [9], and Subgraph Isomorphism (SI) is one such problem. In SI, the objective is to find an injective mapping between the nodes of the two graphs while preserving the adjacency as well as the non-adjacency (induced SI) between the nodes. The graph to be matched is referred to as the *pattern graph* and the graph in which the matching has to be searched is called the *target graph*. SI finds applications in various fields. For example, in bioinformatics, target graphs represent molecules or protein interaction networks within which some user-defined patterns are detected [4]. In social network analysis, target graphs represent relationships between people (for example, drug traffickers in [15]) which are analysed to find interesting patterns. In video analysis for object tracking, target graphs represent the adjacency between different object planes present in the video and the pattern graph represents the object to be tracked in that video [13].

Most of the existing research related to SI either considers both pattern and target graphs to be static [9] or considers target graphs to be dynamic in which static patterns have to be detected [13, 33]. However, time can also be incorporated in the pattern graph in order to model dynamic patterns. Recently, some research has been published where dynamic patterns are detected in dynamic target graphs. For example, [28] proposes an algorithm to efficiently detect 2-node and 3-node, 3-edge dynamic patterns in dynamic graphs. The approach presented in [28] is further extended in [23] to detect larger patterns in larger target graphs. More recently, an algorithm, called RI [4], initially proposed to solve SI in static graphs, has been extended to dynamic graphs [22, 26].

In our opinion, the problem of SI in case of dynamic pattern and target graphs is still at the early stage of development and has not yet reached its full maturity. The existing approaches model a dynamic graph $\mathcal{G}$ as an "aggregated" graph with its edges annotated with the time instants and/or time intervals at which they are present. Such an aggregated model (with time-stamped edges) of dynamic graphs has been applied to represent events such as telephone calls between two individuals [1, 20], chronological events in a company [23], for detecting dynamic communities in dynamic graphs [30] etc. However, there exist applications where this modeling approach is not suitable and a snapshot-based model, where a dynamic graph $\mathcal{G}$ is represented using a set of static graphs (snapshots) G_i at time $t_i \in \mathcal{T}$, is preferred (here $\mathcal{T}$ represents the time domain of $\mathcal{G}$). In the literature, this approach has been applied for link prediction [14], for detecting small-world behaviour [36], for graph clustering [12] etc. in dynamic graphs. The main difference between the two approaches, in

terms of the definition of the model is that, in the former, the temporal order in which the edges appear in the graph is important, while in the latter, the temporal order of the entire snapshots, which represents the evolution of the graph over time, takes precedence.

In addition, concerning the formulation of the problem of SI, the representation of the dynamic graph, either using aggregated model with time-stamped edges or using graph snapshots, affects the way the algorithms are designed. For example, the algorithms proposed in [28], [23] and [26] first map the corresponding edges between pattern and target graphs, and then check for nodes, to find suitable matchings. However, this method is not suitable when the sequence of graph snapshots is considered. In this case, as in case of static graphs [9], the corresponding nodes between the pattern and target graphs have to be mapped before checking for the edges in order to find suitable matchings. This could be attributed to the algorithmic design choice since in case of aggregated representation, the temporal order of the edges presents a constraint which should be solved first before checking for the correspondence between the nodes, whereas in case of snapshot-based representation, the snapshot to be matched is a static graph, hence this constraint is absent. However, as mentioned before, the temporal order of the entire sequence of snapshots has to be considered.

In this paper, considering the snapshot-based representation of dynamic graphs, we propose an algorithm to find all the induced subgraphs of dynamic target graph isomorphic to a dynamic pattern graph. Our algorithm takes into account the chronological order of the snapshots of both graphs and maps the corresponding nodes (before checking the edges) to find the solution(s). To the best of our knowledge, there does not exist another algorithm which considers snapshot-based representation to solve the problem of SI in case of dynamic pattern and dynamic target graphs. The core of our algorithm is the algorithm VF3 [6] which was developed for SI in static graphs. Our choice of using VF3 is due to its performance outlined in [7]. Using the knowledge of previously matched nodes of the pattern graph, we propose the pruning strategy in order to reduce the size of the state-space of VF3. To showcase the performance of the proposed algorithm, we tested and compared it with two existing state-of-the-art algorithms using real-world datasets as target graphs and synthetically generated pattern graphs. It was found that the runtime of our algorithm was at par with the existing algorithms and in case of larger graphs, it was better than one of the existing algorithms. Hence, the proposed algorithm can easily be applied to applications where snapshot-based representation of dynamic graphs is required without having to compromise on the overall performance.

The paper is organised as follows. Section 2 describes the related work for the problem of SI. We discuss the existing approaches for the case of static and dynamic graphs and conclude that for dynamic graphs, the approach proposed in this paper is novel. In Section 3, we formalize the problem of subgraph isomorphism for dynamic graphs using the snapshot-based representation. Before delving into the details of the proposed algorithm in Section 5, we describe the main concepts behind VF3 in Section 4. Section 6 presents the experiments performed, the compar-

ison with existing algorithms and discusses the results obtained. Finally, Section 7 concludes the paper.

2 Related work

2.1 Subgraph isomorphism for static graphs

The problem of exact graph matching has been well studied by Pattern Recognition community in the last decades [9, 39] and graph isomorphism is its most restrictive form requiring the mapping between the nodes of the two graphs while preserving node adjacency as well as non-adjacency. Its variant, subgraph isomorphism (SI), which is the focus of this paper, is known to be NP-Complete in the general case [10].

Various techniques are present in the literature which solve SI by using some heuristics and pruning strategies to reduce the overall runtime. One of the famous algorithms, which is based on tree-based search and backtracking strategy was proposed by Ullmann [37], and was applied to both graph and subgraph isomorphism problems. In the same year, another contribution, also based on backtracking, was made for solving graph isomorphism problem for directed graphs [32]. Other noteworthy algorithms which use tree-based search strategy are VF2 [11], L2G [2], RI [4] and VF3 [6]. Both VF2 and VF3 use a set of feasibility rules to prune the search space, however, VF3 also considers the order on the nodes of the pattern graph to match most constrained nodes first. RI also takes the approach of ordering the nodes on the basis of their structural information. L2G focuses on the backtracking stage in the matching process and improves the pruning power and scalability.

SI could also be modeled as a constraint satisfaction problem (CSP) where the nodes of the pattern graph are assigned different domains i.e., the set of nodes of the target graph which could be mapped to a particular node of the pattern graph. The objective here is to reduce the size of this domain while satisfying the conditions required for isomorphism. This approach was introduced by McGregor [25] and was later enhanced in [34, 38].

Another perspective for applying the techniques of exact graph matching is proposed by Database research community which looks for a given pattern graph in a large database of target graphs. While querying a database of graphs, they propose indexing techniques which directly incorporate the structure and semantics of the query graph and help to reduce the time required to process the query. GraphGrep [17] is one such method designed for querying graph databases. It uses graph expression based query language and prunes the search space by using path-based indexing. He et al. [18] propose graph query language which uses generalized graph matching operator while querying graph database. It uses a pruning strategy based on the neighbourhood and the radius of the node to be included in the mapping.

2.2 Subgraph isomorphism for dynamic graphs

Although most of the research for SI has been focused on static graphs, recent years have seen an emerging interest for dealing with this problem in case of dynamic graphs. Kovanen et al. [19] introduce the framework of temporal motifs defined as the isomorphic temporal subgraphs of dynamic graphs having edges representing consecutive temporal events (aggregated model of dynamic graphs). Isomorphism for such (temporal) subgraphs takes into account their topology as well as the temporal order on their edges. To search for temporal motifs in target graphs, both these properties need to be satisfied, which can be done in different orders. Redmond et al. [29] have proposed three methods - Topology before Time, Time before Topology, and Time and Topology Together - and have compared their performance, reaching to the conclusion that the hybrid approach has the least computational complexity. They consider a temporal order on the edges of the temporal motif and look for the matchings where the edges follow the same order.

Another interesting approach to deal with temporal motifs and to prune the search space is to define a time window within which all the edges belonging to the matched subgraph must occur. Temporal motifs which define such restriction are termed as δ-temporal motifs [28]. Such motifs/patterns have also been searched by the approach described in [23]. Patterns considered in [23] have single edge at each time instant and the temporal order on the edges is given importance to find suitable matchings within a given time window (aggregated model of dynamic graphs).

Recently, the algorithm RI [4], originally developed for static graphs, has been extended to dynamic graphs and the new algorithm proposed is called TemporalRI [22, 26]. The first version of TemporalRI maps the corresponding nodes of pattern and target graphs to find suitable matchings [22], and the second version matches the corresponding edges first and then checks for the nodes [26]. However, none of the two versions of this algorithm allow multiple edges at a given time instant between two nodes and both consider the aggregated model of dynamic graphs.

