Data Mining and Knowledge Discovery
https://doi.org/10.1007/s10618-024-01061-8

ArcMatch: high-performance subgraph matching for labeled graphs by exploiting edge domains

Vincenzo Bonnici[1] · Roberto Grasso[2] · Giovanni Micale[3] · Antonio di Maria[3] · Dennis Shasha[4] · Alfredo Pulvirenti[3] · Rosalba Giugno[5]

Received: 9 August 2023 / Accepted: 12 July 2024

Abstract
Consider a large labeled graph (network), denoted the *target*. Subgraph matching is the problem of finding all instances of a small subgraph, denoted the *query*, in the target graph. Unlike the majority of existing methods that are restricted to graphs with labels solely on vertices, our proposed approach, named can effectively handle graphs with labels on both vertices and edges. ntroduces an efficient new vertex/edge domain data structure filtering procedure to speed up subgraph queries. The procedure, called path-based reduction, filters initial domains by scanning them for paths up to a specified length that appear in the query graph. Additionally, ncorporates existing techniques like variable ordering and parent selection, as well as adapting the core search process, to take advantage of the information within edge domains. Experiments in real scenarios such as protein–protein interaction graphs, co-authorship networks, and email networks, show that s faster than state-of-the-art systems varying the number of distinct vertex labels over the whole target graph and query sizes.

Keywords Subgraph isomorphism · Domain reduction · Path-based reduction · Labelled graphs

1 Introduction

Graphs are mathematical objects used to represent the overall topological relationship among a given set of entities. Entities can be represented by vertices, while their pairwise relationships can be represented by edges. Such a topological structure is often enriched with labels that represent specific properties of vertices and edges.

Responsible editor: Michelangelo Ceci.

Extended author information available on the last page of the article

Applications abound. In computational chemistry, atoms are modelled as vertices labelled with atom symbols, and edges represent chemical bonds (Balaban 1985). Living cells are often modelled via protein–protein interaction networks (Stelzl et al. 2005), where vertices are proteins and edges are their interactions. Vertex labels may denote functional properties of a protein, and edge labels might denote the type of interaction, for example, physical bond or expression correlation. Similar models are employed to represent a wide variety of phenomena of biological systems (Clark et al. 2021), and, in general, to represent relationship data by means of heterogeneous networks (Petković et al. 2022; Bing et al. 2023). In social science, graphs are used to represent social networks, in which vertices are persons and each edge represents a type of social relationship (Zheng and Skillicorn 2017).

A key step in analysing graph structures is to search for a specific substructure, often called a subgraph query, within a given target graph. The goal is to find all or a given number of occurrences of the query within the target graph. The correspondence or mapping between query and target vertices of each occurrence can help to recognize functional modules in biological networks (Milo et al. 2002), to detect hardware trojans (Piccolboni et al. 2017), to identify reusable patterns in software designs (Chaturvedi et al. 2018), or to optimize the geometry of parts and buildings (Zeng et al. 2019; Cao and Hall 2021). In the realm of web platforms like e-commerce, social media, and financial systems, fraud detection is a common key task solved by representing contacts as annotated graphs. Subgraphs of the contact graph in which particular agents engage with a sizable pool of users to mask their identities and activities (Pourhabibi et al. 2020) may indicate fraud. Similarly, a cyber attack is a deliberate and malicious act aimed at compromising the integrity, confidentiality, or availability of data or services within an information system. The ability to attribute such attacks is essential for security but is also notoriously challenging. Subgraphs are queried to identify the initiator of an attack, defining a mapping between an attack pattern and a history log (Avellaneda et al. 2019). Communication devices (i.e routers and switches) play a critical role in the reliable functioning of embedded system networks. They need to be tested in conjunction with many different configurations that represent operating networks. Starting from an available test system set and a multitude of test cases. Subgraphs are queried to determine the mapping that associates the test case elements (the logical network topology) with the appropriate elements of the test systems (the physical network topology) (Strandberg et al. 2018). Uveal melanoma (UM) is a highly malignant intraocular tumour with a poor prognosis and response to therapy. The metastatic microenvironment contains high levels of tumour-associated macrophages (TAMs) that correlate positively with a worse patient prognosis. Given a network-based representation of the TAMs, certain forms of subgraphs identify potential targets for the immunomodulation. The selected targets will be used for pharmacophore-based virtual screening against a library of FDA-approved chemical compounds, followed by refined flexible docking analysis (Weich et al. 2024). In general, it is not clear when two molecular networks, composed of genetic elements with feedback interactions, are similar. One such measure is the size of the set of overrepresented subgraphs that are similar. The method proposed by Huang et al. (2022) clusters networks with diverse functionalities based on the frequency distribution of common subgraphs.

This type of search can also be performed on top of modern graph database systems, such as Neo4j (Hoksza and Jelínek 2015). However, graph database systems, by default, solve a different version of subgraph isomorphism (SubGI). In that default mode, they allow two distinct query vertices to match the same target vertex in a given SubGI instance. Figure 1 gives an example of a use case where distinct query nodes should map to distinct target nodes. The query is to verify within a co-authorship network how many cliques of 3 authors (q_0, q_1 and q_2) that are affiliated with the same department have co-authored a paper with at least two other authors (q_3 and q_4). This example query might be part of research studying cooperation among members of the same department.

As another example, in searching for molecular structures the user does not want to allow two different query atoms to match the same target atom. The same applies to people in social network analysis (Archibald et al. 2021).

For this reason, we focus our attention only on algorithms that solve the definition of subgraph isomorphism in which distinct query vertices must match distinct target vertices, as originally formulated in Ullmann (1976) and in the main related literature.

The problem is known to be NP-Complete (Cook 1971), and several heuristic methodologies for solving subgraph isomorphism have been proposed over the years. An early solution was presented in Ullmann (1976) by Ullmann, which models the problem as a backtracking search. The search space can be seen as a tree which encodes partial mappings and their extensions to complete solutions.

In Cordella et al. (2001), Carletti et al. (2017b), the authors enhanced the search with a set of look-head rules for predicting unfeasible branches of the search space. In Solnon (2010), McCreesh et al. (2020), concepts developed for solving constraint satisfaction problems have been applied. Constraint satisfaction aims at verifying constraints among a set of variables once specific values are assigned to them. The application to subgraph isomorphism represents query vertices as variables whose possible values are target vertices. Labels and edges are represented as constraints of the problem. The set of target vertices that are compatible (i.e. they have the same label) with a given query vertex is called vertex domain. Thus, feasible combinations of target vertices are tested to assess the correspondence of their relationships with the query topology and labels. Sophisticated procedures, called reduction techniques, are applied to refine domains before the verification step, thus reducing the search space and thus (hopefully) decreasing the running time. Recently Han et al. (2013),

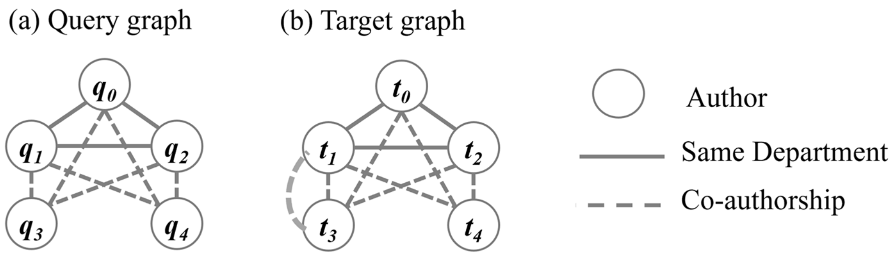

Fig. 1 A use case example of subgraph searching over a co-authorship network

Bi et al. (2016), Han et al. (2019), Kim et al. (2021, 2022), the concept of domains has been extended to edges. Such approaches outperform previous methodologies based on graph algebra (He and Singh 2008) and query decomposition strategies (Sun et al. 2012). This entailed the definition of a new data structure in which feasible correspondences between vertices and edges are stored. A disadvantage of the domain-based approach is the high memory consumption, especially in the presence of edge domains. In Bonnici et al. (2013), a matching order strategy of the query vertices has been defined to maximize the number of constraints that are verified at each step of the computation. Such an approach has been shown to work well to define the ordering of the variables (Sun and Luo 2020; Lee et al. 2012; Bonnici and Giugno 2017; Aparo et al. 2019).

This paper presents a novel approach, called hich works on graphs having labels on vertices and/or edges. It solves subgraph isomorphism and it returns the matching mappings between the query and the target graph, as well as the count of such mappings. ntroduces the following new methodologies for optimizing reduction techniques.

- ilters using vertex domains before filtering using edge domains. This reduces memory consumption.
- A further filtering technique, called path-based domain reduction, verifies the correspondence between paths in the query and the target graph.
- The matching order strategy, defined in Bonnici et al. (2013), aims at applying as many constraints as possible at each step of the searching process is extended in order to exploit the information on domains rather than the graph topology alone.
- The search process is cost-based where the cost is assumed proportional to the size of edge domains. Moreover, it is equipped with a dynamic parent selection that explores the search space according to the effective number of candidate vertices for a given partial mapping.

Path-based domain reduction including their theoretical properties as well as the cost-based search process constitute the main novel algorithmic features of the present work. In addition, the pipeline itself is new and contributes to the high performance of Further, ntroduces the concept of path-based reduction. In addition, rovides the first formal implementation of a backtracking search procedure which is completely driven by edge domains. It extends the valuable variable ordering technique defined in Bonnici et al. (2013) by equipping it with the information residing in domains, and by providing a subprocedure for peripheral query vertices.

Tests conducted on real graphs, where both vertices and edges are labeled, demonstrate that our approach exhibits notably faster performance compared to existing state-of-the-art tools when retrieving all the embeddings of a query graph within a target graph as opposed to just counting them. This is especially pronounced when dealing with high numbers of target and query labels. That tendency holds for all the tested benchmarks, that include protein–protein interaction networks, a co-authorship network, a email exchange network and synthetic graphs generated by most-used models (Barabási–Albert, the Erdos-Rényi and the Forest Fire). Several new or existing techniques are involved in the proposed methodology, making the performance analysis of their combination a hard task. For this reason, we conducted an ablation study aimed at analysing the performance contribution of each technique and combinations of them. This has led to the conclusion that in some cases path reduction is helpful and in some cases it is not. For that reason, we propose two algorithms nd a “light” version of called *lt* (in which path reduction is disabled). *lt* is particularly competitive when the results are capped at a certain number of mappings. The ode is open source and is available at https://github.com/vbonnici/ArcMatch.

In what follows, Sect. 2 gives the problem definition, defines the notation that is used throughout the manuscript, and reports the concepts of interest that currently represent the state of the art of the techniques involved in The novel techniques of re introduced in Sect. 3. Section 4 describes the experiments to assess the performance of s compared with the state-of-the-art. The section gives a description of the principal technical aspects of the existing methodologies that were compared with the proposed approaches, together with their limitations. Section 4 also reports a study regarding the scalability of the compared approaches on varying topological and labelling properties of the involved graphs. Lastly, Sect. 5 concludes the manuscript.

2 Background

This section presents the basic notions regarding graphs and then introduces state-of-the-art techniques for SubGI. A summary of the notation used in this manuscript is reported in Table 1.

2.1 Graphs

A simple graph is a tuple $G = (V, E)$, where V is the set of vertices and E is the set of edges. An undirected edge is a set $\{v, u\}$, with $v, u \in V$, which means that $E \in \{\{u, v\} : u, v \in V\}$. Thus, the graph may contain self-edges but not multi-edges (namely, multiple edges between that same pair of vertices). We say that the edge $\{v, u\}$ is incident to the vertices v and u. Two vertices are adjacent if there exists an edge between them. A path is a sequence of vertices $(v_1, v_2, \ldots, v_n)$, such that $v_j \in V$ for $1 \leq j \leq n$ and $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq n-1$. A vertex may appear multiple times in a path. A ring (or cycle) is a path $(v_1, v_2, \ldots, v_n)$ such that $v_1 = v_n$ and these two vertices are the only two to be repeated in the path, namely $\forall i, j : 1 < i, j < n, i \neq j \Rightarrow v_i \neq v_j$ (i.e. the ring has $n-1$ distinct nodes). The degree $deg(u)$ of a vertex u is given by the number of edges that are incident on it, thus $deg(u) = |\{\{v_i, v_j\} : v_i = u \ or \ v_j = u\}|$. The neighborhood $N(u)$ of a vertex u is given by the set of vertices that are adjacent to u, thus $N(u) = \{v : \{u, v\} \in E \ or \ \{v, u\} \in E\}$. Given a set of vertices $V' \subseteq V$, the neighborhood of V' is defined as $N(V') = (\cup_{v \in V'} N(v))$. A graph can be equipped with labels on vertices and on edges. Given a set of vertex labels A, the function $\alpha : V \mapsto A$ maps a vertex to a single label. Given a set of edge labels B, the function $\beta : V \mapsto B$ maps an edge to a single label (that is, there is one label per node though the label could itself have further structure, e.g. could be a string).

Table 1 Main notation used in this manuscript

Symbol	Definition	Description
G	$(V,E) : E \in \{\{u,v\} : u,v \in V\}$	A graph as a set of vertices V plus a set of edges E
α	$\alpha : V \mapsto A$	A function mapping vertices to labels
β	$\beta : E \mapsto B$	A function mapping edges to labels
G_q	(V_q, E_q)	A query graph
G_t	(V_t, E_t)	A target graph
$deg(u)$	$\|\{\{v_i, v_j\} : v_i = u$ or $v_j = u\}\|$	The degree of the vertex u
$N(u)$	$\{v : \{u,v\} \in E \; or \; \{v,u\} \in E\}$	The neighborhood of the vertex u
$N(V')$	$(\cup_{u \in V'} N(u)) \setminus V' : V' \subseteq V$	The neighborhood of the set of vertices V'
ω	$(v_1, v_2, \ldots, v_n) : v_i \in V, \{v_i, v_{i+1}\} \in E, \forall i : 1 \leq i \leq n-1$	A path of length $\|\omega\| = n$
ω_i	$(v_1, v_2, \ldots, v_i) : \omega = (v_1, v_2, \ldots, v_n), 1 \leq i \leq n$	The first i vertices of ω
$\omega[i]$	$v_i : \omega = (v_1, v_2, \ldots, v_n), 1 \leq i \leq n$	The ith vertex of the path ω
$\omega[i..j]$	$(v_i, v_{i+1}, \ldots, v_{j-1}, v_j) : \omega = (v_1, v_2, \ldots, v_n), 1 \leq i \leq j \leq n$	The subpath of ω from the ith to the jth vertex of it, both included
θ	$(v_1, v_2, \ldots, v_{\|V\|}) : v_i \in V$	An ordering of the vertices of V
θ_i	$(v_1, v_2, \ldots, v_i) : \theta = (v_1, v_2, \ldots, v_{\|V\|}), 1 \leq i \leq n$	A partial ordering, namely the first i vertices of θ
$\theta[i]$	$v_i : \theta = (v_1, v_2, \ldots, v_{\|V\|}), 1 \leq i \leq n$	The ith vertex of θ
(q_i, t_i)	$q_i \in V_q, t_i \in V_t$	A matching pair
M	$M : V_q \mapsto V_t$	A SubGI mapping of G_q into G_t as an injective function
M	$((q_1, t_1), (q_2, t_2), \ldots, (q_n, t_n))$	A SubGI mapping of G_q into G_t as an ordered set of matching pairs
M_i	$((q_1, t_1), (q_2, t_2), \ldots, (q_i, t_i)) : 1 \leq i \leq n$	The first ith pairs of the mapping M
$\mathbb{M}$	$\{M^1, M^2, \ldots, M^k\}, \|\mathbb{M}\| = k$	The entire set of matches between a query G_q and a target graph G_t
$D(q_i)$	$\{t_h \in V_t : t_h \simeq q_i\}$	The domain of the vertex q_i
$D(\{q_i, q_j\})$	$\{\{t_h, t_k\} \in E_t : t_h \in D(q_i), t_k \in D(q_j), \beta(\{t_h, t_k\}) = \beta(\{q_i, q_j\})\}$	The domain of the edge $\{q_i, q_j\}$

2.2 Subgraph isomorphism (SubGI)

Given a query graph $G_q = (V_q, E_q)$ and a target graph $G_t = (V_t, E_t)$, the *subgraph isomorphism problem (SubGI)*, also called monomorphism, consists in finding the occurrences of G_q in G_t. An occurrence is an injective mapping of query vertices to target vertices that preserves the topology and the labeling of the query graph. A mapping can be seen as a function $M : V_q \mapsto V_t$, such that $M(q_i) = t_i$. It is also represented as an ordered vector $M = ((q_1, t_1), (q_2, t_2), \ldots, (q_n, t_n))$, such that $q_i \in V_q$, $t_i \in V_t$. Each pair in M represents the mapping of a query vertex q_i to the target vertex t_i. The *all different* property must hold, viz. $\forall i,j$ *s.t.* $i \neq j$, $q_i \neq q_j$, and $t_i \neq t_j$. The mapping preserves the query topology if there exists a set of edges between the mapped target vertices that is equivalent to the set of query edges. Thus, $\forall \{q_i, q_j\} \in E_q \Rightarrow \{M(q_i), M(q_j)\} \in E_t$. The preservation of labels implies $\forall i : 1 \leq i \leq n \Rightarrow \alpha(q_i) = \alpha(t_i)$, and $\forall \{q_i, q_j\} \in E_q \Rightarrow \beta(\{q_i, q_j\}) = \beta(\{M(q_i), M(q_j)\})$. (Recall that there is at most one edge between two nodes and each edge has one label.)