The need to apply subgraph search in dynamic graphs also appears in case of streaming graphs, where a stream of graphs is queried in a continuous fashion, while the graph is being updated, to find isomorphic subgraphs [8, 35]. Instead of starting the subgraph search from scratch each time the graph stream gets updated, these algorithms perform graph matching in an online fashion while keeping track of partial matchings obtained in the previous time steps [16]. Schiller et al. [31] developed an algorithm to count 4-vertex motifs in streaming graphs. They define motif as a class of isomorphic subgraphs which are searched while continuously updating the given graph. However, they give the classes of such subgraphs as input to the algorithm and update their frequencies without actually solving the problem of SI. A similar approach is taken by [27] to count the frequency of motif instances in a given graph.

2.3 Concluding remarks about related work

It is clear that the problem of SI has been well studied for static graphs and there has been a growing interest for this problem in the realm of dynamic graphs. However, the formalisation of dynamic graphs changes the way this problem is approached and the existing work only considers the aggregated model of dynamic graphs with time-stamped edges. However, to understand and/or visualize the evolution of a dynamic graph over time, it is of interest to model dynamic graphs as a sequence of snapshots [3, 5]. For such use cases, the existing algorithms for subgraph isomorphism in dynamic graphs cannot be applied directly. The contribution of this paper is the algorithm which can be applied to find all the subgraphs of a dynamic target graph isomorphic to a dynamic pattern graph, where both pattern and target graphs are represented as a sequence of snapshots. Streaming graphs and related algorithms are out of scope of this paper. The related work is mentioned for the sake of completeness.

3 Problem formulation

We start by considering a simple static graph $G = (V, E)$ with a set of nodes V and set of edges $E \subseteq V \times V$. Given a pair of static pattern graph $H = (V_H, E_H)$ and static target graph $G = (V_G, E_G)$, the problem of SI is to find an injective mapping μ between the nodes of the two graphs. Mathematically, $\mu \subset V_H \times V_G$ represents a set of pairs of mapped nodes of pattern and target graphs. Following equations give the formal definition of the problem.

$$\forall u \in V_H, \exists\ v \in V_G \mid (u, v) \in \mu \tag{1}$$

$$\forall u, \forall u' \in V_H, u \neq u' \mid \{(u, v), (u', v')\} \in \mu \Leftrightarrow v \neq v' \tag{2}$$

$$\forall (u, u') \in E_H, \exists\ (v, v') \in E_G \mid \{(u, v), (u', v')\} \in \mu \tag{3}$$

$$\forall u, \forall u' \in V_H, \forall (v, v') \in E_G \mid \{(u, v), (u', v')\} \in \mu \Leftrightarrow (u, u') \in E_H \tag{4}$$

Equations 1 and 2 state that all distinct nodes of the H must be mapped to some (but distinct) nodes of G. Equation 3 states that all edges of H are mapped to some edges of G. Lastly, Equation 4 verifies the "induced" version of SI by checking that a mapped edge of G has a corresponding edge in H. Note that in Equations 2 and 4, the double-sided arrow (representing "if and only if") means that both left and right side should be true for the equation to be true. Additionally, if all conditions mentioned above are satisfied, we represent SI between graphs H and G as $\mu : H \hookrightarrow G$. Otherwise, we use the notation $H \not\hookrightarrow G$.

Next, we define a simple dynamic graph $\mathcal{G} = (V_\mathcal{G}, E_\mathcal{G}, \mathcal{T}_\mathcal{G})$ with set of nodes $V_\mathcal{G}$ and set of edges $E_\mathcal{G}$. Its time domain $\mathcal{T}_\mathcal{G}$ is considered to be discrete and can be represented as a set of time instants $\mathcal{T}_\mathcal{G} = \{t_1, t_2, ..., t_k\}$. We consider a snapshot-based model where a graph snapshot G_i at time instant t_i represents its state. Following this, a dynamic graph can be represented as a set of its discrete states, i.e., $\mathcal{G} = \{G_1, G_2, ..., G_k\}$. Given a pair of dynamic pattern graph $\mathcal{H} = (V_\mathcal{H}, E_\mathcal{H}, \mathcal{T}_\mathcal{H}) = \{H_1,\ H_2,\ ...,\ H_m\}$, having m snapshots, and dynamic target graph $\mathcal{G} = (V_\mathcal{G}, E_\mathcal{G}, \mathcal{T}_\mathcal{G}) =$

$\{G_1,\ G_2,\ ...,\ G_n\}$, having n snapshots, with $m \leq n$, the problem of induced SI can be extended to dynamic graphs, i.e.,

- Each snapshot of $\mathcal{H}$ must be mapped to some snapshot of $\mathcal{G}$ while respecting the equations of induced SI mentioned above. Formally,

$$\forall H_i \in \mathcal{H},\ 1 \leq i \leq m, \exists\ \mu_{ij} : H_i \hookrightarrow G_j,\ G_j \in \mathcal{G},\ 1 \leq j \leq n \quad (5)$$

- The temporal order of $\mathcal{H}$ and $\mathcal{G}$ must be respected, i.e., if a snapshot $H_i, 1 \leq i \leq m$ of $\mathcal{H}$ is mapped to a snapshot $G_j, 1 \leq j \leq n$ of $\mathcal{G}$, then the next snapshot H_{i+1} of $\mathcal{H}$ must be mapped to some snapshot G_f of $\mathcal{G}$ where $f > j$. Formally,

$$\forall H_i \in \mathcal{H},\ 1 \leq i \leq m,\ \exists\ \mu_{ij} : H_i \hookrightarrow G_j,\ G_j \in \mathcal{G},\ 1 \leq j \leq n \Leftrightarrow \\ \exists\ \mu_{(i+1)f} : H_{i+1} \hookrightarrow G_f,\ G_f \in \mathcal{G},\ j < f \leq n \quad (6)$$

- To find a complete matching of $\mathcal{H}$, a node $v \in V_{\mathcal{H}}$ must be mapped to the same node $v' \in V_{\mathcal{G}}$ at every time instant where the matching is possible and both nodes v and v' exist. Formally,

$$\forall H_i \in \mathcal{H},\ 1 \leq i \leq m,\ \exists\ \mu_{ij} : H_i \hookrightarrow G_j,\ G_j \in \mathcal{G},\ 1 \leq j \leq n \Leftrightarrow \\ \exists\ v \in V_{H_i}\ \wedge\ v' \in V_{G_j} \mid (v, v') \in \mu_{ij},\ \forall v \in V_{H_i} \wedge \forall v' \in V_{G_j} \quad (7)$$

- It is possible to have some snapshots of $\mathcal{G}$ to which no snapshots of $\mathcal{H}$ are matched. Such snapshots represent *noise*. We include such snapshots of target graph to be more realistic in terms of the sampling rate. For real-world applications, depending on this sampling rate, it is possible to have some snapshots in which no matching is found. Formally,

$$\exists\ G_j \in \mathcal{G},\ 1 \leq j \leq n \mid H_i \not\hookrightarrow G_j, H_i \in \mathcal{H},\ 1 \leq i \leq m \quad (8)$$

Figure 1 shows an example where the pattern graph has five snapshots and the target graph has seven snapshots. The pattern and target graph snapshots are matched as follows: $H_1 \rightarrow G_1$, $H_2 \rightarrow G_2$, $H_3 \rightarrow G_3$, $H_4 \rightarrow G_4$ and $H_5 \rightarrow G_7$, while nodes of the pattern graph are matched as: $A1 \rightarrow A100$, $A4 \rightarrow A35$, $A5 \rightarrow A15$ and $A3 \rightarrow A25$. Note that the temporal order of $\mathcal{H}$ is maintained in the solution found and a node of the pattern graph (e.g. $A1$) is matched to the same node of target graph ($A100$) at every time instant where the matching between $\mathcal{H}$ and $\mathcal{G}$ is possible and the nodes $A1 \in V_{\mathcal{H}}$ and $A100 \in V_{\mathcal{G}}$ exist. Some target graph snapshots (G_5 and G_6) represent "noise", i.e., they are not matched to any snapshot of the pattern graph.

4 Description of VF3 algorithm

VF3 falls in the category of algorithms which use a state-space representation, with backtracking, to keep track of the partial matchings. The state-space is represented as a tree whose nodes represent distinct states. A state is the candidate pair of nodes of pattern and target graphs. It

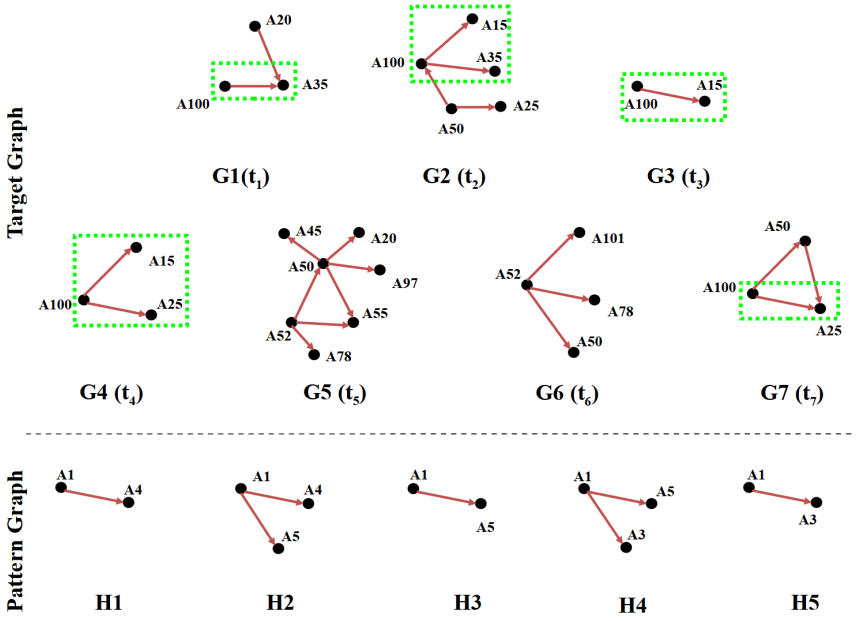

Figure 1: Example showing snapshots of target graph $\mathcal{G}$ and pattern graph $\mathcal{H}$ with detected pattern highlighted in $\mathcal{G}$. Snapshots $G5$ and $G6$ of $\mathcal{G}$ represent "noise" and are not mapped to any snapshot of $\mathcal{H}$. Set of nodes in both graphs remains constant over time but the isolated nodes are not shown

is said to be "consistent" if the partial mapping between the pattern and target graphs satisfies the conditions of SI. All consistent states, with complete mapping of the pattern graph, represent "goal" states. Starting from an empty initial state, VF3 progressively develops the state-space for all nodes of the pattern graph to find suitable matchings. If an "inconsistent" state is encountered, the algorithm backtracks to the previous state and continues the search from a different candidate pair. The algorithm VF3 can be broken down into three steps: (1) ordering the nodes of the pattern graph, (2) computing feasibility sets for pattern and target graphs, and (3) the matching process.

4.1 Ordering the nodes of pattern graph

To determine the order in which the nodes of the pattern graph should be explored, VF3 generates the node exploration sequence before starting the matching procedure. The calculation of this sequence is based on computing the probability that a node u of the pattern graph has a corresponding node v in the target graph. This probability function takes into account the structural and semantic information of both pattern and target graphs (the semantic information may represent the node labels). The generated ordering sequence is to used to formalise, in part, the states in the state-space, i.e., the nodes corresponding to the pattern graph are incorporated, according to this order, into the state-space before starting the matching procedure. The depth of the state-space depends on the number of nodes of the pattern graph.

4.2 Computing feasibility sets

VF3 uses some sets to find the appropriate candidate pair. These sets, called feasibility sets, are computed for both pattern and target graphs. Since the node exploration sequence for the pattern graph is calculated before starting the matching procedure, it is, in turn, used to calculate the feasibility sets once for the pattern graph. However, the feasibility sets for the target graph are updated during the matching process. Concretely, the feasibility sets are the sets of nodes of pattern and target graphs which are not included in the mapping μ but are connected to the nodes already present in μ. In the original paper, there are two kinds of feasibility sets, called predecessor feasibility sets and successor feasibility sets, since the authors have considered the graphs to be directed. However, they can easily be defined for undirected as well as labeled graphs.

4.3 Matching process

The matching process starts from an initial empty state. Using the node exploration sequence generated previously, the pattern graph node for each level of the state-space is fixed. Then, starting from the first node in the sequence, the algorithm looks for the appropriate target graph node. First, a set of potential candidates among all the unmatched nodes of the target graph is computed. For each candidate pair of nodes, before adding

it to the state-space, VF3 verifies if it is *feasible*, i.e., if it satisfies the conditions of SI and whether it *will* lead to further consistent states. For this, VF3 applies 1-lookahead and 2-lookahead rules which check if the current candidate pair, if added in the state-space, will lead (or not) to a consistent state in one step (or in two steps). This is done using the feasibility sets of the target graph. The objective here is to restrict the size of the state-space and improve the overall performance. If the current candidate pair satisfies all three conditions, it is added as a consistent state in the state-space, the feasibility sets of target graph are updated, and the algorithm moves on to the next possible pair of nodes. It is noteworthy that the state-space is extended following the depth-first process. If, however, the current candidate pair is not feasible, the algorithm backtracks to the previous consistent state and checks other candidate pair of nodes. In the end, the goal states, containing the mappings for all pattern graph nodes, represent the final solution. Since there could be multiple matchings, it is possible to have multiple goal states.

4.4 Time complexity analysis

It is important to discuss the time complexity of VF3 since it would help in understanding the time complexity of our algorithm presented in the next section.

The algorithm VF3 is inspired by another algorithm called VF2 [11] which also uses the tree-based search strategy with backtracking for solving the problem of SI in static graphs. The tree represents the state-space which has to be explored to find suitable matchings between the two graphs. In the original paper of VF2, the time complexity is calculated in terms of the size of the state-space to be explored, and in the worst-case is given as $\mathcal{O}(N \cdot N!)$ while the best-case is given as $\mathcal{O}(N^2)$, where N represents the number of nodes.

The motivation behind VF3 is to improve the state-space exploration by restricting the number of un-feasible states in the state-space. This is done using some heuristics and pre-computing some data structures before starting the matching process. In this paper, we discuss three steps of VF3 and in this section we will describe the time complexity for each step.

The first step of VF3, i.e., ordering the nodes of the pattern graph is a two-step process: (1) calculation of the probability value for each pattern graph node and (2) ordering the nodes according to this probability value. In the original article of VF3 [6], the complexity to calculate the probability is mentioned as $\mathcal{O}(\|V_H\|)$, where $\|V_H\|$ is the number of nodes of the pattern graph H. Theoretically, sorting these nodes can be done using an existing sorting algorithm such as quicksort with the worst-case time complexity of $\mathcal{O}(\|V_H\|^2)$. Hence, the overall worst-case time complexity for this step becomes $\mathcal{O}(\|V_H\|^2)$.

Next, pre-computing the feasibility set for the pattern graph depends on the degree of each node of H included in the mapping μ and the depth of the state-space to be explored. The degree of a node in the pattern graph is proportional to the total number of edges in the graph, i.e., $\mathcal{O}(\|E_H\|)$ and the depth of the state-space is $\|V_H\| - 1 \approx \mathcal{O}(\|V_H\|)$. Hence, the time complexity to pre-compute the feasibility set of the pattern graph is

$\mathcal{O}(\|V_H\| \cdot \|E_H\|)$. The feasibility set for the target graph G is calculated during the matching process.

Finally, using the candidate selection process and predefined node ordering of the pattern graph nodes, less un-feasible states are generated and the size of the state-space is reduced during the matching process. A loose upper-bound on the time complexity of the matching process of VF3 can be given as $\mathcal{O}(\|V_H\| \cdot \|V_G\|)$. Hence, the overall time complexity of VF3 can be given as $Comp(VF3) = \mathcal{O}(\|V_H\|^2) + \mathcal{O}(\|V_H\| \cdot \|E_H\|) + \mathcal{O}(\|V_H\| \cdot \|V_G\|)$.

Here, it is important to note that VF3 introduced heuristics, like ordering the nodes and pre-computing the feasibility set of the pattern graph, to perform better than VF2. However, there could be cases where these heuristics do not work.

For example, the nodes of the pattern graph are ordered on the basis of the probability that, for a given pattern node, a corresponding node exists in the target graph. As mentioned above, this probability is calculated using structural (and semantic, if required) information about both graphs. Now, it is possible to have the same probability value for each pattern node and in this case, the node ordering does not make much sense.

Furthermore, given the feasibility sets for both graphs, finding the right node mapping may require high complexity since the choice for node mapping made at the first step may have consequences on all other node mappings.

Hence, in the worst-case, where the heuristics introduced in VF3 do not work, the complexity of VF3 approaches the worst-case complexity of VF2. Although, as shown empirically in the original paper of VF3 [6], VF2 performs worse than VF3, thanks to these heuristics and in practical situations, we can do better with VF3 [7]. Finally, SI in general case rests NP-Complete [10].

In the next section, after presenting our algorithm, we will discuss its time complexity while considering that VF3 performs optimally.

5 Proposed algorithm

In this section we will present our algorithm. It takes as input the dynamic pattern graph $\mathcal{H}$ and the dynamic target graph $\mathcal{G}$, given as the sets of their static snapshots $\mathcal{H} = \{H_1, ..., H_m\}$ and $\mathcal{G} = \{G_1, ..., G_n\}$ respectively, with $m \leq n$, and returns all the matchings between these two graphs if they exist. Given the definition of SI for dynamic graphs (see Section 3), the proposed algorithm performs optimally.

We start the matching process from the first pattern graph snapshot and enumerate through all the snapshots (Algorithm 1). Hence, the description of the algorithm follows the same order.