A slightly different definition of SubGI is *induced SubGI*. This version of the problem introduces the further constraint that no edges can connect target vertices (belonging to an occurrence of the match) other than those present in the query. It is formally expressed as $\forall i,j : \{M(q_i), M(q_j)\} \in E_t \Rightarrow \{q_i, q_j\} \in E_q$.

Multiple mappings obeying the above constraints can exist, and each mapping identifies a given occurrence. The goal of SubGI is either to find all such occurrences or all occurrences up to a certain limit (e.g. 100,000).

Figure 2 shows two graphs having vertex labels represented by the colors, grey and white. The edge labels are represented by their trait, solid or dashed. The query graph occurs twice in the target graph of the example. A first match is given by $M^1 = ((q_4, t_0), (q_0, t_1), (q_1, t_4), (q_3, t_2), (q_2, t_5))$, and the second occurrence $M^2 = ((q_4, t_0), (q_0, t_1), (q_1, t_2), (q_3, t_4), (q_2, t_5))$ that is obtained by exchanging t_2 with t_4, thus by switching the match for q_1 from t_2 to t_4 and vice versa for q_3. The set of mappings of a query graph within a target graph is referred to as $\mathbb{M} = \{M^1, M^2\}$.

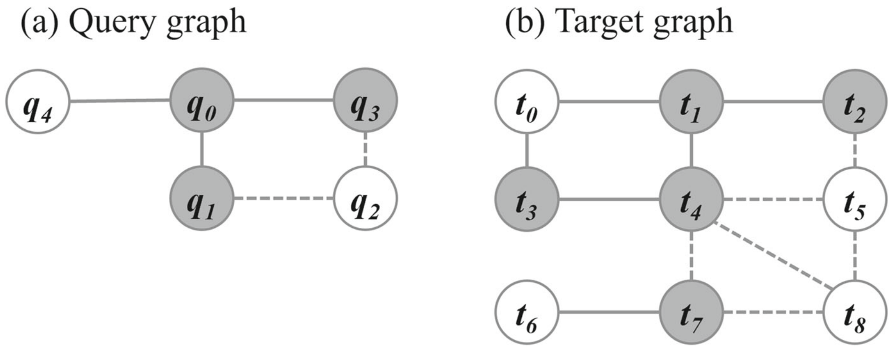

Fig. 2 A query graph and a target graph. Each vertex is labelled white or grey. Each edge is labelled solid or dashed

SubGI is an NP-Complete problem (Cook 1971). Backtracking algorithms have proven to have among the best performance (Ullmann 1976; Cordella et al. 2001; Carletti et al. 2017b; Bonnici et al. 2013), but other approaches based on Cartesian products or join operations have been proposed (Bi et al. 2016; Sun et al. 2012). By analogy to the constraint satisfaction problem (CSP), query vertices can be seen as variables and the goal is to assign values, namely target vertices, to them such that constraints are satisfied (Zampelli et al. 2010). The constraints include topological constraints, labeling constraints as discussed above, as well as the *all different* property of the mapping (Solnon 2010).

An exponential brute force algorithm tries to assign all possible values to variables, and for each complete assignment, the constraints are verified a posteriori. Heuristic algorithms use backtracking, for which the search space can be represented as a tree. The tree has a dummy root, and it has potentially ${V_q}^{|V_t|}$ nodes, plus the root. Each tree node (except the root) represents a vertex mapping (q_i, t_i). A path from each child of the root to a leaf of the tree represents a complete mapping solution. A path from the root to an intermediate node represents a partial solution. The tree defines the order in which mappings are investigated. The process of tree construction does not check constraints, so constraints must be verified separately. That verification can be performed for a full mapping at the leaves of the tree or for a partial mapping at intermediate nodes. In the latter case, if a partial solution violates at least one constraint, then the subtree branching from its last tree node can be ignored. In this way, the size of the search space can be drastically reduced.

Given a partial solution $M_i = ((q_1, t_1), (q_2, t_2), \ldots, (q_i, t_i))$, with $i < |V_q|$, the following constraints are verified:

(1) $\forall j : 1 \leq j < i \Rightarrow t_i \neq t_j$;
(2) $\alpha(q_i) = \alpha(t_i)$;
(3) $\forall j : 1 \leq j < i, \{q_i, q_j\} \in E_q \Rightarrow \{M(q_i), M(q_j)\} \in E_t$. Because we impose an ordering in M_i, subscript indexes represent the order of a given element in M_i.

2.2.1 Variable ordering

The performance of SubGI algorithms depends to a large extent on the order in which query vertices are included in the mapping. Formally, given a query graph $G_q = (V_q, E_q)$, an ordering $\theta = (v_1, v_2, \ldots, v_{|V_q|}) : v_i \in V_q$ is a sequence of the vertices in V_q. Given θ, a partial ordering $\theta_i = (v_1, v_2, \ldots, v_i)$ is defined as the first i vertices of θ. The ordering can be chosen statically (before the search process begins) or dynamically by exploiting the information in the current partial solution (Bonnici and Giugno 2017). Each approach has advantages for certain graphs and disadvantages for others.

Reference Bonnici et al. (2013) introduced a static variable ordering strategy. It is based on the most-constrained fail-first principle. The vertex that is most constrained is the one that is likely to cause a constraint failure. Constraints may be semantic (i.e. vertex label) or topological (i.e. the vertex degree, and edges). Intuitively, the vertex with the highest number of constraints is the one with the highest pruning power. Constraints come in the form of the edges that link a possible next vertex to other vertices that are already in a partial mapping. The more vertices that are already in the ordering and are linked to the next vertex, the more constraining the next vertex will be. Moreover, since multiple vertices with the same constraint value can exist, a tie-breaking rule is adopted. Such a strategy has been shown empirically to be an excellent strategy (Sun and Luo 2020; Lee et al. 2012; Bonnici and Giugno 2017; Aparo et al. 2019).

2.3 Domains

2.3.1 Vertex domains and vertex features

One way to improve the performance of SubGI solvers is to try to filter out some of the assigned values before the backtracking search process begins. For example, for a given query vertex q_i only those elements having the same label as q_i should be considered, rather than trying to assign all target vertices to it. Thus, for each query vertex q_i, a domain $D(q_i)$ is defined. Each domain contains the target vertices that are compatible with the corresponding query vertex. Compatibility is assessed based on the label, but also on the degree. Thus, the degree of the vertices in $D(q_i)$ must be greater than or equal to $deg(q_i)$. Compatibility can also be assessed by degree probability (Carletti et al. 2017b) or by looking at the neighbors of a vertex and their labels (Ullmann 1976).

Even more sophisticated compatibility criteria can be exploited, for example, in Han et al. (2019) the directed acyclic graph (DAG) induced by a vertex of the query graph is compared to the DAG induced by vertices in the target graph. The DAG induced by a vertex is retrieved by performing a breath-first visit of the graph. In general, every kind of invariant based on substructures of the graph topology can be used as a feature to establish vertex compatibility (Dahm et al. 2015; Ullmann 2011; Shang et al. 2008).

Paths, trees and subgraphs are the most used feature types for describing vertices (Sakr and Al-Naymat 2010; Han et al. 2011) by extracting their neighborhood. Such features are often involved in graph indexing techniques. The aim is to build an index of the target graph that can be used to reduce the searching time. The cost for indexing the graph is amortized if multiple queries are run. Surprisingly, sometimes indexing yields advantages in single-query executions (Giugno et al. 2013; Katsarou et al. 2015, 2017).

2.3.2 Domain reduction

Initial vertex domains can be refined via *reduction* procedures in which the global topology of the query is used to discard candidate portions of the target graph.

A well-known procedure is called *arc consistency* (Mackworth 1977), which ensures that target vertices belonging to the domains of two connected query vertices must be connected too. Formally, the procedure verifies that for each query edge $\{q_i, q_j\}$, $\forall t_k \in D(q_i) \exists t_h \in D(q_j) : \{t_k, t_h\} \in E_t$. In addition, we require $\beta(\{t_k, t_h\}) = \beta(\{q_i, q_j\})$. The verification procedure is applied to a given domain

$D(q_i)$ in order to discard target vertices in that domain that violate arc consistency. The removal of a target vertex may influence the state of other domains, previously reduced. Thus, it is necessary to run multiple iterations of the procedure. The number of iterations can either be bounded or the algorithm can be executed until convergence. Convergence is reached when a run produces no reduction of the current domains. Algorithm 1 shows a procedure for applying arc consistency until convergence. The procedure searches for edge existence between vertex domains by also verifying edge label compatibility.

Algorithm 1 Application of arc consistency until convergence.

```
Input:
the query edge set $E_q$,
the target edge set $E_t$,
the vertex domains $D$,
and the edge labelling function $\beta$.

procedure ArcConsistency($E_t$,$E_q$,$D$,$\beta$)
    $reduced \leftarrow true$
    while $reduced$ do
        $reduced \leftarrow false$
        for $\{q_i, q_j\} \in E_q$ do
            for $t_i \in D(q_i)$ do
                if $\nexists t_j \in D(q_j) : \{t_i, t_j\} \in E_t$ AND $\beta(\{t_i, t_j\}) = \beta(\{q_i, q_j\}))$ then
                    $D(q_i) \leftarrow D(q_i) \setminus \{t_i\}$
                    $reduced \leftarrow true$
                end if
            end for
        end for
    end while
end procedure
```

Figure 3a shows the initial vertex domains that are obtained when matching the query and the target graphs of Fig. 2. The target vertices t_1, t_2, t_4 and t_7 initially comprise the domain of the query vertex q_0. They are the target vertices that satisfy the vertex constraints, namely, they are grey and they have a degree equal to or greater than $deg(q_0)$. Vertex t_3 is excluded from the domain because its degree is less than the degree of q_0, while vertex t_0 is excluded because its label is white. The domain composition of the other vertices can be similarly explained. Arc consistency is applied to initial domains and the result of its application until convergence is shown in Fig. 3c. Figure 3b shows an intermediate phase. Thus, the edge $\{t_4, t_7\}$ is not considered between the domains of the vertices q_0 and q_3 because the edge $\{q_0, q_3\}$ is dashed while the edge $\{t_4, t_7\}$ is solid. In the example, vertex t_2 is in the domain of q_0 but it has no valid edge to the domain of vertex q_4, so it is discarded by arc consistency. Similarly, vertex t_7 has no valid edge to any vertex in $D(q_3)$, so it is discarded too.

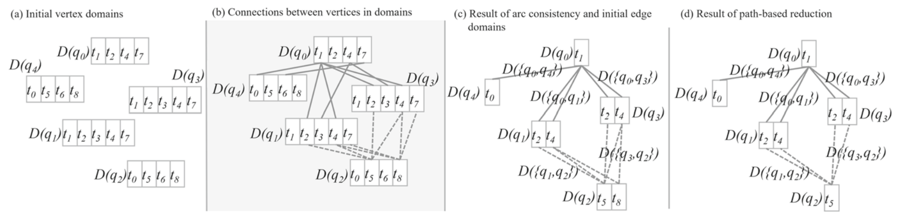
(a) Initial vertex domains
(b) Connections between vertices in domains
(c) Result of arc consistency and initial edge domains
(d) Result of path-based reduction
D(q0) t1 t2 t4 t7
D(q4) t0 t5 t6 t8
D(q3) t1 t2 t3 t4 t7
D(q1) t1 t2 t3 t4 t7
D(q2) t0 t5 t6 t8
D({q0,q4})
D({q0,q3})
D({q0,q1})
D({q3,q2})
D({q1,q2})

Fig. 3 Domains, and their transformation by means of reduction techniques, obtained in solving the example of Fig. 2

Vertex reduction procedures can also be applied during the search process (Haralick and Elliott 1980; Zampelli et al. 2010).

2.3.3 Edge domains and domain graph

The concept of vertex domain naturally extends to edges. Given two query vertices, q_i and q_j, their domains $D(q_i)$ and $D(q_j)$, and an edge $\{q_i, q_j\} \in E_q$ linking them, the domain of such an edge is given by $D(\{q_i, q_j\}) = \{\{t_k, t_h\} \in E_t : t_k \in D(q_i), t_h \in D(q_j)\}$. Vertex domains and edge domains can be combined into a single structure, here called a *domain graph* (DG), first introduced in Han et al. (2013) and then refined in Bi et al. (2016) and subsequently in Han et al. (2019).

An example of a domain graph is given in Fig. 3c. The complex data structure is able to represent vertex domains and all the edges connecting their elements.

3 Methods

The main steps of the proposed methodology for scanning all the occurrences of a query graph G_q over a target graph G_t are:

1. compute initial vertex domains
2. run arc consistency over vertex domains
3. compute initial edge domains
4. run path-based reduction procedure
5. choose a static variable ordering
6. run the backtracking phase to locate matching occurrences.

Step (1) is performed by comparing the labels and degrees of query and target vertices such that, for a query node q_i, $D(q_i) = \{t \in V_t : \alpha(q_i) = \alpha(t), deg(q_i) \leq deg(t)\}$. Arc consistency in step (2) is performed as described in Sect. 2.3.2, and it can be run until convergence or not. Initial edge domains, in step (3), are computed as described in Sect. 2.3.3. Thus, given an edge $\{q_i, q_j\}$, its domain is computed as $D(\{q_i, q_j\}) = \{\{t_k, t_h\} \in E_t : t_k \in D(q_i), t_h \in D(q_j)\}$. The domain graph is composed of vertex and edge domains (defined in Sect. 2.3.3). In step (4), a novel path-based reduction (presented in Sect. 3.1) is applied to the domain graph. The search phase is driven by a new static ordering (step (5)) which combines domain cardinality with the filtering power of a vertex, presented in Sect. 3.2. The backtracking SubGI searching phase (step (6)) is completely driven by the domain graph, and it is described in Sect. 3.3. Candidates of the next vertex to be mapped are chosen by selecting an already mapped vertex, called *parent*. A novel procedure for its selection is given. Furthermore, the search is equipped with a novel *ad hoc* procedure for managing a particular type of query vertices, called *peripheral*, which is introduced in Sect. 3.2.1. Thus, steps 4, 5, and 6 constitute the main algorithmic novelty of the proposed approach.

3.1 Path-based domain reduction

Paths represent a good compromise between complexity and efficacy when used as graph and/or vertex features in graph indexing approaches for SubGI (Bonnici et al. 2010; Giugno et al. 2013). Thus, we decided to exploit this concept in implementing a path-based reduction procedure, here simply called path reduction, to discard elements in the domain graph.