5.1 Matching first pattern graph snapshot

To match the first pattern graph snapshot with one (or more) target graph snapshots (Algorithm 2), we apply VF3 recursively, taking as inputs $H_1 \in \mathcal{H}$ and $G_j \in \mathcal{G}, 1 \leq j \leq n - m + 1$. At this stage, the state-space for

Algorithm 1 Main($\mathcal{H}, \mathcal{G}$)

1: $Tree \leftarrow \emptyset$
2: $root \leftarrow null$
3: **for** i = 1 to m **do**
4: $Solution_i \leftarrow \emptyset$ ▷ stores all possible matchings for H_i
5: $\mathbb{G}_i \leftarrow \emptyset$ ▷ stores all induced subgraphs isomorphic to H_i
6: **if** i = 1 **then**
7: $Tree$ = MapFirstPatternGraph(i, n, m, $Solution_i$, $\mathbb{G}_i$, $Tree$)
8: **end if**
9: **if** i > 1 **then**
10: $Tree$ = MapNextPatternGraphs(i, n, m, $Solution_i$, $\mathbb{G}_i$,
11: $Solution_{(i-1)}$, $\mathbb{G}_{i-1}$, $Tree$)
12: **end if**
13: **end for**
14: $P \leftarrow \emptyset$ ▷ set of root-to-leaf paths
15: P = GetRootLeafPaths($Tree$)
16: **for all** $rlp \in P$ **do**
17: **if** len(rlp) = m + 1 **then**
18: $\mu_{\mathcal{HG}} = \{\mu \mid \forall \mu \in rlp\}$ ▷ $\mu_{\mathcal{HG}}$ is the complete solution
19: **end if**
20: **end for**

every call to VF3 starts from an empty initial state, and we get all possible matchings for the snapshot H_1 (which are stored in the set $Solution_{ij}$, line 5; Algorithm 2). The subgraph of G_j induced by the matching μ_{ij} is given as g_{ij}. Since we want to find the matchings for all the snapshots of the pattern graph in the given target graph sequence (according to the conditions described in Section 3), while matching H_1, we do not need to check the target graph snapshots for $j > n - m + 1$. Hence, the search space is limited to some extent.

5.2 Matching next pattern graph snapshots

Once we have all possible matchings for H_1, we continue to match next pattern graph snapshots $H_i, 1 < i \leq m$ (Algorithm 3).

For a snapshot H_i (current pattern snapshot to be matched), present at time instant $t_i \in \mathcal{T}_{\mathcal{H}}, 1 < i \leq m$, we compare its node set (V_{H_i}) with the node set ($V_{H_{i-1}}$) of the previous snapshot H_{i-1} which is already matched (lines 1-2; Algorithm 3). Since VF3 maps all the nodes of a given pattern graph, the nodes common between V_{H_i} and $V_{H_{i-1}}$ were already mapped while matching H_{i-1}. Our objective here is to use this knowledge to prune the state-space of VF3. Hence, if there exist nodes in H_i which did not exist in H_{i-1}, we call the function FindNewMapping(), else we call the function UsePreviousMapping() (line 3-7; Algorithm 3).

Let us suppose that the previous snapshot H_{i-1} gets matched to some snapshot G_j (with node set V_{G_j}) of the target graph (at time instant $t_j \in \mathcal{T}_{\mathcal{G}}$) (Algorithm 4). Corresponding matching between two graphs can be written as $\mu_{(i-1)j}$ which is a set of pair of nodes of snapshots H_{i-1} and

Algorithm 2 MapFirstPatternGraph(i, n, m, $Solution_i$, $\mathbb{G}_i$, $Tree$)

for j = 1 to n - m + 1 **do**
  $Solution_{ij} \leftarrow \emptyset$
  $\mathbb{G}_{ij} \leftarrow \emptyset$
  if VF3(H_i, G_j) = true **then**
    $Solution_{ij} \leftarrow \{\mu_{ij} \mid \mu_{ij} = V_{H_i} \times V_{g_{ij}},\ g_{ij} \subset G_j\}$
      ▷ $Solution_{ij}$ stores all matchings between H_i and G_j
    Add μ_{ij} as child node of $root$ in $Tree$
    Insert $Solution_{ij}$ in $Solution_i$
    $\mathbb{G}_{ij} \leftarrow \{g_{ij} \mid g_{ij} \subset G_j\}$
      ▷ $\mathbb{G}_{ij}$ is set of subgraphs of G_j isomorphic to H_i
      ▷ $len(Solution_{ij}) = len(\mathbb{G}_{ij})$
    Insert $\mathbb{G}_{ij}$ in $\mathbb{G}_i$
  end if
end for
  return $Tree$

Algorithm 3 MapNextPatternGraphs(i, n, m, $Solution_i$, $\mathbb{G}_i$, $Solution_{(i-1)}$, $\mathbb{G}_{i-1}$, $Tree$)

$NAM_i \leftarrow V_{H_i} \cap V_{H_{i-1}}$ ▷ set of nodes of H_i already mapped
$NNM_i \leftarrow V_{H_i} - NAM_i$ ▷ set of nodes of H_i not mapped
if $NNM_i \neq \emptyset$ **then**
  $Tree$ = FindNewMapping(i, n, m, NAM_i, NNM_i, $Solution_i$, $\mathbb{G}_i$,
    $Solution_{(i-1)}$, $\mathbb{G}_{i-1}$, $Tree$)
else
  $Tree$ = UsePreviousMapping(i, n, m, $Solution_i$, $\mathbb{G}_i$, $Solution_{(i-1)}$,
    $\mathbb{G}_{i-1}$, $Tree$)
end if
  return $Tree$

Algorithm 4 FindNewMapping(i, n, m, NAM_i, NNM_i, $Solution_i$, $\mathbb{G}_i$, $Solution_{(i-1)}$, $\mathbb{G}_{i-1}$, $Tree$)

1: $\mathbb{F}(H_i) \leftarrow \{u \mid u \in NNM_i, \exists\ u' \in NAM_i,\ (u,\ u') \in E_{H_i}\}$
▷ $\mathbb{F}(H_i)$ is the new feasibility set for H_i
2: **for** j = i - 1 to n - m + i - 1 **do**
3: $\mu_{(i-1)j} \leftarrow Solution_{(i-1)j}$ ▷ previous matchings
4: $\mu'_{(i-1)j} \leftarrow$ subset of $\mu_{(i-1)j}$ with nodes present in V_{H_i} and V_{G_j}
5: $g'_{(i-1)j} \leftarrow$ subgraph of G_j corresponding to $\mu'_{(i-1)j}$
6: **for** f = j + 1 to n - m + i **do**
7: **if** $g'_{(i-1)j} \subset G_f$ **then**
8: $\mathbb{F}(G_f) \leftarrow \{p \mid p \in V_{G_f}, \exists\ p' \in V_{g'_{(i-1)j}},\ (p,\ p') \in E_{G_f}\}$
▷ $\mathbb{F}(G_f)$ is the new feasibility set for G_f
9: **if** VF3(H_i, G_f, $\mu'_{(i-1)j}$, $\mathbb{F}(H_i)$, $\mathbb{F}(G_f)$) = true **then**
10: $Solution_{if} \leftarrow \{\mu_{if} \mid \mu_{if} = V_{H_i} \times V_{g_{if}},\ g_{if} \subset G_f\}$
11: Add μ_{if} as the child node of $\mu_{(i-1)j}$ in $Tree$
12: Insert $Solution_{if}$ in $Solution_i$
13: $\mathbb{G}_{if} \leftarrow \{g_{if} \mid g_{if} \subset G_f\}$
14: Insert $\mathbb{G}_{if}$ in $\mathbb{G}_i$
15: **end if**
16: **end if**
17: **end for**
18: **end for**
return $Tree$

G_j (line 3; Algorithm 4). Formally,

$$\mu_{(i-1)j} = \{(u, v) \mid u \in V_{H_{i-1}} \wedge v \in V_{G_j}\}$$

Note that since both snapshots are static graphs, their node sets are represented using the corresponding notations (Section 3).

The induced subgraph of the snapshot G_j, isomorphic to H_{i-1} and corresponding to the mapping $\mu_{(i-1)j}$, is written as $g_{(i-1)j}$. Since H_i is the current snapshot to be matched, we consider only those node pairs in $\mu_{(i-1)j}$ which have the nodes present in V_{H_i} as first part and their corresponding nodes in V_{G_j} as second part (line 4; Algorithm 4). The updated set of matchings is given as $\mu'_{(i-1)j} \subset \mu_{(i-1)j}$ and the updated corresponding subgraph of G_j is given as $g'_{(i-1)j}$ (line 5; Algorithm 4).

Since the pattern snapshot H_i occurs after (in the future of) the snapshot H_{i-1}, it must be matched to some snapshot G_f of the target graph such that $f > j$. For this, we need to look for the subgraph $g'_{(i-1)j}$ in every snapshot $G_f, f > j$ of the target graph. Note that, here, we do not need to check all the snapshots of the target graph with index $f > j$. We use the *sliding window* (Figure 2) to define the limits within which the snapshot H_i should be matched. Since we have to match all the snapshots of the pattern graph to find the complete solution, H_i must be matched to some snapshot G_f with $j < f \leq n-m+i$. Here, again, some limitation is applied to restrict the size of the search space. For example, from Figure 2, if the snapshot H_1 gets matched to the snapshot G_3, then the matching

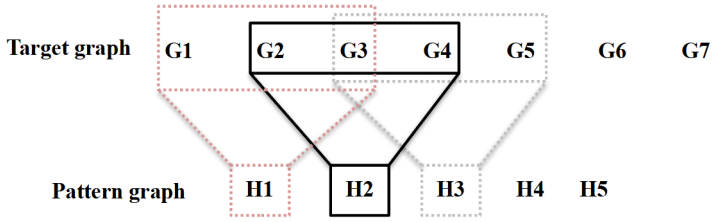

Figure 2: The location of the sliding window within which a match has to be searched for depends on the index of the pattern graph snapshot, and its size depends on the number of snapshots of both graphs

for the snapshot H_2 has to be searched in snapshot G_4 only.

5.2.1 Pruning the state-space of VF3

Once the snapshot G_f of the target graph containing the subgraph $g'_{(i-1)j}$ of G_j is selected, we need to apply VF3 again to map the un-mapped nodes of H_i (the nodes which were not present in H_{i-1}). Hence, taking H_i and G_f as inputs we apply VF3 again. However, here we use the knowledge of the previously mapped nodes to prune the state-space of VF3.

Instead of starting from an empty initial state in the state space, as is done in the static version of VF3 or while matching H_1, we start with non-empty initial state represented using the mapping $\mu'_{(i-1)j}$, and continue with VF3 until all the un-mapped nodes of H_i are mapped (line 9; Algorithm 4). To achieve this, we re-define the feasibility sets for both H_i and G_f (lines 1, 8; Algorithm 4). Recall that the feasibility set is the set of nodes of a graph which are neighbors of the nodes included in the mapping, and is defined for both pattern and target graphs to implement VF3 [6]. Finally, the matchings found between the snapshots H_i and G_f are stored in the set $Solution_{if}$ (line 10; Algorithm 4) and the corresponding subgraphs are stored in $\mathbb{G}_{if}$ (line 13; Algorithm 4).

5.2.2 Matching pattern snapshots without using VF3

It is also possible that the node set of H_i (the current pattern snapshot to be mapped) is the subset of the node set of H_{i-1} (previous pattern snapshot which is already mapped), i.e., all the nodes of H_i were also present in H_{i-1} (Algorithm 5). In this case, since all the nodes of H_{i-1} were mapped in the previous iteration, there is no need to apply VF3 again while matching the current snapshot H_i. We just need to look for the suitable target snapshot G_f which contains the subgraph $g'_{(i-1)j}$ of G_j in order to respect the temporal order in the final solution. This, in turn, increases the overall efficiency of the algorithm.

Algorithm 5 UsePreviousMapping(i, n, m, $Solution_i$, $\mathbb{G}_i$, $Solution_{(i-1)}$, $\mathbb{G}_{i-1}$, $Tree$)

for j = i - 1 to n - m + i - 1 **do**
  $\mu_{(i-1)j} \leftarrow Solution_{(i-1)j}$
  $\mu'_{(i-1)j} \leftarrow$ subset of $\mu_{(i-1)j}$ with node pairs present in V_{H_i} and V_{G_j}
  $g'_{(i-1)j} \leftarrow$ subgraph of G_j corresponding to $\mu'_{(i-1)j}$
  for f = j + 1 to n - m + i **do**
    if $g'_{(i-1)j} \subset G_f$ **then**
      $\mu_{if} \leftarrow \mu'_{(i-1)j}$ ▷ current matching is same as previous matching
      $Solution_{if} \leftarrow \{\mu_{if} \mid \mu_{if} = V_{H_i} \times V_{g_{if}},\ g_{if} = g'_{(i-1)j}\}$
      Add μ_{if} as the child node of $\mu_{(i-1)j}$ in $Tree$
      Insert $Solution_{if}$ in $Solution_i$
      $\mathbb{G}_{if} \leftarrow \{g_{if} \mid g_{if} \subset G_f\}$
      Insert $\mathbb{G}_{if}$ in $\mathbb{G}_i$
    end if
  end for
end for
  return $Tree$

5.3 Finding the complete solution

To keep track of the possible matchings for all the pattern graph snapshots, we maintain a $Tree$ whose nodes represent the individual matchings (except empty *root* node). $Tree$ corresponding to the example pattern and target graph pair in Figure 1 is shown in Figure 3. Matchings computed by looking for the induced subgraph $g'_{(i-1)j}$ in the snapshot $G_f, j < f \leq n-m+i$ of the target graph are stored as the children of the node representing the previous matching $\mu_{(i-1)j}$, and all the matchings corresponding to the snapshot H_i of the pattern graph are stored at depth i. Hence the maximum depth of $Tree$ is equal to the number of snapshots of the pattern graph.

It is evident that there could be multiple matchings between a pair of pattern and target graph snapshots at the same depth (e.g. μ^1_{11} and μ^2_{11} are two separate matchings between H_1 and G_1 at depth 1). The set of matchings lying on every root-to-leaf path of length $m+1$ (considering path length in terms of the number of nodes present on the path), highlighted in blue in Figure 3, gives the final solution and the size of this tree depends on the number of all the matchings found for each pattern graph snapshot.

5.4 Time complexity analysis

In this section, we will discuss the time complexity of our algorithm which, as mentioned before, uses VF3 at its core.

In order to find the matching of the first snapshot $H_1 \in \mathcal{H}$ of the dynamic pattern graph (Algorithm 2), we apply VF3 recursively between H_1 and $G_j \in \mathcal{G},\ j \in w = [1,\ n-m+1]$, where w is the size of the sliding window within which the solution has to be found. Given the complexity

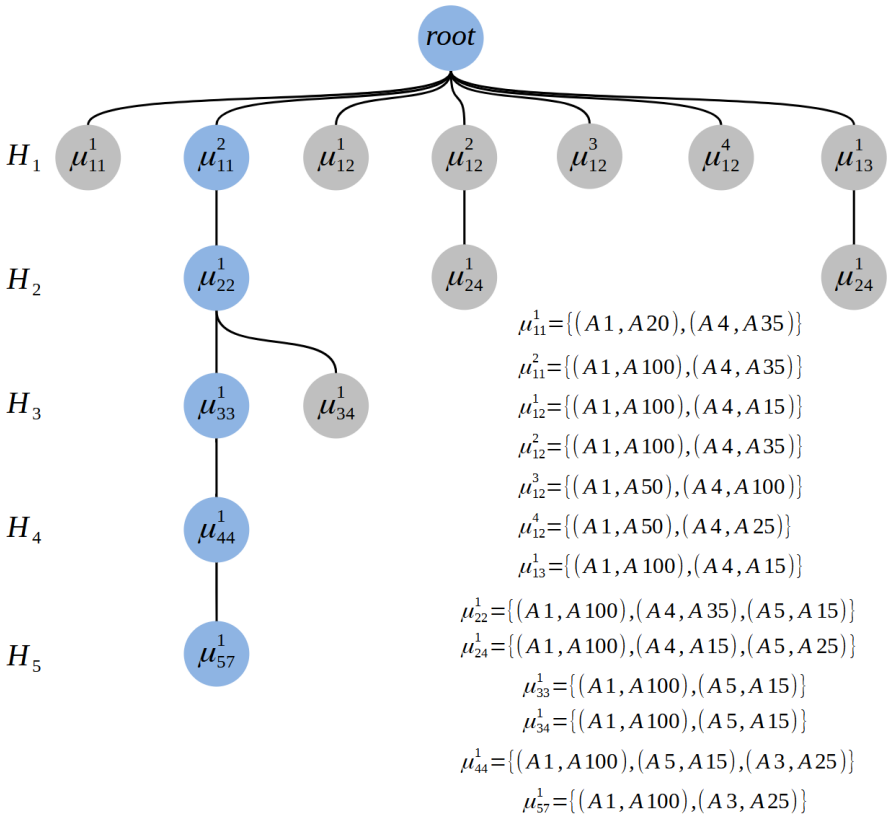

Figure 3: *Tree* maintained throughout the execution of the algorithm. Nodes in blue represent the root-to-leaf path which gives the complete matching between $\mathcal{G}$ and $\mathcal{H}$ in Figure 1

of VF3 as $Comp(VF3)$ (see Section 4.4), the time complexity of Algorithm 2 can be given as $\mathcal{O}(w \cdot Comp(VF3))$.

In Algorithm 3 (lines 1-2), the complexity of calculating the set NAM_i of nodes of snapshot H_i, $i > 1$ already mapped while mapping the previous snapshot H_{i-1} and the set NNM_i of nodes not mapped depends on the number of nodes of snapshots H_i and H_{i-1} and, in the worst-case, can be given as $\mathcal{O}(max(\|V_{H_i}\|, \|V_{H_{i-1}}\|))$.

The objective of Algorithm 4 is to map the nodes of the pattern snapshot H_i which are not previously mapped. Here, the worst case is when all but one nodes of H_i are not mapped, so $\|NNM_i\| = \|V_{H_i}\| - 1 \approx \mathcal{O}(\|V_{H_i}\|)$. Hence, the worst-case time complexity to calculate the new feasibility set of snapshot H_i (line 1) is $\mathcal{O}(\|V_{H_i}\| \cdot \|E_{H_i}\|)$.