The path reduction procedure is performed after arc consistency has been run on vertex domains until convergence. Initial edge domains are extracted as described in Sect. 2.3.3. Then, the reduction is run based on comparing the paths that start from a given query vertex with the paths starting from target vertices. ompares query paths to target paths as follows. For each target vertex t_h in the domain $D(q_i)$, the reduction verifies that for each path (up to a given length) starting from q_i, at least one corresponding path starts from t_h. The correspondence is verified by looking at vertex and edge domains. Formally, given a path $(u_1, u_2, \ldots, u_n)$ such that $u_1 = q_i$ (the starting node), $u_j \in V_q$ and $\{u_{j-1}, u_j\} \in E_q$ for $2 \leq j \leq n$, a target vertex t_h is included in the domain of q_i if and only if there exists at least one path $(v_1, v_2, \ldots, v_n)$ such that $v_1 = t_h$, $v_j \in D(u_j)$ and $\{v_j, v_{j+1}\} \in D(\{u_j, u_{j+1}\})$. Since target paths are extracted from the domain graph, label compatibility is implicitly ensured. A *maximal path* is a path whose length is exactly *lp* or is a ring, or it cannot be extended. Formally, given a graph $G = (V, E)$ and a path length parameter *lp*, let $paths_G(lp)$ be the set of paths in G having length *lp*, and let $rings_G(lp)$ be the set of rings in G having length *lp*. Moreover, let $mpaths_G(n)$ the set of paths in G of length n such that $\forall \omega = (v_1, v_2, \ldots, v_n), \omega \in paths_G(n) : N(\omega) \backslash \{v : v \in \omega\} = \emptyset$. The set of maximal paths of G with parameter *lp* is defined as $maxpaths_G(lp) = paths_G(lp) \cup rings_G(lp) \cup \{mpaths_G(n) : 1 \leq n < lp\}$.

Maximal paths from a minimal length of 3 to a maximal length of 6 are taken into account. Such a length is a user-defined parameter. A path length of 6 achieves a good trade-off between computational costs and filtering power (Bonnici et al. 2010; Giugno et al. 2013). Note that the consistency of paths of length 2 is already ensured by arc consistency if edge labels are taken into account during the process.

Figure 3d shows an example of path reduction after the application of arc consistency. The figure shows that path reduction is more powerful than simple arc consistency. The target vertex t_8 does not violate arc consistency. In fact, it has at least one valid edge between vertices in the domains $D(q_1)$ and $D(q_3)$. However, the query graph (shown in Fig. 2) has the ring $(q_2, q_3, q_0, q_1, q_2)$. Thus, from vertex t_8 of domain $D(q_2)$, we should be able to navigate the domain graph (Fig. 3c) and to obtain a compatible path. However, such a path can not be extracted from the domain graph. The only ring that can be obtained starting form t_8 is $(t_8, t_4, t_1, t_4, t_8)$ but that would violate the *all different* constraint because t_8 appears twice. Thus, t_8 is removed form $D(q_2)$.

Figure 3 also gives an example to motivate why arc consistency should be applied before path-based reduction. The intermediate step of the panel (b) of the figure is never materialized by the proposed methodology for two reasons. First, arc consistency is able to substantially reduce the domains. This can be a substantial cost because each edge domain can have up to $|E_t|$ elements. Thus, arc consistency represents a fast low-memory technique for reducing both vertex domains and edge domains.

The example of Fig. 3 shows the effectiveness of arc consistency. Panel (b) of the figure shows the complete initial data structure of the proposed SubGI instance. It contains a high number of candidates in both vertex and edge domains. Applying arc consistency on the initial vertex domains (panel (a) of the figure) yields an instance of edge domains as shown in panel (c) of the figure. Panel (c) of the figure is sensibly smaller than panel (b).

Algorithm 2 implements the procedure for extracting maximal paths from the query graph. The procedure is a depth-first search (DFS) which takes as input the current visited path ω and the max path length lp. The variable ω can also be seen as the stack which drives the DFS visits. PathReduction visits only query vertices, and it is run by setting each query vertex as the source, namely $PathReduction(\omega = [v])$ for $v \in V_q$. Push and pop operations are implemented by adding a vertex to the tail of the vector $\omega[i+1] \leftarrow v$, and by removing the vertex from the tail $\omega \leftarrow \omega[1...i]$. In each recursive DFS call, the neighbors of the vertex that are at the top of the stack are scanned in order to extend the path. If a neighbor is at the bottom of the stack, then such a path is a ring. Otherwise, only vertices that are not already in the stack are visited. The assertion of *not visited* is here expressed as $v \notin \omega$. However, an array of Booleans can be used to efficiently implement the search such that the `visited` flag is activated/deactivated during the recursive process. When a valid neighbor is pushed on the stack, it is checked and whenever a maximal path of length lp is formed, a recursive DFS call is run. After this, the neighbor is removed from the stack.

The algorithm takes a trace of paths that cannot be extended by exploiting the variable `extended`, such that each time a maximal path is obtained as an extension of the current path, or the current path is a non-expandable path, the `VerifyPath` procedure is called to verify the domains.

Algorithm 2 Path-based reduction procedure. It implements a recursive DFS over the query graph for retrieving maximal paths (up to length *lp*).

```
Input:
a path $\omega$,
the path reduction parameter $lp$,
and the vertex and edge domains $D$.

procedure PathReduction($\omega$,$lp$,$D$)
    $i \leftarrow |\omega|$
    $u \leftarrow \omega[i]$
    $extended \leftarrow false$
    for $v \in N(u)$ do
        if $v = \omega[1]$ then                              ▷ It is a ring
            $\omega[i+1] \leftarrow v$
            $VerifyPath(\omega, true, D)$
            $\omega \leftarrow \omega[1 \dots i]$
        else                                                 ▷ Try to extend it
            if $v \notin \omega$ then
                $extended \leftarrow true$
                $\omega[i+1] \leftarrow v$
                if $i+1 = lp$ then                           ▷ Max-length path
                    $VerifyPath(\omega, false, D)$
                end if
                $PathReduction(\omega, lp, D)$
                $\omega \leftarrow \omega[1 \dots i]$
            end if
        end if
    end for
    if not $extended$ then                                   ▷ Unextendable path
        $VerifyPath(\omega, false, D)$
    end if
end procedure
```

Once a maximal path ω is found, Algorithm 3 verifies that each target graph vertex in the domain of the source vertex $\omega[1]$ is compatible with $\omega[1]$ If a given target vertex is not compatible with $\omega[1]$, then it is removed from the domain of $\omega[1]$.

Algorithm 3 Path reduction by verifying the existence of at least one path in the domain graph corresponding to the query path ω.

Input:
a path ω,
a variable *isRing* to specify if ω is a ring or not,
and the vertex and edge domains D.

```
procedure VerifyPath($\omega$, $isRing$, $D$)
    $reduced \leftarrow false$
    for $v \in D(\omega[1])$ do
        $\widehat{\omega} \leftarrow (v)$
        if not $VerifyPathDFS(\omega, \widehat{\omega}, isRing, D)$ then
            $D(\omega[1]) \leftarrow D(\omega[1]) \setminus \{v\}$
            $reduced \leftarrow true$
        end if
        if $reduced$ then
            $RefineDomains(v)$
        end if
    end for
end procedure
```

Algorithm 4 Path reduction to verify the existence of at least one path $\widehat{\omega}$ in the domain graph corresponding to the query path ω.

Input:
a query path ω,
a target path $\widehat{\omega}$,
a variable *isRing* to specify if ω is a ring or not,
and the vertex and edge domains D.
Return:
true if $\widehat{\omega}$ is a path in the target graph that corresponds to the query path ω according to the vertex and edge domains, *false* otherwise.

```
procedure VerifyPathDSF($\omega, \widehat{\omega}, isRing, D$)
    if $|\widehat{\omega}| = |\omega|$ then return $true$
    else
        $i \leftarrow |\widehat{\omega}|$
        $u_q \leftarrow \omega[i]$
        $v_q \leftarrow \omega[i+1]$
        for ($\{u_t, v_t\} \in D(\{u_q, v_q\}) : u_t = \widehat{\omega}[i]$) do
            if $isRing$ AND ($|\widehat{\omega}| = |\omega| - 1$) then
                if $v_t = \widehat{\omega}[1]$ then return $true$
                end if
            else
                if $v_t \notin \widehat{\omega}$ then
                    $\widehat{\omega}[i+1] \leftarrow v_t$
                    if $VerifyPathDFS(\omega, \widehat{\omega}, isRing, D)$ then return $true$
                    end if
                    $\widehat{\omega} \leftarrow \widehat{\omega}[1 \dots i]$
                end if
            end if
        end for
        return $false$
    end if
end procedure
```

The compatibility reduction is performed by Algorithm 4 which extracts paths starting from the target vertex. The extraction is done recursively by a DFS visit over the target graph and according to the current state of edge domains. Given a query path ω, a corresponding target path $\widehat{\omega}$ is searched by matching vertices in ω with vertices in the corresponding domains. Edge domains guide how to go forward in the path. In extending the path from position i to position $i+1$, we take into account the two consecutive query vertices $\omega[i]$ and $\omega[i+1]$. The partial path $\omega[1...i]$ is already mapped to some target vertices, and the procedure has to find a vertex to which to map $\omega[i+1]$. Because the two query vertices are consecutive in the path, the edge $\{\omega[i], \omega[i+1]\}$ links them. Moreover, domains are in a consistent state such that if $\{u_t, v_t\}$ in $D(\{\omega[i], \omega[i+1]\})$ then $u_t \in D(\omega[i])$ and $v_t \in D(\omega[i+1])$. Thus, a vertex for $\omega[i+1]$ can be extracted from the edges in $D(\{\omega[i], \omega[i+1]\})$. The procedure also verifies that each extension does not contain duplicate vertices, except in the case of rings. In a ring the first and the last vertex are equal. The procedure checks that constraint separately.

The reason why we decided to perform Algorithm 2 before Algorithm 3 is to avoid an exponential explosion in memory requirements. The reason is that for each query path, an exponential number of paths could match within the target graph. Thus, if the DFS visit over the query graph is performed together with the multiple DFS visits over the target graph, an exponential number of parallel DFSs must be run and stored at the same time.

Similarly to arc consistency, the removal of one candidate in a domain may affect the consistency of the other domains. Thus, a consistency check procedure must be applied. The procedure is described in Algorithm 5. It first checks the edge domains linked to the query vertex whose domains have been changed. Then, the effects of such a reduction are propagated in an iterative way. At each iteration, the edge domain consistency is verified. Subsequently, vertices that are inconsistent with edge domains are removed.

"Appendix B" further investigates the relation between arc consistency and the proposed path-based reduction.

Algorithm 5 Refinement of domains after the alteration of the domain of the query vertex s.

```
Input:
a query vertex $s$,
the query edge set $E_q$,
and the vertex and edge domains $D$.

procedure RefineDomains($s$,$E_q$,$D$)
    for $\{s, v_q\} \in E_q$ do
        for $\{u_t, v_t\} \in D(\{s, v_q\})$ do
            if $u_t \notin D(s)$ then
                $D(\{s, v_q\}) \leftarrow D(\{s, v_q\}) \setminus \{u_t, v_t\}$
            end if
        end for
    end for
    $reduced \leftarrow true$
    while $reduced$ do
        $reduced \leftarrow false$
        for $\{u_q, v_q\} \in E_q$ do
            for $\{u_t, v_t\} \in D(\{u_q, v_q\})$ do
                if $u_t \notin D(u_q)$ OR $v_t \notin D(v_q)$ then
                    $D(\{u_q, v_q\}) \leftarrow D(\{u_q, v_q\}) \setminus \{u_t, v_t\}$
                    $reduced \leftarrow true$
                end if
            end for
        end for
        for $\{u_q, v_q\} \in E_q$ do
            for $u_t \in D(u_q)$ do
                if $\nexists \{x, y\} \in D(\{u_q, v_q\}) : x = u_t$ then
                    $D(u_q) \leftarrow D(u_q) \setminus u_t$
                    $reduced \leftarrow true$
                end if
            end for
        end for
    end while
end procedure
```

3.1.1 Relation among different values of *lp*

Theorem 1 *Given two different values of path length, lp_1 and lp_2, such that $lp_1 \leq lp_2$, if a target vertex is discarded from a given domain by using lp_1 then it is also discarded by using lp_2.*

Proof Given a query path $\omega = (q_1, q_2, \cdots, q_n)$ such that $n = lp_2$, the verification of ω entails the verification of each prefix of the path. Moreover, all the sets of paths that are scanned for lp_1 are scanned for lp_2 because paths of length lp_2 includes all the path of length $lp_1 < lp_2$.

Thus, when evaluating a path of length $n = lp_2$, the procedure `VerifyPath` will discard at least the vertices that `VerifyPath` discards when evaluating the smaller path length lp_1. □

Let $R_{FC}(G_q, G_t, lp)$ be the set of removal operations that are performed by a single run of `PathReduction` for a given value of lp and without propagation, namely by discarding the call to the procedure `RefineDomains`.

Theorem 2 *Given two different values of path length, lp_1 and lp_2, such that $lp_1 < lp_2$, $R_{FC}(G_q, G_t, lp_1) \subseteq R_{FC}(G_q, G_t, lp_2)$.*

Proof According to Theorem 1, given a query path $\omega = (q_1, q_2, \ldots, q_n)$ of length $n = lp_2$, the procedure `VerifyPath` includes the verification of ω_i such that $1 \leq i < n$. For $i = n - 1$, the extension of a target path $\widehat{\omega_{n-1}}$ to a path $\widehat{\omega_n}$ is performed by the procedure `VerifyPathDFS`. If no valid extension is found, then the procedure stops with a negative result. This means that a path that is consistent until length $n - 1$ may be inconsistent at length n. Thus, at length n a further set of removals is produced. Such a consideration can be applied inductively from $n - 2$ to $n - 1$, and in general from i to $i - 1$ with $1 < i < n$. Thus, for any $i = lp_1$ such that $1 < i < lp_2$, $R_{FC}(G_q, G_t, lp_1) \subseteq R_{FC}(G_q, G_t, lp_2)$. □

3.1.2 Safety of the path-based reduction

In what follows, we show the safety of *PathReduction*. Namely, the procedure does not discard from the domains any vertex or edge that is included in at least one match between the query and the target graph. Given a query graph G_q, we let $\mathbb{M}$ be the set of matches between G_q and a target graph G_t. In order to ensure the safety of the procedure, we must ensure that the reduced domains are safe for $\mathbb{M}$. Safety means that $\forall M \in \mathbb{M}, \forall v_q \in V_q \Rightarrow M(v_q) \in D(v_q)$ AND $\forall \{q_i, q_j\} \in E_q \quad w\{M(v_i), M(v_j)\} \in D(\{v_i, v_j\})$.

Theorem 3 *Given a query graph G_q, a target graph G_t, their corresponding set of matches $\mathbb{M}$, and a state of vertex and edge domains that are safe for $\mathbb{M}$, then `PathReduction` alters such domains such that they are still safe for $\mathbb{M}$.*

Proof Given a query graph G_q, let Ω_q to be the set of paths of G_q, from length 1 to the maximum path length in G_q. `PathReduction` visits a subset $\Omega'_q \subseteq \Omega_q$ of paths, depending on the lp parameter. Let v_t be a target vertex in the domain of v_q such that $\exists M \in \mathbb{M} : M(v_q) = v_t$. For each path $\omega_q \in \Omega'_q$ such that $\omega_q[1] = v_q$, there exists a corresponding path ω_t such that $\omega_q[i] = M(\omega_q[i]) = \omega_t[i]$ for $2 \leq i \leq |\omega_q|$. If we suppose that all the domains, except $D(v_q)$, are safe, then v_t can not be removed from $D(v_q)$ because for each path ω_q starting from v_q, $M(\omega_q[i])$ is in $D(\omega_q[i])$.

The reduction of edge domains is performed by the procedure `RefineDomains`. Since we suppose the safety of initial domains, `RefineDomains` cannot remove any target edge $\{t_i, t_j\}$ for which a corresponding mapping to $\{v_i, v_j\}$ exists in $\mathbb{M}$. This last assertion comes from the fact that a target edge is removed from an edge domain only after one removal from a vertex domain. Therefore, the cause for removing the edge $\{t_i, t_j\}$ from $D(\{v_i, v_j\})$ is that t_i has been removed from $D(v_i)$ or t_j has been removed from $D(v_j)$. Thus, the safety of the removal of an edge from a domain is ensured by the safety of the removal of one of its vertices from the corresponding domain. Thus, the overall path reduction procedure alters domains such that they are still safe for $\mathbb{M}$. □

3.2 Vertex ordering strategy

dopts a modified version of the strategy proposed in Bonnici et al. (2013) for defining a matching ordering among query vertices. Given a partial ordering θ_i, the node v to be chosen next in the ordering is selected among the neighbors of θ_i. Then, the neighborhood of v is divided into three sets:

- $\mathbb{N}_1(v) = \{u \in \theta_i : \{u, v\} \in E\}$
- $\mathbb{N}_2(v) = \{u \in N(\theta_i) : \{u, v\} \in E\}$
- $\mathbb{N}_3(v) = N(v) \setminus \{\mathbb{N}_1(v) \cup \mathbb{N}_2(v)\}$

Figure 4 shows an example of such a partition of the neighborhood of the vertex v.