Then, for each target snapshot G_j within the sliding window of size $w = [i-1,\ n-m+i-1]$, we have to iterate through all the previous matchings found (lines 3-5). McCreesh et al. [24] give the expected

number of matchings for induced subgraph isomorphism between a static pattern snapshot H_{i-1} and target snapshot G_j as

$$\kappa = \frac{\|V_{G_j}\|! \cdot p(G_j)^{p(H_{i-1}) \cdot \binom{\|V_{H_{i-1}}\|}{2}} \cdot (1 - p(G_j))^{(1-p(H_{i-1})) \cdot \binom{\|V_{H_{i-1}}\|}{2}}}{\|V_{H_{i-1}}\|!} \tag{9}$$

where $p(G_j)$ (and $p(H_{i-1})$) is the probability of having an edge between a pair of nodes in target snapshot G_j (and pattern snapshot H_{i-1}). Since this is induced subgraph isomorphism, the probability of not having an edge (i.e. non-adjacency) between two nodes is also considered to calculate κ. It is noteworthy that the probability of having an edge is related to the density of the graph, and that κ gives an upper-bound on the size of the set $\mu_{(i-1)j}$ (line 3).

For each solution found, the next step is to look for a subgraph $g'_{(i-1)j}$, associated to it, in the snapshot G_f (line 7). The time complexity for this step is proportional to $\|V_{G_f}\|$ and $\|E_{G_f}\|$. Similar to line 1, the feasibility set of G_f is calculated in $\mathcal{O}(\|V_{G_f}\| \cdot \|E_{G_f}\|)$ (line 8) in the worst-case.

Finally, in line 9, we apply VF3 starting from a non-empty initial state to restrict the size of the state-space. In the worst-case, where only one node of H_i is previously mapped (while matching H_{i-1}), we have to map all the remaining nodes. Hence the time-complexity of VF3 in this case would be $Comp(VF3) = \mathcal{O}(\|V_{H_i}\|^2) + \mathcal{O}(\|V_{H_i}\| \cdot \|V_{G_f}\|)$. However, in the best case, when only one node of H_i has to be mapped (with all other nodes mapped previously), the time complexity of VF3 becomes $Comp(VF3) = \mathcal{O}(\|V_{G_f}\|)$.

Hence, the overall time complexity of Algorithm 4 is $\mathcal{O}(\|V_{H_i}\| \cdot \|E_{H_i}\|) + w \cdot \kappa \cdot [\mathcal{O}(\|V_{G_f}\| \cdot \|E_{G_f}\|) + Comp(VF3)]$.

Similar to Algorithm 4, the time complexity for Algorithm 5 can be given as $w \cdot \kappa \cdot \mathcal{O}(\|V_{G_f}\| \cdot \|E_{G_f}\|)$ since in this case there is no need to apply VF3 and the feasibility sets are not calculated. However, the subgraphs associated to previous solutions have to be searched.

To find the complete solution using $Tree$ that is maintained throughout the execution, we find all root-to-leaf paths in $Tree$ (lines 14-19, Algorithm 1) with the complexity of $\mathcal{O}(M)$, where M is the number of nodes (individual matchings) in $Tree$.

In the next section, we perform the experiments to calculate the average runtime of our algorithm with different graph datasets and graph parameters.

6 Experiments

In this section, we will outline the experiments conducted to showcase the performance of our algorithm in terms of its runtime. We compare our algorithm with two existing ones, Mackey's algorithm proposed in [23], and the second version of TemporalRI proposed in [26]. Our choice is motivated due to the following reasons:

1. Both these algorithms are developed for solving the problem of induced SI in case of dynamic pattern and dynamic target graphs, al-

beit they represent a dynamic graph using an aggregated model with time-stamped edges as compared to the snapshot-based approach of our algorithm.

2. The general approach of TemporalRI algorithm is to extend an existing algorithm (RI [4]) to the case of dynamic graphs. We have taken a similar approach.
3. To the best of our knowledge, there does not exist another algorithm to solve induced SI which considers snapshot-based model of dynamic graphs.
4. In the original papers of TemporalRI [22, 26], the authors have compared it with Mackey's algorithm as well. Hence, our choice of comparing our algorithm with these two algorithms is meaningful.
5. The code for both these algorithms is provided by the respective authors.

For the comparison, we used two existing graph datasets - Email-Eu-core-temporal and CollegeMsg - available from SNAP Graph Library [21] as target graphs. Email-Eu-core-temporal was generated using email communication between employees of a European research institution. CollegeMsg represents communication between users on an online social network. Both datasets represent directed graphs. However, the proposed algorithm is applicable to undirected graphs as well. For the experiments, the pattern graphs were generated randomly having varying number of nodes and snapshots since our objective was to test our algorithm for different values of pattern graph parameters. However, note that since patterns are random, it cannot be stated that there exists dynamic subgraph isomorphism between a pattern and target graph pair, and it is possible to find zero graph mappings (solutions) in this case.

We measured the total algorithm runtime, including pre-processing and matching time, excluding, however, the time required to load the graph files. All the experiments were performed on a PC with Intel Core-i7 3.6GHz CPU and 32GB RAM running Ubuntu 18.04 LTS. Our algorithm as well as Mackey's algorithm were implemented in C++, hence they were compiled with GNU C++ compiler (version 7.5.0) with level-3 optimisation whereas TemporalRI algorithm was implemented in Java, hence, was compiled and executed using OpenJDK version 17.0.1.

It needs to be pointed out that the implementation provided by the authors for Mackey's algorithm [23] did not suit our needs since it does not work correctly with the graphs having multiple edges at the same time. After discussing with the authors, we found out that if multiple edges have the same value for time attribute (which means, theoretically, they occur at the same time), the sequential order of occurrence of the edges takes precedence. For example, if two edges $e1$ and $e2$ have the time attribute t but if $e1$ is read before $e2$ in the list of edges given as input, the algorithm considers $e1$ to occur before $e2$. This is suitable for the use cases where edges appear in the chronological order, however, it leads to unwanted matchings in our case. This problem has also been highlighted by the authors of TemporalRI [26], hence they had to slightly modify the implementation to make it work. Since the authors of TemporalRI

Table 1: Results obtained using Email-Eu-core-temporal dataset. Average runtime (in secs), standard deviati
percentage of graph instances solved for each pattern graph parameter are presented for three algorithms. Para
correspond to zero number of matchings found between graph pairs

Pattern Graph		Mackey's algorithm			TemporalRI			Our alg	
Nodes	Snapshots	Time	Std. Dev.	%	Time	Std. Dev.	%	Time	St
2	5	3.150	0.030	100	0.789	0.019	100	**NULL**	
2	10	701.644	1.437	100	162.562	2.545	100	**NULL**	
2	30	**NULL**	**NULL**	**0**	**NULL**	**NULL**	**0**	318.729	2
3	5	0.942	0.561	100	4.307	2.157	100	38.705	1
3	10	5.933	4.169	100	194.608	142.209	96	23.122	
3	30	10.884	7.282	100	**NULL**	**NULL**	**0**	14.430	
5	5	4.834	6.548	100	5.042	4.675	100	0.023	
5	10	11.806	25.156	100	178.152	77.800	100	0.021	
5	30	3.016	5.194	100	767.85	77.449	8	0.011	
10	5	171.800	203.744	36	1.533	0.523	100	0.022	
10	10	156.700	180.427	40	53.115	41.828	100	0.020	
10	30	182.212	143.398	40	0.245	0.036	100	0.010	
50	5	**NULL**	**NULL**	**0**	0.200	0.012	100	0.024	
50	10	**NULL**	**NULL**	**0**	0.355	0.015	100	0.021	
50	30	**NULL**	**NULL**	**0**	1.952	0.027	100	0.011	

also provide the new implementation of Mackey's algorithm, we used this version of the implementation as opposed to the original version.

6.1 Dataset preprocessing

As mentioned above, we used two datasets. Originally, Email-Eu-core-temporal dataset is collected for the period of 803 days and CollegeMsg dataset is collected for the period of 193 days. Both these datasets are available as time-stamped edge lists with time granularity of one second. As a pre-processing step, we changed the time granularity of both the datasets to one day, thus having several edges at every time instant and every graph snapshot. Furthermore, duplicate and self-loop edges were removed. Finally, we only considered the data for a smaller duration. For Email-Eu-core-temporal dataset, we considered the data for 50 days having 738 total nodes and 20331 total edges, and for CollegeMsg dataset we considered the data for 100 days having 1765 total nodes and 29407 total edges. The reason behind this choice is explained in the following section.