We define five measures on a given vertex v for ordering query vertices:

- $N_1(v) = |\mathbb{N}_1(v)|$
- $N_2(v) = |\mathbb{N}_2(v)|$
- $N_3(v) = |\mathbb{N}_3(v)|$
- $N_4(v) = deg(v)$
- $N_5(v) = |D(v)|$

Thus, given a partial ordering θ_i and two query vertices, v' and v'' that are neighbors of at least one vertex in θ_i, the next vertex to be included in the ordering is chosen by comparing sequentially their N_i, for $1 \leq i \leq 5$, measures. The vertex v' with the smallest i such that $N_i(v') > N_i(v'')$, for $1 \leq i \leq 4$, is chosen. If two vertices have exactly the same values, then N_5 is taken into account such that the v' is selected if $N_5(v') < N_5(v'')$. If still no vertex can be chosen, then the one with the lowest identifier is selected.

An exception to the ordering rule is given by vertices with singleton domains. Such vertices are put before the non-singleton vertices in the ordering.

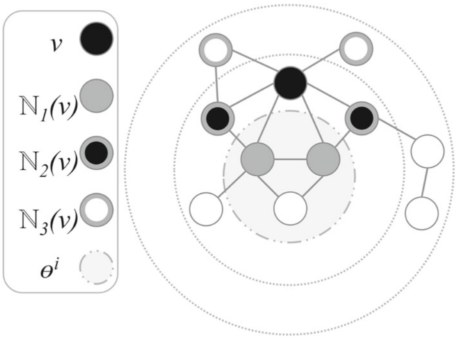

Fig. 4 A 3-ways division of the neighborhood of a vertex v given a partial ordering θ_i

3.2.1 Peripheral vertices

Postponing Cartesian products of disjoint domains is a well-known heuristic for SubGI (Bi et al. 2016). Here, we propose a methodology which combines the postponing of peripheral query vertices with an *ad hoc* procedure for matching vertices with disjoint domains.

Given a query graph $Q = (V_q, E_q)$ and a partial ordering $\theta = (q_0, q_1, \ldots, q_j)$ defined on the query vertices, a query vertex q_i is *peripheral* if $|\{q_k : \{q_i, q_k\} \in E_q\}| = 1$

Two vertices, q_i and q_j, have joint domains if at least one element of $D(q_i)$ is also in $D(q_j)$. We do not take into account peripheral vertices with singleton domains since they are put at the being of the ordering. Then, the set of peripheral vertices having joint domains is retrieved, and only one vertex is selected for each set. The selected vertex is the one with the biggest domain. Selected vertices are put at the end of the matching ordering.

Once a partial solution regarding all non-peripheral disjoint vertices is found, domains of peripheral disjoint vertices can be independently reduced. They have disjoint domains, thus the *all different* property does not need to be tested among them. This reduces the time requirement. Moreover, if just the counting of the subisomorphisms is required, it can be quickly computed by multiplying the size of their reduced domains.

3.3 Extension of partial mappings and dynamic parent selection

Algorithm 6 implements the backtracking procedure of the search process. It recursively extends partial matches to complete solutions. In each recursive step, a parent query vertex s is chosen dynamically. Such a parent is the currently matched vertex which minimizes the number of candidates to the current state i (line 3). The minimization is obtained by counting the query edges that are in the domain $D(\{s, \theta[i]\})$. Once a parent is chosen, candidate target vertices are extracted by retrieving the subset of edges $\{u_t, v_t\} \in D(\{s, \theta[i]\})$ such that $M(s) = u_t$ (line 4). Thus, the target vertices of type $v_t \notin \widehat{\theta}$ (if not already matched) are evaluated. The evaluation is made by verifying the connectivity of each candidate with the vertices that are in the partial matching, according to the query topology (line 5–10). The verification is performed by exploiting the edge domains (line 7). If a candidate verifies the connectivity, then it is added to the partial match in order to extend it (line 11–16). If the length of the formed partial match is equal to the number of query vertices, then a complete solution has been found and it is reported by the algorithm (line 13) via the *Report match* instruction. *Report match* is a wrapper of an abstract function which is in charge of counting or listing the matches.

Algorithm 6 Backtracking procedure for searching all the occurrences starting from a given target vertex $\widehat{\theta}[1]$ by following the variable ordering θ.

```
Input:
a query vertex ordering $\theta$,
a list $\widehat{\theta}$ of matching target vertices according to the query vertex ordering,
the query edge set $E_q$,
and the vertex and edge domains $D$.

procedure SearchOccurrences($\theta, \widehat{\theta}, E_q, D$)
    $i \leftarrow |\widehat{\theta}|$
    $s \leftarrow \theta[\underset{x}{\mathrm{argmin}}\{\{u,v\} \in D(\{\theta[x], \theta[i]\}) : 1 \leq x < i\}]$
    for $\{u_t, v_t\} \in D(\{s, \theta[i]\}) : u_t = M(s)$ AND $v_t \notin \widehat{\theta}$ do
        $feasible \leftarrow true$
        for $\{\theta[i], \theta[j]\} \in E_q : j < i$ do
            if $\{v_t, \widehat{\theta}[j]\} \notin D(\{\theta[i], \theta[j]\})$ then
                $feasible \leftarrow false$
                break
            end if
        end for
        if $feasible$ then
            $\widehat{\theta}[i] \leftarrow v_t$
            if $|\theta| = |\widehat{\theta}|$ then
                report match $M = ((\theta[i], \widehat{\theta}[i]) : 1 \leq i \leq |\theta|)$
            else
                $SearchOccurrences(\theta, \widehat{\theta}, E_q, D)$
            end if
            $\widehat{\theta} \leftarrow \widehat{\theta}[1 \ldots i-1]$
        end if
    end for
end procedure
```

Please recall that the proposed algorithm is not intended to work with multi-graphs, which admit multiple edges between two vertices. The restriction to simple graphs ensures that the list of candidates does not contain duplicate vertices, thus no duplicate matches are reported. Moreover, because the `SearchOccurrences` procedure performs tail recursion, we implement it in an iterative fashion, thus eliminating the call stack overhead.

3.4 Overall matching procedure and its complexity

Algorithm 7 describes the pseudo-code of the overall approach. It solves the SubGI problem by calling at line 13 the function that is defined in Algorithm 6, `SearchOccurrences`.

Algorithm 7 Matching procedure

```
Input:
the query graph $G_q = (V_q, E_q)$,
the target graph $G_t = (V_t, E_t)$,
the path reduction parameter $lp$,
the vertex and edge domains $D$,
and the edge labelling function $\beta$.

1: procedure MATCH($G_q$,$G_t$,$lp$,$D$,$\beta$)
2:     for $q_i \in V_q$ do                                    ▷ initial vertex domains
3:         $D(q_i) \leftarrow \{v \in V_t : \alpha(q_i) = \alpha(v)$ AND $deg(q_i) \leq deg(v)\}$
4:     end for
5:     $ArcConsistency(E_t, E_q, D, \beta)$          ▷ arc consistency over vertex domains
6:     for $\{q_i, q_j\} \in E_q$ do                            ▷ initial edge domains
7:         $D(\{q_i, q_j\}) \leftarrow \{(x, y) \in E_t : x \in D(q_i)$ AND $y \in D(q_j)$ AND $\beta(\{x, y\}) = \beta(\{q_i, q_j\})\}$
8:     end for
9:     for $q_i \in V_q$ do                                    ▷ path-based reduction
10:        $\omega \leftarrow (q_i)$
11:        $PathReduction(\omega, lp, D)$
12:    end for
13:    $\theta \leftarrow variable\_ordering(G_q, D)$                  ▷ variable ordering
14:    for $u_t \in D(\theta[1])$ do                               ▷ backtracking
15:        $\widehat{\theta} \leftarrow (u_t)$
16:        $SearchOccurrences(\theta, \widehat{\theta}, E_q, D)$
17:    end for
18: end procedure
```

In what follows, we give an analysis of the time complexity of the overall procedure. In doing this, we suppose that the set-theoretical membership operator $\in$ can be computed in constant time (e.g., using hashing), as well as the mapping of an element to its labels, and the comparison of the labels of two elements.

The complexity for building the initial vertex domains (lines 2–4) is $\Theta(|V_q| \cdot |V_t|)$, assuming we can compute the degree of a vertex in constant time (again using, for example, hashing).

The time complexity of *ArcConsistency* is $O(e \cdot k^3)$Dechter et al. (2003), where e is the number of constraints to verify and k is the size of the largest vertex domain. In our case, $e = |E_q|$ because the set of query edges are the constraints. The procedure works only over vertex domains, thus, k is the size of the largest vertex domain that can be at most $|V_t|$. Thus, the complexity of the procedure equals $O(|E_q| \cdot |V_t|^3)$.

The complexity of retrieving the initial edge domains (lines 6–7 of Algorithm 7) is $O(|E_q| \cdot |E_t|)$.

The variable ordering regards the query vertices. At each step of the ordering, a vertex must be chosen by scanning those not already included in the ordering. After the selection, the five measures reported in Sect. 3.2 of the neighbors of the selected vertex must be updated. Thus, an upper bound to the time complexity of the procedure is given by $O(|V_q| \cdot (|V_q| + |V_q|)) = O(|V_q|^2)$, because a vertex can have at most $|V_q|$ neighbors.

Algorithm 4 has complexity proportional to the number of paths of length *lp* that can be extracted from the query graph. For each query path, it scans all the corresponding target paths, thus the complexity is multiplied by the size of vertex domains. Let d_q be the average degree of the query graph. Then, an approximate number of paths of length *lp* is given by the falling factorial $(d_q)_{lp}$. If we suppose that each edge domain contains each vertex edge, *VerifyPath* comes with a complexity of $O(lp \cdot |E_t|)$. Such a cost is multiplied by the total number of paths, thus we obtain $O((d_q)_{lp} \cdot lp \cdot |E_t|)$.. Lastly, for each query path, domains are refined by Algorithm 5. Such a procedure has a complexity proportional to $O(|E_q| \cdot |E_t| \cdot t)$, where t is the number of times the cycle at line 10 is repeated. Because this repetition cannot be easily predicted, we give an upper bound: it can be performed at most $|V_t|$ times if each query vertex domain contains each target vertex and if one element of the domain is discarded at each cycle. Unfortunately, we cannot make any assumptions regarding the size of the domains and the number of times the refinement procedure is run. Thus, Algorithm 4 has a complexity bounded by $O(|E_t| \cdot ((d_q)_{lp} \cdot lp + |E_q| \cdot |V_t|))$. However, if we suppose $(d_q)_{lp} \cdot lp << |E_q| \cdot |V_t|$, then the complexity becomes $O(|E_t| \cdot |E_q| \cdot |V_t|)$. This means that it equals the cost of *ArcConsistency* when $|E_t| = |V_t|^2$ multiplied by a factor $((d_q)_{lp} \cdot lp)$ proportional to the number of query paths. That expresses the fact that the proposed path reduction technique extends *ArcConsistency* by generalizing from single edges to paths.

Lastly, the *SearchOccurrences* procedure implements the searching process of the subgraph isomorphism problem which is known to be NP-Complete Cook (1971). The problem can be modelled as a constraint satisfaction problem where variables are the query vertices and values are the target vertices. The size of a domain of a query vertex can be at most $|V_t|$, and the process can be modelled as a simple combinatorial process that generates all possible combinations of $|V_q|$ target vertices. The total number of possible combinations is $\Theta(|V_t|^{|V_q|})$. However, for each combination, we must verify the constraints (represented by the query edges). Thus, the overall complexity is bounded by $O(|V_t|^{|V_q|} \cdot |E_q|)$ if we admit that checking the existence of an edge between two matched target vertices has a constant cost. In our case, the search process is driven by edge domains. It is shown to be complete and safe (see next Section), thus it scans for the entire set of $|V_t|^{|V_q|}$ combinations. Algorithm 6 implements a non-redundant generation of such combinations by enumerating elements in edge domains (line 4). At each step of the process, the query edges linking vertices to a previous state of the process are verified (lines 5–11). Thus, for each combination, each edge query edge is verified only once. Assuming that set inclusion of line 7 can be performed in constant time, the complexity is bounded by $O(|V_t|^{|V_q|} \cdot |E_q|)$, because unfeasible combinations are never examined. At each step of the process, the algorithm chooses a parent vertex by means of line 3. Such a choice can be computed statically before the search process, thus its total complexity is $\Theta(|E_q|)$. Thus, the search process has overall complexity $O(|V_t|^{|V_q|} \cdot |E_q|)$.

As a result, the time complexity of the entire matching procedure of as two main contributions. The first contribution is given by the path reduction procedure, with a cost bounded by $O(|E_t| \cdot ((d_q)_{lp} \cdot lp + |E_q| \cdot |V_t|))$. The second main contribution is given by the search process, bounded by $O(|V_t|^{|V_q|} \cdot |E_q|)$. Overall, the time requirement of the path reduction has a predominant sub-cost $(|E_q| \cdot |V_t|)$ due to the domain refinement. Turning off the refining step noticeably reduces the time spent on pre-processing. On the other hand, the more precise the reduction is, the fewer unfeasible combinations are explored during the search process. Furthermore, we recall that the filtering power of the reduction also depends on the degree of compatibility between query and target elements, which is generally unpredictable.

3.5 Safety and completeness

In what follows, we prove the safety and completeness of Algorithm 7. Given a query G_q and a target G_t graph, and let $\mathbb{M}$ be the entire set of mappings of G_q in G_t, safety means that every match found by Algorithm 7 is in $\mathbb{M}$. Completeness means that every match in $\mathbb{M}$ is found by Algorithm 7.

A match is an assignment of values to every query variable, that is the assignment of a target vertex to every query vertex. Domains define the compatibility between query and target vertices. The initial domains contain the entire set of target vertices.

A brute force algorithm generates all possible assignments of target vertices to query vertices, and, for each assignment, verifies the constraints of the SubGI instance. It is safe because only assignments that verify constraints are returned. A brute force algorithm is complete because it generates every possible assignment, which implies that $\mathbb{M}$ is a subset of the generated assignment. By contrast, Algorithm 7 applies an initial filtering to vertex domains such that, for each query vertex, only target vertices having the same labels and compatible degrees are selected. In addition, similarly to vertex domains, an initial safe filtering is applied by checking label compatibility. Thus, initial edge domains do not violate safety.

`ArcConsistency` is safe for $\mathbb{M}$, because Theorem 6 shows that the filtering operations executed by `ArcConsistency` are a subset of those executed by `PathReduction`, and Theorem 3 shows that `PathReduction` is safe. Therefore, `ArcConsistency` does not violate safety and completeness. Edge domains are built from a set of vertex domains that are safe for $\mathbb{M}$. Thus, Algorithm 6 is run on a safe set of domains.

The backtracking procedure for generating the assignment, namely `SearchOccurrences`, is run after setting the first variable in the ordering to a value that is in its domain. This means that every partial mapping of length 1 does not violate safety. Thus, safety and completeness depend on Algorithm 6.

The difference between Algorithm 6 and the brute force algorithm described above is that it generates potential assignments by scanning edge domains rather than vertex domains. However, the two procedures have the same aim, which is to find target vertex combinations that satisfy the SubGI constraints.

Theorem 4 *Algorithm 6 is safe.*

Proof Algorithm 6 is safe because for every partial assignment, and thus for every complete assignment, it verifies the *all different* constraint and all the topological constraints. The *all different* constraint is verified by requiring $v_t \notin \widehat{\theta}$. For a given position i ($1 \leq i \leq |V_q|$). The topological constraints that apply to $\theta[i]$ can be divided into three groups. Group one contains the edges that link to previous positions of θ. Those are verified by lines from 6 to 9. Group two contains the edge that link $\theta[i]$ with its parent positions, namely $\{s, \theta[i]\}$, where s is the chosen parent state. This constraint is verified because every extracted target edge $\{u_t, v_t\} \in D(\{s, \theta[i]\})$ is, by construction of $D(\{s, \theta[i]\})$, in E_t and this verifies edge label compatibility. Group three contains the edges that link to future states, and that will be verified in such future states. At position $i = |\theta|$, only the first two groups of edges can occur because no further states exist. Vertex label compatibility is verified because $\{u_t, v_t\} \in D(\{s, \theta[i]\})$ implies $v_t \in D(\theta[i])$, which implies $\alpha(v_t) = \alpha(\theta[i])$, by construction. In conclusion, every SubGI constraint is verified, thus Algorithm 6 is safe. □

Safety implies that the set of assignments generated by the algorithm is a subset of $\mathbb{M}$. However, to prove completeness, we need to show that the assignment set is exactly $\mathbb{M}$.

Theorem 5 *Algorithm 6 is complete, if domains are safe for a given* $\mathbb{M}$.

Proof Given a state i ($1 \leq i \leq |\theta|$), an algorithm is complete for θ_i if: (i) it is complete for θ_{i-1}; (ii) given the set of assignments generated for θ_{i-1}, identified by $\mathbb{M}_{i-1}$, it generates all the assignments in $\mathbb{M}_i$.

For $i = 1$, the completeness of the algorithm in ensured by the safety of $D(w[1])$. For $i > 1$, let A be the set of target vertices that extend a given mapping in $\mathbb{M}_{i-1}$ to one or more mappings in $\mathbb{M}_i$. Let s be the parent state chosen by the algorithm. Let B be the set of target vertices retrieved by extracting from $D((s, \theta[i]))$ all the target edges (u_t, v_t) for which $u_t = M(\theta[s])$. That is $B = \{v_t : \{u_t, v_t\} \in D(\{s, \theta[i]\})$ AND $u_t = M(s)\}$. Thus B is the set of unverified target vertices that Algorithm 6 retrieves at line 4 to extend partial solutions. Unverified means that the SubGI constraints have still to be checked for those vertices. If the algorithm is complete for θ_{i-1}, then $A \subseteq B$ because of the safety of edge domains. The verification is performed at line 4 by ensuring $v_t \notin \widehat{\theta}$, and at lines 5–10 by verifying topological constraints. The verification produces a set $B' \subseteq B$ of target vertices. $B' = A$ because of the safety of the algorithm. Then, since, by hypothesis, the algorithm is complete for θ_{i-1}, it is necessarily complete for θ_i because B' contains all and only those vertices that extend mappings of $\mathbb{M}_{i-1}$ to mappings in $\mathbb{M}_i$. When $i = |\theta|$, $\mathbb{M}_i$ equals $\mathbb{M}$, which means that the produced mappings are exactly the complete solutions of the SubGI instance. Therefore, Algorithm 6 is complete. □

In general, the ordering is not a relevant aspect regarding safety and completeness, because matches in $\mathbb{M}$ are independent of it. However, it is important to notice that every possible ordering includes each query vertex exactly once.

4 Experiments

We run a detailed evaluation of all the techniques that are explained above through an ablation study. After considering the results of such an evaluation, that are reported in "Appendix A", we decided to release two different versions of hich exploit a different combination of the techniques described above. The standard version, simply called fully exploits edge domain reduction. By contrast, *ArcMatch-lt* performs vertex domain reduction until convergence but does not perform edge domain reduction. Both configurations use edge domains during the search process and employ specialized techniques for peripheral query vertices (see Sects. 2.3.2, 3.1, 3.2.1).

Note: s intended to be the very general-purpose version of the proposed approach. However, some existing tools focus on solving a special case of SubGI in which they report only up to a certain predefined maximum number of matches. To compare with such systems when the predefined number is relatively small, *lt* is better.

We have compared ith other systems by taking into account the features and limitations of the state-of-the-art approaches RI-DS (Bonnici et al. 2013), DAF (Han et al. 2019), VEQ (Kim et al. 2021), Glasgow (McCreesh et al. 2020). RI-DS speeds up the search process by giving precedence to query vertices that maximize the number of constraints that can be verified at a given step of the process (see also Sect. 3.2). Because RI-DS already includes vertex domains and arc-consistency, it can be used to evaluate the effectiveness of introducing edge domains and path-based reduction. DAF is a recent method that uses a combined vertex and edge domain structure. Instead of path-based reduction, it reduces domains by comparing the directed acyclic graphs induced by vertices. DAF manages graphs with labels only on vertices, not on edges. This limitation also affects its predecessors Turbo$_{ISO}$ (Han et al. 2013) and CFL-Match (Bi et al. 2016). VEQ is based on the DAF techniques but it exploits the constraints induced during the matching process to reduce the search space. The released VEQ software searches only up to 100k matches, and, similarly to DAF, allows labels on vertices but not on edges. The Glasgow solver is a recent algorithm based on constraint programming. It makes use of domain-specific search and inference techniques for solving computationally hard SubGI instances which are often represented by graphs with a few labels.

Table 2 reports the functional features of the state-of-the-art software packages with which we compare.

RI-DS, and lt), and Glasgow are able to list all the subgraph isomorphism matches between the query and the target graph. By contrast, current implementations of DAF and VEQ return only the count of the matches.

DAF (and similarly VEQ) first performs the combinatorial search on a core set of query vertices with constraints affecting other vertices in the core. Peripheral vertices, whose constraints do not affect subsequent vertices, are examined later.

Table 2 Functional features of the compared approaches

Algorithm	Vertex labels	Edge labels	Listing	Counting	Count limit
ArcMatch	✓	✓	✓	✓	No limit
RI-DS	✓	✓	✓	✓	No limit
Glasgow	✓	✓	✓	✓	No limit
DAF	✓			✓	No limit
VEQ	✓			✓	100k

Thus, the core set is examined using a SubGI searching approach. Then, for each instance of the core set, the number of global instances produced by extending the core is calculated by exploiting the current domain sizes of peripheral vertices. This strategy implies that DAF does not give the details of global matches, so it cannot return the mapping instances between the target graph and the query. Moreover, the available version of VEQ does not retrieve all the matches between a query graph and a target graph, but it stops at a maximum of 100,000. Thus, the results reported here refer to the counting problem. Further, in the tests in which VEQ is included, algorithms were modified to stop at the first 100k occurrences. Every approach reported in Table 2 can solve the counting problem. Thus, our experiments show results for counting, rather than for the listing problem.

Concerning the techniques introduced in VF2 (Cordella et al. 2001) and VF3 (Carletti et al. 2017b), in Bonnici et al. (2013) it is shown that RI-DS outperforms VF2. VF3 is a state-of-the-art algorithm based on a set of feasibility rules for matching vertices that are applied during the search process algorithm, and it allows labels on vertices and edges. Unfortunately, the currently available executable can solve only induced subgraph isomorphism. for this reason, we performed a comparison between VF3 and a version of hat has been modified for searching induced substructures.

Each test in each benchmark consisted of a single query on a single target graph, and by setting a timeout of 600 s. Queries were randomly extracted subgraphs from target graphs. The query extraction procedure uses a uniform random distribution to pick up a vertex from the target graph. That constitutes the initial vertex of the query. Subsequently, within the target graph, a neighbor of the vertices that currently comprise the query is randomly selected. The selected vertex is added to the query together with all the edges that link it with any of the vertices already in the query. The process is repeated until the desired number of vertices (or edges) is reached.

The command `/usr/bin/time -f`“`%e %s %M`” was used to measure time and memory consumption. In addition, internal timers have been injected into the source code. All tests were run on an Intel(R) Core(TM) i7-5960x with 64-Gb of RAM machine running a Ubuntu 64-bit 18.04 LTS system. All the evaluated tools are written in C++. DAF and VEQ are provided as executable files but without source code.

We applied the empirical non-parametric paired test described in Katari et al. (2021) to evaluate the statistical significance of a predominance of an algorithm with respect to the others. In particular, we tested whether the algorithm with the lowest average running time was also statistically the best one. A formal description is given in “Appendix C”. We put an asterisk on the charts if the fastest algorithm is statistically significantly better than all other algorithms at a *p* value level of 0.05 or less.

We also compute speed-ups of the proposed approach with respect to the state-of-the-art methodologies. In doing this calculation, we take into account timeouts (when a query takes over 600 s). Then, we divided the running time of the compared algorithm by the running time of or *ArcMatch-lt*), and we computed the average along all the tested queries.

In what follows, Sect. 4.1 gives details of the benchmarks used for the comparisons. Section 4.2 reports the main tests that regard the search for subgraph isomorphism with an unlimited number of matches to be reported. The results of tests performed by limiting the number of reported matches to the first 100k occurrences in Sect. 4.3. This type of search is not the main goal of the proposed approach, but we show such results to enable comparison with VEQ. Section 4.5 reports results on searches of induced subgraphs. This type of search is not the main goal of the proposed approach, but we show such results to enable comparison with VF3. Section 4.4 reports scalability tests according to the time requirements of the compared algorithms. The memory requirement of all the compared algorithms is below 300Mb, so we consider memory consumption to be irrelevant. However, memory footprints are reported in “Appendix D”. Lastly, Sect. 4.6 reports a comparison between nd a well-known graph database management system, Neo4j. Because of the complexity of database management systems, the reported results are aimed at showing that the performance of such systems can still be improved.

4.1 Benchmarks

We conducted the experiments on two different types of graphs, depending on the type of labels that are taken into account. *Fully labelled graphs* refers to experiments on graphs in which both vertices and edges have one label each. By contrast, *vertex labelled graphs* concerns experiments in which only vertex labels are used. Graphs also differ by the application domain: protein–protein interactions networks, co-authorship and email networks.

The PPI (Protein–Protein interaction) networks data set is a popular benchmark for subgraph isomorphism (Bonnici et al. 2013), because it is composed of different target graphs varying in size and density. It contains PPIs belonging to 11 different species, that have from 5000 to 12000 vertices, and an average degree ranging from 8 to 54. Target graphs have labels on vertices and edges. The number of labels varies from 8, 16, 32, 64, 128 to 256. Queries were randomly extracted from the labelled target graphs by setting the number of vertices to 4, 8, 12, 16 or 32. For each number of query vertices, 10 queries were extracted from each labelled target. The average query density is 0.169, with minimum and maximum values of 0.057 and 0.437, respectively.

DBLP is a co-authorship network in which vertices are authors and they are connected if they published at least one research paper together. We downloaded the

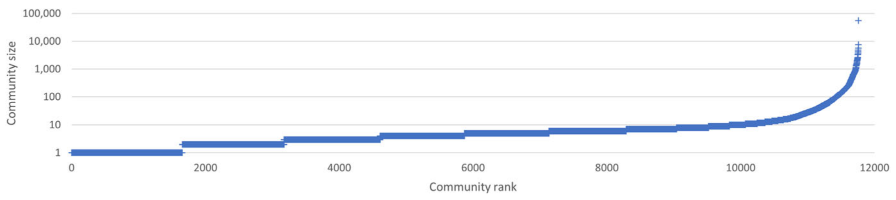

Fig. 5 Rank (x-axis) obtained by order communities according to their size (y-axis) in increasing order. By assigning a distinct label to each community. The chart shows that labels are not uniformly distributed within the graph. In particular, there is one community which predominates with more than 50k members

network from the SNAP database.[1] The graph has 317,080 vertices and 1,049,866 edges. We randomly labelled the network with 20 (as proposed by DAF's authors Han et al. (2019)), 100 and 1000 vertex labels. Moreover, we labelled the network with a set of 11,760 labels corresponding to the output of a community detection procedure, as described in Yang and Leskovec (2015). Each label identifies a specific community: each vertex is labelled according to the one community it belongs to. Figure 5 reports the size of communities in terms of the number of vertices. The largest community, consisting of 56k vertices, represents vertices which were not assigned to a particular community. Thus, 17% of the vertices are labelled with this particular 'unassigned' community. The second largest community consists of 7.5k (2%) vertices. The 50 (over 11k) largest communities cover in total 50% of the network.

Sets of 100 queries were randomly extracted for each of the four target networks (three random labelling plus one community membership). Each set is composed of queries having the same number of vertices ranging from 5, 10, 15, 20 to 50. Thus, 500 queries were extracted from each target graph, with a total of 2000 queries for the entire benchmark. The average query density is 0.191, with minimum and maximum values of 0.040 and 0.640, respectively.

We finally evaluate a network that was generated using email data, from October 2003 to May 2005, from a large European research institution. Vertices represent individuals, and a directed edge between individuals i and j is created if i sends at least one email to j. We downloaded the network from the SNAP database.[2] The graph has 265,214 vertices and 420,045 edges. Thus, compared to the other two benchmarks, it is more sparse. Because no vertex labels are provided in the original version, we randomly labelled the network with 20, 100 and 1000 different labels. Queries were extracted from one of the three labelled targets by varying the number of query vertices from 5 to 50.

4.2 Subgraph isomorphism

Our comparisons regarding subgraph isomorphism involve the state-of-the-art methodologies that have the same functionality as see the functionality discussion in Sect. 4 and reported in Table 2) in solving such a problem. This means that they can

[1] https://snap.stanford.edu/data/com-DBLP.html.

[2] https://snap.stanford.edu/data/email-EuAll.html.

handle graphs with labels on vertices and edges, and they search and return all the occurrences of the query graph within the target graph. Those are RI-DS, Glasgow and

Table 3 and Fig. 6 show the comparison over the PPI benchmark on target graphs having both vertex and edge labels. The proposed approach results in the fastest method for the majority of the problem instances that were taken into account. as an average speed-up of a factor of approximately 2 with respect to RI-DS and a factor of more than 80 with respect to Glasgow. s, however, outperformed by RI-DS for large queries (32 vertices). This disadvantage is mainly due to the costs of the reduction techniques implemented by They prove to be expensive on large queries or when the information coming from edge labels is not enough to cover the cost of exploiting it through reduction. RI-DS has less expensive reduction techniques and it completely avoids the handling of edge labels in such techniques.

Subsequently, we restricted the test to a graph having labels only on vertices in order to compare the proposed approach with DAF. It is a methodology in which the internal domain structure is more similar to thus it represents a more close approach. Figure 7 shows the comparison on the PPI benchmark for target graphs having vertex labels but ignoring edge labels. In this case, s the fastest algorithm when the results are investigated by looking at the trends (Fig. 7a) w.r.t. the query size. Up to 16 query vertices, s statistically better than all the other approaches. For 32 query vertices is still the fastest on average, but it becomes comparable to DAF and thus the statistical significance of its performance is lost. When the results are examined by looking at the trends on varying the number of vertex labels (Fig. 7b), again s the fastest approach for the majority of the instances except when graphs are equipped with a small number of labels (8 vertex labels). In this case, ecomes comparable to DAF. However, the Fig. 7c shows the details of such results. In particular, it shows that s statistically the best approach for queries having between 8 and 16 vertices, independently of the number of vertex labels. as an average speed-up of a factor of 547, 1418 and 1196 with respect to RI-DS, Glasgow and DAF, respectively.

Table 4 and Fig. 8a–c report the results of the proposed approach, RI-DS and DAF over the co-authorship network. as an average speed-up of a factor of 19 and 357 with respect to RI-DS and DAF, respectively. The smaller the number of random labels, the greater the running time of all the compared approaches. As noted, however, the unassigned community covers 17% of the network, so the overall label frequency distribution is skewed (see Fig. 5). Both factors affect the filtering power of each algorithm. utperforms other systems when the number of query vertices is sufficiently high, for example, when that number exceeds 15.

Finally, Fig. 9 reports results on the email benchmark. In this type of network, 20 labels are not enough to provide a sufficient amount of information to be exploited by the reduction procedure of Nevertheless, overall, as an average speed-up of a factor of 632 and 3.85 compared with RI-DS and DAF, respectively.