6.2 Results with random pattern graphs

We generated random pattern graphs having different number of nodes and number of snapshots. For each set of parameters, 25 graphs were

Table 2: Results obtained using CollegeMsg dataset. Average runtime (in secs), standard deviation (in secs) ar
graph instances solved for each pattern graph parameter are presented for three algorithms. Parameters in ora
to zero number of matchings found between graph pairs

Pattern Graph		Mackey's algorithm			TemporalRI			Our alg	
Nodes	Snapshots	Time	Std. Dev.	%	Time	Std. Dev.	%	Time	St
2	5	0.255	0.001	100	0.210	0.011	100	111.391	
2	10	3.795	0.014	100	1.969	0.110	100	170.280	
2	30	11.153	0.018	100	1.771	0.022	100	128.407	
2	100	11.127	0.012	100	0.079	0.009	100	0.001	
3	5	0.288	0.060	100	0.601	0.180	100	106.194	1
3	10	0.290	0.067	100	1.791	1.163	100	93.871	1
3	30	0.291	0.080	100	1.480	0.460	100	61.631	
3	100	0.321	0.089	100	0.151	0.010	100	0.001	
5	5	23.893	41.811	100	0.525	0.113	100	0.109	
5	10	26.239	56.002	100	1.465	0.336	100	0.106	
5	30	9.606	16.745	100	0.154	0.011	100	0.085	
5	100	31.611	65.074	100	0.185	0.013	100	0.001	
10	5	166.803	197.719	20	0.389	0.103	100	0.110	
10	10	235.464	135.553	28	0.966	0.229	100	0.105	
10	30	316.184	249.576	24	0.163	0.012	100	0.083	
10	100	332.44	72.745	12	0.271	0.035	100	0.001	
50	5	**NULL**	**NULL**	**0**	0.237	0.012	100	0.113	
50	10	**NULL**	**NULL**	**0**	0.389	0.018	100	0.107	
50	30	**NULL**	**NULL**	**0**	1.976	0.027	100	0.084	
50	100	**NULL**	**NULL**	**0**	23.161	0.927	100	0.002	

generated (875 graphs in total) and we calculated average runtime (in seconds) and standard deviation using the two datasets as target graphs and random pattern graphs. The results of the experiments obtained with Email-Eu-core-temporal dataset are presented in Table 1 and for CollegeMsg dataset in Table 2. In both tables, we note the performance of all three algorithms as well as the percentage of graph instances solved (out of 25) within a pre-defined timeout of 900 seconds (15 minutes). If none of the instances were solved before timeout, the average runtime and standard deviation could not be calculated and NULL values were noted. For all the experiments, the number of snapshots of pattern graph were always less than or equal to the number of snapshots of the target graph (according to the requirements mentioned in Section 3).

Since we obtained multiple NULL values for all three algorithms (see Table 1 and 2) using the two considered "smaller" datasets, we do not report the results for any of the algorithms with larger datasets. As a test, we applied TemporalRI using Email-Eu-core-temporal dataset having the duration of 500 days (978 total nodes, 207025 total edges) and pattern graphs having 2 nodes, but we got NULL values for all 25 graph instances checked.

The results of the experiments can be summarized as follows:

- In general, for smaller pattern graphs, Mackey's algorithm and TemporalRI have similar performance and both are better than our algorithm.
- As the size and the number of snapshots of the pattern graph grow, the performance of Mackey's algorithm degrades since the algorithm suffers from memory issues (also noted by the authors of TemporalRI [22]). For the largest pattern graph with varying number of snapshots, Mackey's algorithm was never able to finish before timeout for both datasets.
- The performance of TemporalRI for large (and long) target graph is better than that of our algorithm. However, for small (and short) target graph, our algorithm performs better than TemporalRI. We believe this is due to the fact the our graph model is based on (multiple) snapshots whereas for TemporalRI, there is single pattern/target graph with timestamped edges. With more number of snapshots of the target graph, the execution time of our algorithm increases.
- In both Tables 1 and 2, for the pattern graph parameters shown in orange, all three algorithms returned zero mappings, i.e., no dynamic subgraph isomorphism was found between a pattern and target graph pair. In such cases, in general, the performance of TemporalRI and our algorithm is similar, i.e., both algorithms stop early when there does not exist any solution. Mackey's algorithm, on the other hand, does not properly use the heuristics to detect such cases.
- Looking at the number of graph instances solved before timeout by all three algorithms (the percentage column in Tables 1 and 2), Mackey's algorithm solves the least. TemporalRI and our algorithm are similar in this regard as they solve equivalent number of graph instances before timeout.

- In general, out of the three algorithms tested, for larger target and pattern graphs, TemporalRI performs the best and Mackey's algorithm performs the least best. Our algorithm is in the second place in terms of execution time. For smaller pattern graphs, both Mackey's algorithm and TemporalRI are better than ours.

Hence, looking at the results obtained, it can be said that for solving the problem of dynamic subgraph isomorphism, the choice of model of dynamic graphs plays a critical role. It is better to use an aggregated model of dynamic graphs with time-stamped edges for this problem, as is the case in Mackey's algorithm (ignoring the memory related issues) and in TemporalRI. However, in some applications, it is not possible to model the dynamic graph in such a way and snapshot-based model is indispensable. In such cases, for solving the problem of SI, the existing algorithms can not be applied. Our algorithm, on the other hand, inherently takes into account the snapshot-based modeling approach of dynamic graphs and can easily be applied in such cases.

Table 3: Results obtained by increasing the size of the sliding window for Email-Eu-core-temporal dataset. The initial size of the sliding window is w. Average runtime (in secs) is reported for each graph parameter. Parameters in orange correspond to zero number of matchings found between graph pairs

Pattern Graph						
Nodes	Snapshots	w	$w+1$	$w+2$	$w+3$	$w+4$
2	5	NULL	NULL	NULL	NULL	NULL
3	5	**75.806**	**78.987**	**83.048**	**84.744**	**86.143**
5	5	0.023	0.023	0.023	0.024	0.024
10	5	0.022	0.022	0.022	0.022	0.023
50	5	0.024	0.024	0.024	0.025	0.025

6.3 Effect of the size of the sliding window

In this section, we will discuss the effect of the size of the sliding window used in the proposed algorithm in order to find suitable matchings. The idea of using the sliding window is to restrict the number of snapshots of the target graph in which the matching has to be searched, for a given pattern graph snapshot. As mentioned in Section 5.2, the size of this window depends on the number of snapshots of the target and the pattern graph.

The experiments discussed in the previous section were performed using the sliding window whose size (say w) was calculated based on the number of snapshots of the input graphs. The experiments discussed in this section were performed with different sizes of the sliding window. The initial size (w) was incremented step-by-step for each experiment and the effect on the average runtime of the algorithm was noted. For these experiments, we used 10 graphs for each pattern graph parameter with both Email-Eu-core-temporal and CollegeMsg datasets and the timeout

Table 4: Results obtained by increasing the size of the sliding window for CollegeMsg dataset. The initial size of the sliding window is w. Average runtime (in secs) is reported for each graph parameter. Parameters in orange correspond to zero number of matchings found between graph pairs

Pattern Graph						
Nodes	Snapshots	w	$w+1$	$w+2$	$w+3$	$w+4$
2	5	**97.915**	**99.194**	**99.963**	**100.502**	**100.472**
3	5	**162.201**	**164.631**	**166.660**	**169.547**	**169.006**
5	5	0.109	0.106	0.107	0.109	0.109
10	5	0.107	0.105	0.106	0.106	0.108
50	5	0.111	0.109	0.110	0.111	0.112

of 900 seconds (15 minutes). The results are given in Table 3 and Table 4 respectively.

As can be seen from both tables, increasing the size of the sliding window increases the average runtime of the algorithm since more target graph snapshots are explored for finding one or more matchings for a given pattern graph snapshot. The graph parameters for which no solution exists (indicated in orange in Tables 3 and 4), increasing the size of the sliding window has no effect on the runtime of the algorithm since the algorithm does not continue exploring the snapshots in this case. And graphs for which the algorithm could not terminate before timeout (NULL values in Table 3) with initial size of the sliding window, it does not terminate when the size of the sliding window is increased.

Finally, it is noteworthy that we do not recommend increasing the size of the sliding window while using the proposed algorithm. During our experiments, we noted that the number of matchings found, for a given pattern and target graph pair, with different sizes of the sliding window remained the same. Increasing the sliding window size only increases the algorithm runtime without affecting the "goodness" of the final solution.

7 Conclusion

In this paper, we focused on the problem of subgraph isomorphism in case of dynamic graphs. Although this problem has been well studied for static graphs, it is still in its infancy for dynamic graphs. One of the possible reasons behind this could be the plethora of dynamic graph models present in the literature since the choice of the model affects the problem definition and, in turn, the performance of the designed algorithms.

We proposed an algorithm for solving the induced version of this problem. The proposed algorithm inherently considers the snapshot-based representation of dynamic graphs and respects the temporal order of these snapshots. This approach is novel since the existing state-of-the-art algorithms model the dynamic graphs as an aggregated graph with timestamped edges. The difference in the modeling of the dynamic graphs affects the algorithm design. For the existing algorithms, the chronological

order of the edges presents an important constraint, whereas in our case, the chronological order of the entire graph snapshot takes precedence.