Table 3 Average running times in seconds over the PPI benchmark, grouping queries by: (1) the number of query vertices along with varying numbers of vertex and edge labels, (2) the number of vertex labels along with varying numbers of query vertices and edge labels, (3) the number of edge labels along with varying numbers of query vertices and vertex labels

Algorithm	Query vertices					Vertex labels						Edge labels					
	4	8	12	16	32	8	16	32	64	128	256	8	16	32	64	128	256
ArcMatch	0.02	0.02	0.02	0.03	0.07	0.04	0.03	0.03	0.03	0.03	0.03	0.04	0.03	0.03	0.03	0.03	0.03
RI-DS	0.05	0.05	0.05	0.05	0.05	0.05	0.04	0.05	0.04	0.05	0.05	0.06	0.05	0.05	0.05	0.05	0.05
Glasgow	1.81	2.03	2.25	1.92	1.99	2.70	1.98	1.89	1.80	1.77	1.76	3.71	1.89	1.88	1.88	1.89	1.89

s nearly almost the fastest

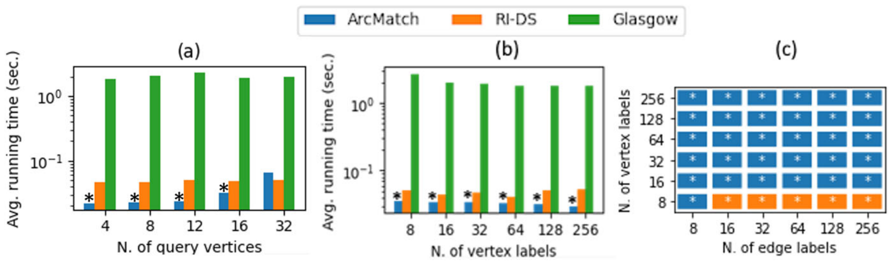

Fig. 6 Comparison of ith those systems that allow an arbitrarily large number of returned matches on the Protein–Protein Interaction (PPI) graphs with both vertex and edges labels. **a** Average running times grouped by the number of query vertices along with varying numbers of vertex and edge labels. **b** Average running times grouped by the number of vertex labels along with varying numbers of query vertices and edge labels. **c** Algorithm with the lowest average running time, grouping queries by the number of query vertices and edge labels, along with varying numbers of vertex labels. s nearly always the fastest, sometimes significantly so

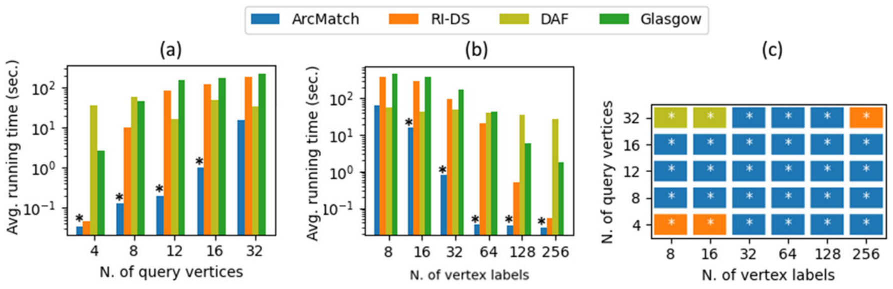

Fig. 7 Comparison of ith other systems that allow an arbitrarily large number of returned matches on the Protein–Protein Interaction (PPI) graphs with vertex labels but ignoring edge labels. Subfigures **a** and **b** depict the average running times, grouping queries by the number of query vertices along with varying the numbers of vertex labels, and by the number of vertex labels along with varying numbers of query vertices, respectively. Subfigure **c** shows the algorithm with the lowest average running time, grouping queries by the number of query vertices and the number of vertex labels. ends to dominate with 8 or more query vertices, as long as there are a sufficient number of vertex labels

4.3 Subgraphs isomorphism with a limited number of matches

In searching for the first 100k occurrences, all the approaches finished a similar number of instances as when searching for all the occurrences (not shown here). The analysis focuses on target graphs that have vertex labels but ignore edge labels in order to compare the proposed approach with VEQ, which is equipped with an evolved set of techniques introduced in DAF but whose available implementation only reports the first 100k occurrences. *lt* always outperforms or the 100,000 match problem. Complex path reduction procedures are not necessary for this sort of limited search. The convergence of vertex domain reduction is enough to reduce overall execution times. For this reason, we only show the performance of lt.

Figure 10 presents a comparison of retrieving the first 100k matches on the two real benchmarks. In general, VEQ finishes a total number of instances that is

Table 4 Average running times over the co-autorship network, grouping queries by: (1) the number of query vertices along with varying the numbers of vertex labels, (2) the number of vertex labels along with varying numbers of query vertices

Algorithm	Query vertices					Vertex labels			
	5	10	15	20	50	20	100	1000	11760
ArcMatch	1.97	2.92	6.06	6.51	14.70	8.94	0.30	0.23	16.27
RI-DS	0.39	5.53	21.11	40.66	185.64	82.02	0.40	0.39	119.84
DAF	154.85	155.26	176.23	166.03	207.47	70.36	7.10	10.42	600.0

For large numbers of query vertices or vertex labels, s the fastest

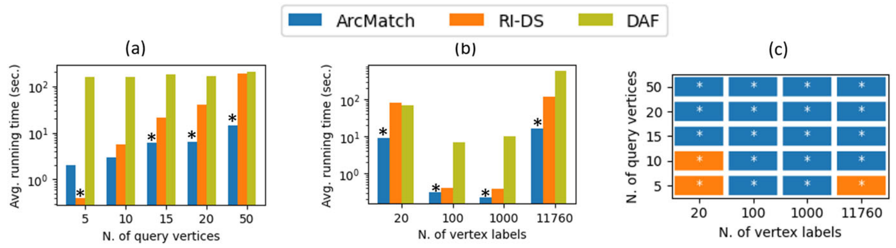

Fig. 8 Comparison of ith those systems that allow an arbitrarily large number of returned matches over the co-authorship benchmark (which has vertex labels only). Subfigures **a** and **b** depict the average running times, grouping queries by the number of vertex labels along with varying numbers of query vertices, and by the number of query vertices along with varying the number of vertex labels, respectively. Subfigure **c** shows the algorithm with the lowest average running time, grouping queries by the number of query vertices and the number of vertex labels. emonstrates superior performance when the number of query vertices is sufficiently high (e.g., 15 or more) or there are many vertex labels

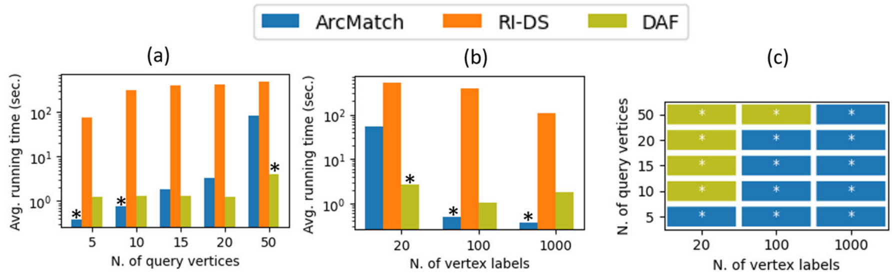

Fig. 9 Comparison of ith those systems that allow an arbitrarily large number of returned matches over the email benchmark (which has vertex labels only). Subfigures **a** and **b** depict the average running times, grouping queries by the number of vertex labels along with varying numbers of query vertices, and by the number of query vertices along with varying numbers of vertex labels, respectively. Subfigure **c** shows the algorithm with the lowest average running time, grouping queries by the number of query vertices and the number of vertex labels. emonstrates superior performance when the number of vertex labels is sufficiently high (e.g., 100 or more)

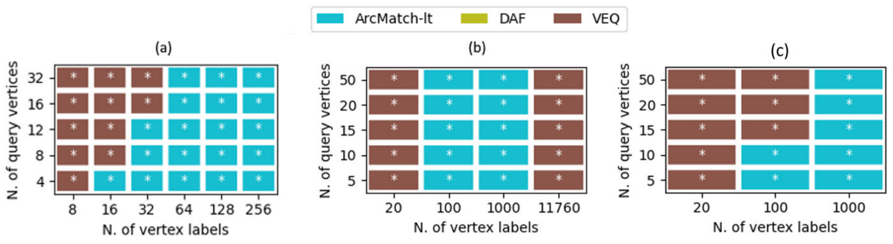

Fig. 10 The figure shows the algorithm with the lowest average running time, grouping queries by the number of query vertices and the number of vertex labels for the PPI benchmark (subfigure **a**), for the co-authorship benchmark (subfigure **b**), and for the email benchmark (subfigure **c**). lt outperforms the other approaches for a sufficiently high number of vertex labels when they are uniformly distributed (thus excluding the 11760-labels labeling of the co-authorship network) over the target graph vertices

comparable to that of lt (not shown here). Regarding the PPI benchmark, *ArcMatch-lt* is much faster than the other systems, especially when there are a large number of possible vertex labels, exceeding 64. *ArcMatch-lt* has an average speed-up of a factor of 1337 and 1.68 with respect to DAF and VEQ, respectively. For the co-authorship network, *ArcMatch-lt* exhibits the highest performance when the number of vertex labels is between 100 and 1000. VEQ performs better in other cases. *ArcMatch-lt* has an average speed-up of 429 and 1.45 concerning DAF and VEQ, respectively. Similar results are shown for the email networks, for which btain an average speed-up of 3.79 and 1.04 compared to DAF and VEQ, respectively.

4.4 Scalability tests on synthetic graphs

We performed a scalability analysis of the compared algorithms in order to investigate how the running time of the algorithm depends on various properties of the involved query and target graphs.

The number of target vertices is considered to be the most important factor affecting the running time of SubGI algorithms (Aparo et al. 2019; Carletti et al. 2017a; McCreesh et al. 2018; Carletti et al. 2013, 2020; Han et al. 2019). For that reason, we used synthetic graphs generated by means of specific random models: Barabási and Albert (1999), the Erdos and Rényi (1959) and the Forest Fire Leskovec et al. (2005) models. Such a collection was previously used in Aparo et al. (2019). Target graphs were created by fixing the desired number of vertices of target graphs, then varying the parameters of the random models which control the number of edges and thus the density of the graph. Query graphs were extracted randomly for target graphs. See (Aparo et al. 2019) for a detailed description of the benchmark.

We investigate the scalability of the monomorphism problem alone and only for those algorithms which have no strict limitation on the number of return matches. For that reason, VF3, VEQ and lt were excluded from the analysis. We also excluded the Glasgow algorithm for the analysis of synthetic graphs because it is optimized for search on unlabeled graphs.

For each benchmark, the running times of the algorithm were grouped by specific properties of the involved graph in order to investigate the dependency of the algorithm w.r.t. the specified properties. In particular, we took into account the number of vertices of the target graph, the density of the target graph, the number of distinct labels in the target graph and the number of vertices of the query graph. These properties are most often used to compare the performance of SubGI algorithms (Aparo et al. 2019; Carletti et al. 2017a; McCreesh et al. 2018; Carletti et al. 2013, 2020; Han et al. 2019).

Results regarding the Barabási and Albert (1999), the Erdos and Rényi (1959) and the Forest Fire (Leskovec et al. 2005) models are shown in Figs. 11, 12 and 13, respectively. All the analyses show that the running time of oes not depend on the number of target vertices and on variations in target density. By contrast, when the number of distinct target labels is large, xploits the filtering power of vertex labels very well. As a result, the running time of ecreases as the number of distinct target labels increases. While all algorithms improve with large numbers of vertices, xploits such information the best. Similarly, but with an opposite trend, all the algorithms show a clear dependence on the number of query vertices. Each algorithm shows a specific trend, but in general s the fastest algorithm for reasonable query sizes (up to 32).

There is a slight anomaly regarding Barabási–Albert and Forest Fire networks. In generators for that model, large target graphs have lower edge density because

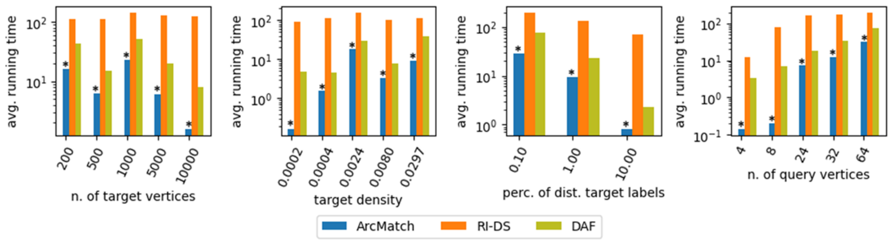

Fig. 11 Scalability analysis for the synthetic benchmark built by means of the Barabási–Albert model. Charts are drawn by grouping the running times of each algorithm according to the properties of the target and query graphs. While the number of target vertices and overall target density do not affect the relative computational requirements of the algorithms, the number of query vertices has a strong effect. onsistently wins, often in a statistically significant manner (low p value)

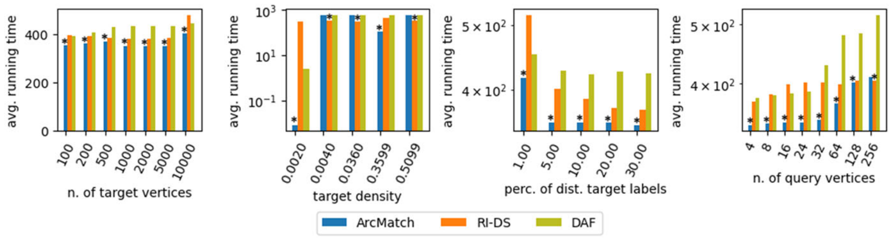

Fig. 12 Scalability analysis for the synthetic benchmark built by means of the Erdos model. Charts are drawn by grouping the running times of each algorithm according to properties of the target and query graphs. Number of target vertices and target density do not affect trends in computational requirements of the algorithms. Dependencies emerge for number of distinct target labels and number of query vertices. onsistently wins, often with a low p value

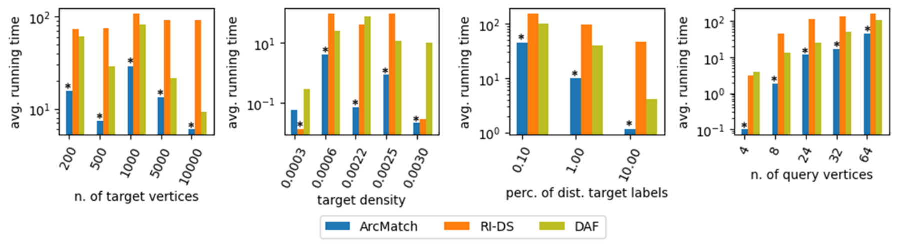

Fig. 13 Scalability analysis for the synthetic benchmark built by means of the Forest Fire model. Charts are drawn by grouping the running times of each algorithm according to properties of the target and query graphs. Number of target vertices and target density do not affect trends in computational requirements of the algorithms. Dependencies emerge for number of distinct target labels and number of query vertices. onsistently wins, often with a low p value

generating them with high-density values requires large computational resources. That lower edge density is the reason the graphs show a decrease in running time as the number of nodes in the target graph increases.

With respect to the Barabási–Albert model, as an average speed-up of a factor of 4185 and 96 compared with RI-DS and DAF, respectively. For the Erdos model, the speed-up is a factor of 7109 and 864, and for the Forest Fire model, it is 1677 and 271, in comparison with RI-DS and DAF, respectively.

4.5 Induced subgraph isomorphism

Lastly, we compared the performance of nd vF3 in searching for induced subgraphs. We used the collection of fully labelled PPI networks described in Sect. 4.1.

Figure 14 shows the results of the comparison. The VF3 strategy exploits feasibility rules based on the topology of the query and is less dependent on the arrangement of labels. By contrast, the euristic exploits labelled paths. It is relevant to observe that the overhead due to the construction of edge domains does not capture the topological properties of the query that can be exploited in the case of induced subgraphs. Specifically, oes not capture the *negative* edges that are constraints for the induced subgraph isomorphism. For these reasons, the VF3 strategy is better when few vertex labels are present in the target graph, but VF3's advantage is reduced for increasing numbers of labels. Compared to vF3, for large numbers of labels, as a speed-up as high as a factor of 29.

4.6 Comparison with multigraph-focused approaches

A multigraph is a pair $G = (V, E)$ in which V is the set of vertices and E is a multiset of pairs of vertices. As for simple graphs, multigraphs can be equipped with vertex and edge labelling functions. Switching from multigraphs to graphs entails merging multiple edges between the same pair of vertices into a single edge. If labels are assigned to an edge, then the co-domain of the labelling function for edges switches from a single element of a set of labels to a subset of such a set of elements. A similar switch can be done for graphs in which multiple labels are assigned to a single

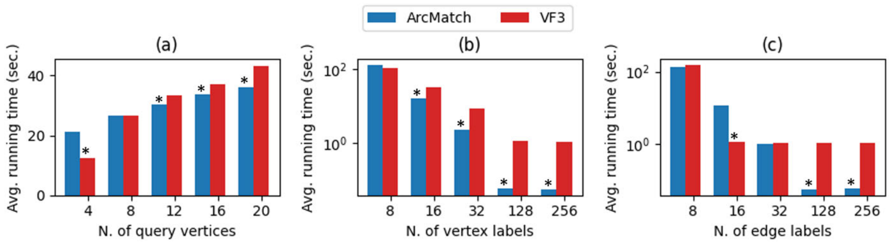

Fig. 14 Comparison of ith VF3 when searching for all the induced subgraph isomorphism occurrences of queries over the Protein–Protein Interaction benchmark. Average running times are grouped by: **a** the number of query vertices along with varying numbers of vertex and edge labels; **b** the number of vertex labels along with varying numbers of query vertices and edge labels. **c** the number of edge labels along with varying numbers of query vertices and vertex labels. When the number of vertex labels is greater than 8 or the number of query vertices is greater than 8, s faster, thanks to the effectiveness of its reduction techniques

vertex. In this case, a trivial adaptation of the proposed approach can be obtained by implementing a specific comparator function for the set assigned to vertices and edges. The behavior of such a comparator depends on the specific application. It can be a set inclusion operator or any other set-theoretical comparison. This approach provides a solution for applying the proposed methodology to multigraphs, or to the more generic structures called property graphs (Comyn-Wattiau and Akoka 2017). However, specific indexing techniques for managing such type of graphs, besides the strategic choices that drive the search process, may be developed in order to speed the search process up. As a proof of potential methodological extension of o this type of graphs, we compared the performance of nd Neo4j (version 4.4.29) in searching for subgraph isomorphisms. We used the collection of the fully labelled Protein–Protein Interaction networks described in Sect. 4.1. This type of graph does not contain multiple edges between vertices, so the retrieved matches of Neo4j equal the retrieved substructures of A Neo4j server instance was installed on the testing machine. For each SubGI instance, the target graph was converted into the textual input format of Neo4j and loaded into a blank Neo4j server instance. Labels are attached to vertices and edges as attributes of such elements. The query graph was translated into a Neo4j query according to the Cypher query language and passed to Neo4j through the Neo4j *cypher-shell* command. The running time of the experiment includes the loading of the target graph into the Neo4j instance and the search for the matches, but it does not contain the time needed for the translation of the two graphs.

Neo4j is a NoSQL database management system which includes several capabilities besides the core subgraph search. It manages vertices and edges having multiple labels, and constructs indexing data structures for their efficient scanning once a query is run. This means that, on loading a target graph into the server instance, Neo4j spends time in an indexing phase, which hopefully will reduce the querying time. When such indexing structures are employed, it is reasonable to share the indexing cost across multiple queries. However, indexing data structures are not the goal of the proposed study. In addition, Neo4j is based on join operations that are performed among elements of the query. This means that a query graph is split into a series of filtering and join operations. The filtering operation corresponds to the

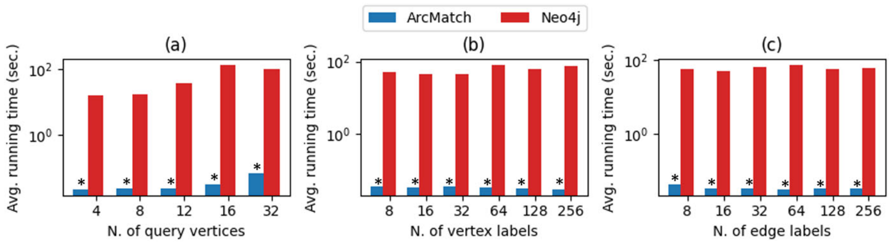

Fig. 15 Comparison of ith Neo4j when searching all the induced subgraph isomorphism occurrences of queries over the PPI (Protein–Protein Interaction) benchmark. Average running times are grouped by: **a** the number of query vertices along with varying numbers of vertex and edge labels; **b** the number of vertex labels along with varying numbers of query vertices and edge labels. **c** the number of edge labels along with varying numbers of query vertices and vertex labels

search for target vertices or edges having a given label equal to an element of the query graph. Then, such a set of substructures is joined to the set of temporary structures that have been retrieved by the previous series of join operations. The sequence of filtering and join operations is dynamically chosen according to predictions on the size of the temporary sets of substructures. Figure 15 shows the results of the comparison. Neo4j has an average running time which is close to 100 s. On average (not shown in the figure) Neo4j requires 4 s to load the target graph (thus, indexing it) and to internally parse the query graph (which is given according to the Cypher query language).

5 Conclusions

In the context of subgraph isomorphism, where a query graph is searched within a target graph, the domain of a query element such as a vertex is the set of compatible elements in the target graph. Recently, the concept of edge domains has been the focus of increasing interest (Han et al. 2019; Kim et al. 2021), especially when they are combined with vertex domains into a unified data structure.

Using such a data structure, ntroduces a new technique for reducing domains which exploits the topological relationship of domains, specifically, their correspondence to paths of the query graph. This approach generalizes already existing techniques for reducing domains, such as arc consistency, extending those techniques from taking into account constraints coming from single edges to structural constraints arising from paths of the query graph.

Tests on real networks have shown that combining these new approaches with existing techniques effectively reduces running times when the number of distinct labels on vertices and/or edges of the target graphs is high. A slight variant of called *ArcMatch-lt*, performs better when few matches have to be retrieved.

The authors of state-of-the-art algorithms have studied the scalability of their algorithms with respect to the number of target vertices (Aparo et al. 2019; McCreesh et al. 2018; Carletti et al. 2013, 2017a, 2020; Han et al. 2019). By contrast, our approach is most sensitive to the number of target labels and the query size.

We have seen that there is a trade-off between computing constraints (which may take substantial time) and matching time (which is reduced by constraints). In this study, we propose two different versions of the proposed approach. When only a limited number of matches is required instead of the complete set of query embeddings, it is better to economize on the constraint enforcement. In future work, we aim to further investigate possible improvements of the approach, in particularly in searching for a way to auto-tune the selection for the best reduction configuration according to the required output and the properties of the input graphs. In addition, we also believe that the novel path-based reduction technique could be applied in CSP solvers to improve their performance.

Appendix A: Detailed comparison of the techniques involved in *ArcMatch*

In what follows, we evaluate the advantage that each feature gives to A baseline version of the methodology is included: it builds vertex and edge domains, reduces vertex domains with a single run of arc consistency, computes the variable ordering as in Bonnici et al. (2013) (measures N_1, N_2 and N_3 are used), but it does not apply further techniques. In the baseline version, the search process is not performed on top of the domain graph as described in Sect. 3.3. In this way, the baseline is very close to the RI-DS algorithm with the difference that it computes the initial content of edge domains. The baseline version is then augmented with the following features in order to determine both the advantage/disadvantage of each feature individually and in combination.

The features are:

- *NS*: the variable ordering is performed by also using measures N_4 and N_5 (see Sect. 3.2).
- *ED*: the search process is driven by the domain graph. Thus, candidates are extracted from edge domains (see Sect. 3.3). However, no dynamic parent selection is performed. Parents are chosen statically before the search process begins as in Bonnici et al. (2013).
- *DY*: as for ED but dynamic parent selection is enabled (see Sect. 3.3).
- *VC*: arc consistency is applied to vertex domains until convergence (see Sect. 2.3.2).
- *RE*: apply path reduction procedure (`PathReduction`) as it is described in Sect. 3.1, but disable the call to the function `RefineDomains`, which reduces edge domains.
- *RR*: reduce edge domains by also enabling `RefineDomains`
- *PV*: enable the management of peripheral vertices by using the procedure described in Sect. 3.2.1.

16 versions of ere obtained by combining the different features (NS, ED, DY, VC, RE, RR and PV). Table 5 shows the number of queries that were finished within the 600 s timeout by the tested solutions.

Results are grouped by number of target vertex labels (from 1 to 1024). In that grouping, graphs having more than one edge label are discarded. In subsequent tests, only instances regarding target graphs having one vertex label are taken into account, and they are grouped by number of edge labels (from 1 to 1024). The last column reports the total number of finished instances, as a function of the number of labels that are assigned to vertices or edges. Each reported grouping is composed of a total of 250 instances except for the *all* grouping that includes all the corresponding instances. For each column, the solution that solved the maximum number of instances is highlighted in bold.

The table shows that RI-DS and Glasgow are always outperformed by at least one olution. As expected, all the approaches are more effective with increasing numbers of labels, regardless of whether the labels are assigned to vertices or edges. The reason is that increasing the number of labels decreases the number of target vertices (edges) that are compatible with a given query vertex (edge). With more that 32 labels, almost every instance is solved by all the approaches within the timeout. 7 is the configuration that performs best overall. It finished a total of 8761 instances. Such a configuration does not go to convergence when reducing vertex domains. Thus, it postpones a more accurate filtering of domains to path-based reduction. This result shows the advantages of using vertex and edges domains but without forcing their reduction to convergence.

Similarly, when there are no labels or, equivalently, the same label for all vertices and the same label for all edges, solutions that do not exploit complex techniques, such as domain reduction, enjoy substantial advantages (see for example 1). When there are a small number of labels, the filtering power of reduction techniques is minimal, so their benefits are not worth their computational costs.

When the number of vertex/edge candidates is high, a specialized procedure for managing peripheral query vertices is worthless. The technique can only exploit domains that do not overlap, but, with a few labels, domains tend to have non-null intersections. Thus, overriding the general ordering strategy by postponing the processing of peripheral vertices is counterproductive.

The second configuration that performs well overall is 13. It does not reduce edge domains but it exploits them during the searching process. Moreover, it reduces vertex domains until convergence to make up for the reduced candidate filtering power.

Table 6 reports the running time of the compared solutions. Average (and standard deviation) on running times for different groupings are reported. Results are grouped by vertex labels by taking into account only instances with one edge label. Instances with one label have an average running time close to 600 because a timeout of 600 s is applied and most of the instances reached the timeout. The average running time correlates with the number of finished instances (see Table 8). With 128 and more labels, the instances are solved in less than 1 s. In Table 9, all instances are grouped by the number of query vertices, independently of from the number of target labels. The table shows that the running time depends on the number of query vertices. For

Table 5 Number of finished instances for the PPI benchmark, by varying number of vertex and edge labels, on searching for all the occurrences

Algorithm	eatures							Vertex labels (1 edge label)							Edge labels (1 vertex label)							All
	NS	ED	DY	VC	RE	RR	PV	1	16	32	128	256	1024	All	1	16	32	128	256	1024	All	
1								**46**	128	210	250	250	250	1134	**46**	46	48	177	211	244	772	8107
2		✓						39	129	211	250	250	250	1129	39	95	199	249	250	250	1082	8422
3		✓	✓					**46**	143	220	250	250	250	1159	**46**	**244**	**248**	250	250	250	**1288**	8651
4	✓	✓	✓					45	140	218	250	250	250	1153	45	**244**	**248**	250	250	250	1287	8645
5	✓	✓	✓		✓			34	140	219	250	250	250	1143	34	225	242	250	250	250	1251	8610
6	✓	✓	✓				✓	20	186	237	250	250	250	1193	20	218	227	246	247	247	1205	8628
7	✓	✓	✓		✓		✓	42	**245**	**250**	250	250	250	**1287**	42	230	244	250	250	250	1266	**8761**
8				✓				**46**	131	212	250	250	250	1,139	**46**	54	189	250	250	250	1039	8382
9		✓		✓				39	132	217	250	250	250	1138	39	119	234	250	250	250	1142	8491
10		✓	✓	✓				**46**	143	220	250	250	250	1159	**46**	**244**	**248**	250	250	250	**1288**	8651
11	✓	✓	✓	✓				45	140	219	250	250	250	1154	45	**244**	**248**	250	250	250	1287	8646
12	✓	✓	✓	✓	✓			33	140	219	250	250	250	1142	33	229	247	250	250	250	1259	8618
13	✓	✓	✓	✓			✓	19	187	239	250	250	250	1195	19	226	243	250	250	250	1238	8664
14	✓	✓	✓	✓	✓		✓	12	186	241	250	250	250	1189	12	215	240	250	250	250	1217	8644
15	✓	✓	✓	✓	✓	✓		33	140	219	250	250	250	1142	33	230	**248**	250	250	250	1261	8620
16	✓	✓	✓	✓	✓	✓	✓	12	186	241	250	250	250	1189	12	218	242	250	250	250	1222	8649

Table 6 Average running times (and standard deviation) over varying number of vertex labels on the PPI benchmark when searching for all the occurrences

Algorithm	eatures							Vertex labels (1 edge label)					
	NS	ED	DY	VC	RE	RR	PV	1	16	32	128	256	1024
1								504.61 (207.37)	308.61 (292.66)	113.61 (226.97)	0.70 (5.19)	0.03 (0.02)	0.02 (0.02)
2		✓						537.77 (152.51)	309.28 (291.17)	103.03 (219.34)	0.05 (0.24)	0.02 (0.02)	0.02 (0.01)
3		✓	✓					503.02 (208.77)	273.83 (291.06)	81.66 (197.48)	0.11 (1.11)	0.02 (0.02)	0.02 (0.02)
4	✓	✓	✓					505.19 (206.59)	278.31 (290.58)	88.23 (203.57)	0.23 (3.21)	0.02 (0.02)	0.02 (0.02)
5	✓	✓	✓		✓			563.60 (107.77)	278.90 (290.47)	88.11 (203.79)	0.08 (0.61)	0.03 (0.06)	0.03 (0.05)
6	✓	✓	✓				✓	559.47 (144.67)	167.44 (261.26)	39.85 (139.82)	0.02 (0.02)	0.02 (0.01)	0.02 (0.02)
7	✓	✓	✓		✓		✓	549.25 (124.52)	160.60 (89.73)	20.79 (58.50)	0.03 (0.07)	0.03 (0.06)	0.03 (0.05)
8				✓				504.69 (207.34)	303.95 (292.20)	102.71 (218.46)	0.34 (3.58)	0.02 (0.02)	0.02 (0.02)
9		✓		✓				537.98 (152.17)	301.72 (290.69)	90.28 (205.76)	0.04 (0.17)	0.02 (0.02)	0.02 (0.02)
10		✓	✓	✓				503.64 (208.41)	273.86 (291.26)	81.67 (197.75)	0.11 (1.12)	0.02 (0.02)	0.02 (0.02)
11	✓	✓	✓	✓				506.10 (206.15)	278.57 (290.71)	87.60 (202.71)	0.07 (0.67)	0.02 (0.02)	0.02 (0.02)
12	✓	✓	✓	✓	✓			563.99 (106.86)	278.97 (290.43)	88.37 (204.12)	0.08 (0.61)	0.03 (0.06)	0.03 (0.05)
13	✓	✓	✓	✓			✓	559.97 (144.66)	163.34 (259.42)	31.67 (126.59)	0.02 (0.01)	0.02 (0.01)	0.02 (0.01)
14	✓	✓	✓	✓	✓		✓	587.13 (64.90)	163.99 (260.70)	28.13 (117.27)	0.03 (0.06)	0.03 (0.06)	0.03 (0.05)
15	✓	✓	✓	✓	✓	✓		563.47 (106.68)	278.98 (290.55)	87.92 (203.59)	0.09 (0.69)	0.03 (0.06)	0.03 (0.06)
16	✓	✓	✓	✓	✓	✓	✓	587.31 (64.85)	163.70 (260.52)	29.02 (118.25)	0.03 (0.06)	0.03 (0.06)	0.03 (0.05)

Table 7 Average running times (and standard deviation) over the fully labelled PPI benchmark extracting only instances where target graphs have one edge label, when searching for all the occurrences

Algorithm	eatures							Query vertices					All
	NS	ED	DY	VC	RE	RR	PV	4	8	12	16	32	
1								20.53 (86.53)	111.31 (231.98)	170.27 (264.51)	208.89 (283.14)	261.99 (295.58)	154.60 (257.92)
2		✓						48.16 (134.30)	111.87 (231.61)	168.43 (263.41)	205.84 (282.51)	257.51 (295.67)	158.36 (258.45)
3		✓	✓					19.20 (81.77)	105.62 (227.25)	157.68 (259.83)	185.95 (273.17)	247.10 (293.65)	143.11 (251.22)
4	✓	✓	✓					21.01 (87.67)	105.80 (227.33)	160.73 (261.29)	188.67 (273.45)	250.46 (293.90)	145.33 (252.28)
5	✓	✓	✓		✓			69.69 (171.99)	105.95 (227.31)	161.02 (260.99)	189.15 (273.89)	249.82 (294.26)	155.13 (256.92)
6	✓	✓	✓				✓	66.24 (184.19)	100.23 (223.88)	111.80 (232.41)	132.03 (242.21)	228.72 (288.61)	127.80 (242.61)
7	✓	✓	✓		✓		✓	62.44 (157.59)	95.40 (216.02)	100.16 (223.91)	100.84 (223.79)	112.99 (232.94)	94.36 (212.99)
8				✓				20.59 (86.83)	111.37 (232.09)	169.63 (264.31)	204.66 (280.68)	253.52 (295.63)	151.95 (256.51)
9		✓		✓				48.33 (134.63)	111.87 (231.63)	167.46 (263.35)	199.11 (278.89)	248.27 (293.71)	155.01 (256.29)
10		✓	✓	✓				19.71 (84.00)	105.60 (227.24)	157.83 (259.93)	186.02 (273.35)	246.94 (293.77)	143.22 (251.40)
11	✓	✓	✓	✓				21.77 (90.96)	105.81 (227.34)	160.93 (260.98)	188.81 (273.56)	249.68 (294.19)	145.40 (252.41)
12	✓	✓	✓	✓	✓			70.01 (172.46)	105.95 (227.30)	161.07 (260.97)	189.26 (273.91)	249.93 (294.35)	155.25 (256.99)
13	✓	✓	✓	✓			✓	66.66 (185.32)	100.20 (223.90)	112.34 (234.02)	128.51 (239.17)	221.48 (287.61)	125.84 (241.63)
14	✓	✓	✓	✓	✓		✓	89.30 (207.19)	100.25 (223.88)	112.37 (234.01)	128.77 (240.27)	218.76 (286.33)	129.89 (243.92)
15	✓	✓	✓	✓	✓	✓		69.58 (171.11)	106.25 (227.39)	160.82 (260.96)	189.04 (273.95)	249.74 (294.28)	155.09 (256.81)
16	✓	✓	✓	✓	✓	✓	✓	89.45 (207.53)	100.25 (223.88)	112.36 (234.02)	128.51 (240.04)	219.54 (286.18)	130.02 (243.96)