The core of our algorithm is another algorithm (VF3) designed for the case of static graphs. We extend it to be applied to dynamic graphs and discuss the pruning strategy for the reduction of the state-space using the knowledge of previously matched nodes. We present the theoretical time complexity analysis of VF3 which helps in outlining the time complexity analysis of our algorithm. Furthermore, we tested the performance of our algorithm and compared it with two existing state-of-the-art algorithms using the real-world datasets. From the results, it was concluded that our algorithm is the second best overall in terms of runtime, especially for larger target and pattern graphs. One of the existing state-of-the-art could not solve any graph instances for larger pattern graphs. Furthermore, it was concluded that the snapshot-based model of dynamic graphs is probably not the best choice for solving the problem of subgraph isomorphism. However, since in some applications, it is not possible to model a dynamic graph using an aggregated model with timestamped edges, the existing algorithms are not applicable. Since the performance of our algorithm is at par with that of the existing algorithms, it can be applied in cases where the snapshot-based model of dynamic graphs is required. In fact, we have applied the proposed algorithm for detecting dynamic patterns in two diverse domains - urban road traffic and invasive team sports in our previous works.

Acknowledgements

This work has been funded partially with the support from European Union with the European Regional Development Fund (ERDF) and with the support of the pLaTINUM ANR project (ANR-15-CE23-0010-01).

Statements and Declarations

Competing interests

The authors declare that they have no competing interests.

Availability of data and materials

The datasets used in this paper as well as the source code can be found here: `https://github.com/KamalDSOberoi/DynamicSubgraphIsomorphism`

References

[1] T. Alakörkkö and J. Saramäki. Circadian rhythms in temporal-network connectivity. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 30(9):093115, 2020.

[2] I. Almasri, X. Gao, and N. Fedoroff. Quick mining of isomorphic exact large patterns from large graphs. In *2014 IEEE International Conference on Data Mining Workshop*, pages 517–524, Dec 2014.

[3] Michele Berlingerio, Francesco Bonchi, Björn Bringmann, and Aristides Gionis. Mining graph evolution rules. In Wray Buntine, Marko Grobelnik, Dunja Mladenić, and John Shawe-Taylor, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 115–130. Springer Berlin Heidelberg, 2009.

[4] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinformatics*, 14(7):S13, Apr 2013.

[5] Eren Cakmak, Udo Schlegel, Dominik Jäckle, Daniel Keim, and Tobias Schreck. Multiscale snapshots: Visual analysis of temporal summaries in dynamic graphs. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):517–527, 2021.

[6] 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 Transactions on Pattern Analysis and Machine Intelligence*, 40(4):804–818, April 2018.

[7] Vincenzo Carletti, Pasquale Foggia, Antonio Greco, Alessia Saggese, and Mario Vento. Comparing performance of graph matching algorithms on huge graphs. *Pattern Recognition Letters*, 2018.

[8] Sutanay Choudhury, Lawrence Holder, George Chin, Khushbu Agarwal, and John Feo. A selectivity based approach to continuous pattern detection in streaming graphs. In *18th International Conference on Extending Database Technology*, pages 157–168, 2015.

[9] Donatello Conte, Pasquale Foggia, Mario Vento, and Carlo Sansone. Thirty Years Of Graph Matching In Pattern Recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 18(3):265–298, 2004.

[10] Stephen A. Cook. The complexity of theorem-proving procedures. In *Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, USA, 1971. ACM.

[11] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, Oct 2004.

[12] Joseph Crawford and Tijana Milenković. Cluenet: Clustering a temporal network based on topological similarity rather than denseness. *PLOS ONE*, 13(5):1–25, 2018.

[13] Fabien Diot, Elisa Fromont, Baptiste Jeudy, Emmanuel Marilly, and Olivier Martinot. Graph mining for object tracking in videos. In Peter A. Flach, Tijl De Bie, and Nello Cristianini, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 394–409, Berlin Heidelberg, 2012. Springer.

[14] Aswathy Divakaran and Anuraj Mohan. Temporal link prediction: A survey. *New Generation Computing*, 38(1):213–258, 2020.

[15] Wenfei Fan. Graph pattern matching revised for social network analysis. In *Proceedings of the 15th International Conference on Database Theory*, page 8–21, 2012.

[16] Wenfei Fan, Jianzhong Li, Jizhou Luo, Zijing Tan, Xin Wang, and Yinghui Wu. Incremental graph pattern matching. In *International Conference on Management of Data*, pages 925–936, New York, NY, USA, 2011. ACM.

[17] R. Giugno and D. Shasha. Graphgrep: A fast and universal method for querying graphs. In *Object recognition supported by user interaction for service robots*, volume 2, pages 112–115 vol.2, Aug 2002.

[18] Huahai He and Ambuj K. Singh. Graphs-at-a-time: Query language and access methods for graph databases. In *International Conference on Management of Data*, pages 405–418, New York, NY, USA, 2008. ACM.

[19] Lauri Kovanen, Márton Karsai, Kimmo Kaski, János Kertész, and Jari Saramäki. Temporal motifs in time-dependent networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2011(11):P11005, nov 2011.

[20] Lauri Kovanen, Kimmo Kaski, János Kertész, and Jari Saramäki. Temporal motifs reveal homophily, gender-specific patterns, and group talk in call sequences. *Proceedings of the National Academy of Sciences*, 110(45):18070–18075, 2013.

[21] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, 2014. Accessed Jan. 2022.

[22] Giorgio Locicero, Giovanni Micale, Alfredo Pulvirenti, and Alfredo Ferro. TemporalRI: A subgraph isomorphism algorithm for temporal networks. In Rosa M. Benito, Chantal Cherifi, Hocine Cherifi, Esteban Moro, Luis Mateus Rocha, and Marta Sales-Pardo, editors, *Complex Networks & Their Applications IX*, pages 675–687, Cham, 2021. Springer International Publishing.

[23] P. Mackey, K. Porterfield, E. Fitzhenry, S. Choudhury, and G. Chin. A chronological edge-driven approach to temporal subgraph isomorphism. In *2018 IEEE International Conference on Big Data*, pages 3972–3979, Dec 2018.

[24] Ciaran Mccreesh, Patrick Prosser, Christine Solnon, and James Trimble. When Subgraph Isomorphism is Really Hard, and Why This Matters for Graph Databases. *Journal of Artificial Intelligence Research*, 61:723 – 759, 2018.

[25] J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19(3):229 – 250, 1979.

[26] Giovanni Micale, Giorgio Locicero, Alfredo Pulvirenti, and Alfredo Ferro. TemporalRI: Subgraph isomorphism in temporal networks

with multiple contacts. *Applied Network Science*, 6(1):55:1–55:22, 2021.

[27] Kingshuk Mukherjee, Md Mahmudul Hasan, Christina Boucher, and Tamer Kahveci. Counting motifs in dynamic networks. *BMC Systems Biology*, 12(1):1–12, 2018.

[28] Ashwin Paranjape, Austin R. Benson, and Jure Leskovec. Motifs in temporal networks. In *10th International Conference on Web Search and Data Mining*, pages 601–610, New York, NY, USA, 2017. ACM.

[29] Ursula Redmond and Pádraig Cunningham. Subgraph isomorphism in temporal networks. *arXiv preprint arXiv:1605.02174*, 2016.

[30] Giulio Rossetti and Rémy Cazabet. Community discovery in dynamic networks: A survey. *ACM Computing Surveys*, 51(2):1–37, feb 2018.

[31] Benjamin Schiller, Sven Jager, Kay Hamacher, and Thorsten Strufe. Stream - a stream-based algorithm for counting motifs in dynamic graphs. In Adrian-Horia Dediu, Francisco Hernández-Quiroz, Carlos Martín-Vide, and David A. Rosenblueth, editors, *Algorithms for Computational Biology*, pages 53–67, Cham, 2015. Springer International Publishing.

[32] Douglas C. Schmidt and Larry E. Druffel. A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices. *Journal of ACM*, 23(3):433–445, July 1976.

[33] Konstantinos Semertzidis and Evaggelia Pitoura. Top-k durable graph pattern queries on temporal graphs. *IEEE Transactions on Knowledge and Data Engineering*, 31(1):181–194, 2019.

[34] Christine Solnon. Alldifferent-based filtering for subgraph isomorphism. *Artificial Intelligence*, 174(12):850 – 864, 2010.

[35] X. Sun, Y. Tan, Q. Wu, and J. Wang. Hasse diagram based algorithm for continuous temporal subgraph query in graph stream. In *6th International Conference on Computer Science and Network Technology*, pages 241–246, Oct 2017.

[36] J. Tang, S. Scellato, M. Musolesi, C. Mascolo, and V. Latora. Small-world behavior in time-varying graphs. *Phys. Rev. E*, 81:055101, May 2010.

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

[38] Julian R. Ullmann. Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. *Journal of Experimental Algorithmics*, 15:1.6:1.1–1.6:1.64, 2011.

[39] Mario Vento. A long trip in the charming world of graphs for pattern recognition. *Pattern Recognition*, 48(2):291 – 301, 2015.