Results are grouped by the number of query vertices

Table 8 Number of finished instances for the vertex-labelled PPI benchmark, by varying the number of vertex labels and the number of query vertices, on searching for the first 100k occurrences

Algorithm	eatures							Vertex labels						Query vertices					All
	NS	ED	DY	VC	RE	RR	PV	1	16	32	128	256	1024	4	8	12	16	32	
1								241	238	248	250	250	250	300	300	299	299	279	1477
2		✓						241	238	248	250	250	250	300	300	299	299	279	1477
3		✓	✓					238	246	249	250	250	250	300	300	300	300	283	1483
4	✓	✓	✓					241	245	249	250	250	250	300	300	300	300	285	1485
5	✓	✓	✓		✓			39	243	249	250	250	250	282	257	250	250	242	1281
6	✓	✓	✓				✓	203	240	249	250	250	250	298	293	290	287	274	1442
7	✓	✓	✓		✓		✓	34	240	249	250	250	250	279	255	250	249	240	1273
8				✓				245	243	249	250	250	250	300	300	299	300	288	1487
9		✓		✓				241	242	249	250	250	250	300	300	299	299	284	1482
10		✓	✓	✓				238	246	249	250	250	250	300	300	300	300	283	1483
11	✓	✓	✓	✓				242	244	249	250	250	250	300	300	300	300	285	1485
12	✓	✓	✓	✓	✓			38	244	249	250	250	250	282	256	250	250	243	1281
13	✓	✓	✓	✓			✓	207	244	249	250	250	250	298	293	290	292	277	1450
14	✓	✓	✓	✓	✓		✓	38	241	249	250	250	250	282	256	250	249	241	1278
15	✓	✓	✓	✓	✓	✓		41	243	249	250	250	250	284	257	250	250	242	1283
16	✓	✓	✓	✓	✓	✓	✓	39	241	249	250	250	250	283	256	250	249	241	1279

Table 9 Average running times over the fully labelled PPI benchmark and over varying number of query vertices, when searching for all the occurrences

Algorithm	eatures							Query vertices					All
	NS	ED	DY	VC	RE	RR	PV	4	8	12	16	32	
1								7.87 (53.92)	59.82 (178.46)	69.43 (190.07)	77.36 (199.87)	96.16 (218.69)	62.13 (180.54)
2		✓						8.13 (57.61)	26.73 (120.94)	47.21 (158.54)	57.30 (174.94)	67.20 (188.38)	41.31 (149.26)
3		✓	✓					3.27 (34.09)	18.86 (103.44)	27.11 (122.61)	32.20 (133.17)	41.85 (151.28)	24.66 (116.91)
4	✓	✓	✓					3.57 (36.58)	18.89 (103.49)	27.62 (123.67)	32.65 (133.79)	42.37 (152.03)	25.02 (117.64)
5	✓	✓	✓		✓			11.94 (74.79)	20.51 (106.34)	29.61 (126.61)	35.54 (137.98)	48.43 (160.38)	29.20 (125.25)
6	✓	✓	✓				✓	17.12 (96.58)	23.90 (115.99)	22.13 (111.15)	26.98 (121.13)	42.37 (151.39)	26.50 (120.88)
7	✓	✓	✓		✓		✓	10.75 (68.35)	17.69 (98.16)	18.96 (102.63)	20.25 (105.40)	25.51 (117.17)	18.63 (99.77)
8				✓				6.75 (50.72)	43.20 (154.62)	49.81 (163.57)	53.42 (169.44)	61.86 (181.80)	43.01 (152.82)
9		✓		✓				8.14 (57.76)	25.43 (118.15)	40.70 (146.63)	51.10 (166.19)	56.41 (174.46)	36.36 (140.26)
10		✓	✓	✓				3.34 (35.02)	18.81 (103.44)	27.08 (122.68)	32.14 (133.26)	41.59 (151.19)	24.59 (116.96)
11	✓	✓	✓	✓				3.69 (37.96)	18.84 (103.50)	27.60 (123.61)	32.60 (133.87)	42.05 (151.95)	24.96 (117.69)
12	✓	✓	✓	✓	✓			11.94 (75.01)	20.27 (106.25)	29.12 (126.53)	34.71 (137.42)	44.51 (155.09)	28.11 (123.67)
13	✓	✓	✓	✓			✓	13.97 (87.55)	20.70 (108.77)	20.62 (108.34)	24.05 (114.64)	38.47 (145.50)	23.56 (114.76)
14	✓	✓	✓	✓	✓		✓	17.89 (97.62)	22.55 (112.85)	21.85 (111.18)	25.47 (118.08)	39.19 (145.81)	25.39 (118.37)
15	✓	✓	✓	✓	✓	✓		11.86 (74.45)	20.20 (106.28)	28.82 (126.14)	34.00 (136.68)	43.64 (154.64)	27.71 (123.22)
16	✓	✓	✓	✓	✓	✓	✓	17.90 (97.77)	21.80 (110.83)	21.13 (109.76)	24.86 (117.11)	38.85 (146.12)	24.91 (117.64)

Table 10 Average running times (and standard deviation) over the vertex-labelled PPI benchmark, by varying number of vertex labels in the target graph, on searching for the first 100k occurrences

Algorithm	eatures							Vertex labels						All
	NS	ED	DY	VC	RE	RR	PV	1	16	32	128	256	1024	
1								43.47(112.01)	30.52(130.56)	4.85(53.55)	0.02(0.02)	0.02(0.01)	0.02(0.01)	13.15(75.46)
2		✓						43.92(112.09)	30.53(130.58)	4.85(53.55)	0.02(0.02)	0.02(0.02)	0.02(0.02)	13.23(75.51)
3		✓	✓					49.94(127.63)	10.48(75.67)	2.44(37.94)	0.02(0.01)	0.02(0.01)	0.02(0.01)	10.49(64.97)
4	✓	✓	✓					44.25(112.39)	13.29(84.87)	2.44(37.95)	0.02(0.02)	0.02(0.02)	0.02(0.02)	10.01(61.57)
5	✓	✓	✓		✓			557.68(114.28)	21.89(106.01)	3.08(38.25)	0.03(0.06)	0.03(0.06)	0.03(0.05)	97.12(216.31)
6	✓	✓	✓				✓	147.81(228.20)	27.03(120.13)	6.24(50.59)	0.02(0.01)	0.02(0.01)	0.02(0.01)	30.19(119.72)
7	✓	✓	✓		✓		✓	561.77(111.81)	31.46(126.85)	3.57(39.05)	0.03(0.06)	0.03(0.05)	0.03(0.05)	99.48(218.86)
8				✓				32.41(85.70)	21.03(104.92)	3.49(40.83)	0.03(0.16)	0.02(0.02)	0.02(0.01)	9.50(59.04)
9		✓		✓				43.92(112.12)	22.31(110.70)	4.30(44.56)	0.02(0.02)	0.02(0.02)	0.02(0.02)	11.77(68.72)
10		✓	✓	✓				50.29(127.72)	10.49(75.69)	2.44(37.95)	0.02(0.01)	0.02(0.01)	0.02(0.01)	10.55(65.04)
11	✓	✓	✓	✓				43.11(107.90)	15.44(92.46)	2.43(37.95)	0.02(0.01)	0.02(0.01)	0.02(0.01)	10.17(61.97)
12	✓	✓	✓	✓	✓			559.34(110.71)	19.34(97.17)	2.88(38.08)	0.03(0.06)	0.03(0.06)	0.03(0.04)	96.94(216.06)
13	✓	✓	✓	✓			✓	143.84(224.04)	17.03(94.69)	2.47(37.95)	0.02(0.01)	0.02(0.01)	0.02(0.01)	27.23(113.24)
14	✓	✓	✓	✓	✓		✓	555.04(117.78)	28.28(119.65)	3.38(38.90)	0.03(0.06)	0.03(0.06)	0.03(0.05)	97.80(216.51)
15	✓	✓	✓	✓	✓	✓		555.32(114.75)	20.33(101.55)	2.77(38.02)	0.03(0.06)	0.03(0.06)	0.03(0.05)	96.42(215.26)
16	✓	✓	✓	✓	✓	✓	✓	555.11(118.20)	27.16(118.00)	2.77(38.02)	0.03(0.06)	0.03(0.06)	0.03(0.05)	97.52(216.50)

some algorithms, there is an order of magnitude increase from 4 to 32 query vertices (for example 1 and 10), while other configurations just double their time. The predominance of 7 is very clear. Such a configuration is always the best solution except for instances with only one label and for queries with just 4 vertices. These types of instances are too simple to benefit from the advantages of sophisticated filtering techniques, such as edge domain reduction. In fact, configurations that do not reduce edge domains are the fastest ones on queries with four vertices. Similar considerations apply when no edge labels are present, as it is shown in Table 7 (Tables 8, 9).

Lastly, we evaluated the significance of the features in searching for the first 100k matches. Results are shown in Table 10. Combinations of features that do not exploit complex reduction techniques outperform more sophisticated approaches. For this number of matches, 8, which exploits arc-consistency only, is the fastest solution, and 7 is slower. More generally, combinations that exploit edge domains enjoy significant advantages when there are no labels. When there are many labels, gaps between the studied solutions shrink. From 32 labels on, all solutions have compatible running times.

In conclusion, we select 7 as the main version of the proposed approach and propose 13 as a light version that can be used, for example, for finding the first 100k matches.

Appendix B: Relation between arc consistency and path-based reduction

In the constraint satisfaction field, the concept of arc consistency is extended to *path consistency* (Dechter et al. 2003). The reader might think there is a correspondence between the path-based reduction technique proposed in this study and path consistency. They differ, however. Path consistency is defined only in terms of the CSP (Constraint Satisfaction Problem) and aims to verify the entire set of rules relating to two or more variables. Translated into a graph theory formulation, such a concept will check for the correspondence of the subgraph between a given query vertex and a target vertex. In our path-based reduction, we are interested in verifying only that there is a corresponding path starting from the two vertices. No edges (rules) between vertices (variables) are examined outside of the path. Path consistency also differs from the reduction presented in Han et al. (2019), since DAF checks for non-induced substructures, rather than simple paths. Thus, DAF requires higher computational costs, it is potentially more effective in reducing domains but there is no guarantee of such performance.

Given two query vertices, v_q and u_q, connected by the edge (v_q, u_q), arc consistency verifies that each target vertex $v_t \in D(v_q)$ is consistent with respect to such a constraint. Thus, v_t must be connected with at least one vertex in $D(u_q)$. If v_t makes $D(v_q)$ inconsistent, than v_t is removed from $D(v_q)$, namely $D(v_q)$ is reduced. The removal may have a cascading effect on the other domains. Thus the reduction of $D(v_q)$ propagates to the other domains. The propagation can be performed in two different ways: i) in conjunction with the domain reduction; (ii) or it can be performed by multiple runs of arc-consistency. In the first case, each removal is systematically propagated to the other domains. In the second, case, each run is performed over the result of the previous one, and the process is repeated until convergence. In any case, the two approaches produce the same effects. In the first solution is adopted, and it is implemented by the procedure `RefineDomains`.

`PathReduction` generalizes the concept of arc consistency by imposing consistency not at the edge level but at the path level. For each path ω such that $\omega[1] = v_q$, each target vertex in $D(v_q)$ must be the starting vertex of a corresponding path supported by the current state of the domains. Thus, it is trivial to show that `PathReduction` is equivalent to arc-consistency when $lp = 2$, namely when paths composed of only one edge are taken into account.

Note that the propagation is an additional feature with respect to one simple application of arc-consistency. In some situations, propagation is a costly operation whose benefits are not worth the cost. Given a query graph $G_q = (V_q, E_q)$, the cost of each run of arc-consistency is proportional to $|E_q|$ because edges are scanned. By contrast, the cost of `PathReduction` is proportional to the number of query paths, which may grow exponentially with respect to $|E_q|$.

A question arises: if no propagation is performed, is `PathReduction` more powerful than arc-consistency?

In what follows, we refer to $r_{v_q}^{v_t}$ as the operation which removes the target vertex v_t from $D(v_q)$. Given a query graph G_q and a target graph G_t, $R_{AC}(G_q, G_t)$ represents the set of removal operations that are performed by a single run of arc-consistency over the initial domains. $R_{AC}^{i}(G_q, G_t)$ represents the set of removal operations obtained in i runs of arc-consistency. Lastly, $R_{FC}(G_q, G_t, lp)$ represents the set of removal operations that are performed by a single run of `PathReduction` without propagation, namely by discarding the call to the procedure `RefineDomains`.

Theorem 6 Given a query graph G_q and a target graph G_t, $R_{AC}(G_q, G_t) \subseteq R_{FC}(G_q, G_t, lp)$ if $lp > 2$.

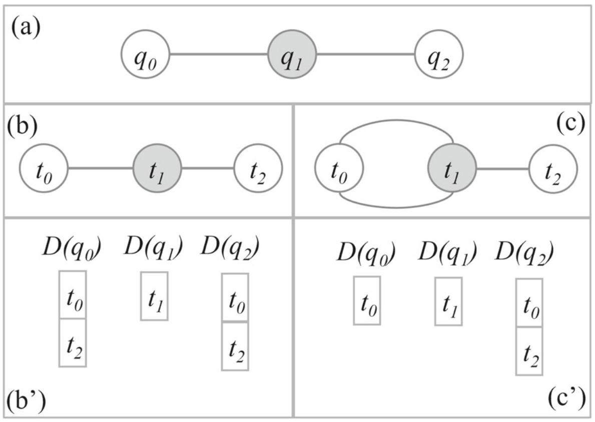

Fig. 16 Example of possible differences between arc-consistency and `PathReduction`

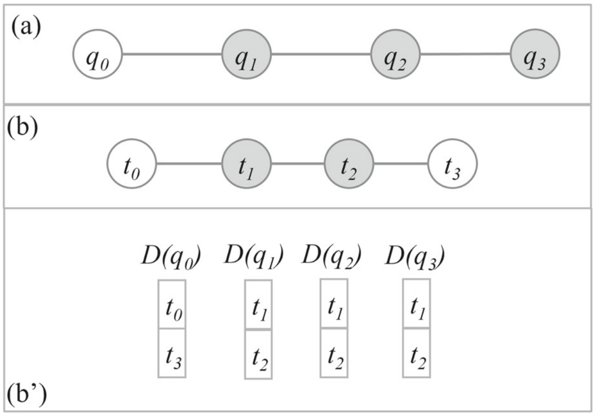

Fig. 17 Example of possible differences between arc-consistency and `PathReduction`

Proof If $lp = 2$, $R_{FC}(G_q, G_t, 2)$ equals $R_{AC}(G_q, G_t)$, because the two techniques both scan for edges and produce the same result (Figs. 16, 17).
Thus, if no propagation is applied, arc consistency is equivalent to `PathReduction` with $lp = 2$, namely $R_{AC}(G_q, G_t) \subseteq R_{FC}(G_q, G_t, 2)$. However, Theorem 1 states that $R_{FC}(G_q, G_t, lp_1) \subseteq R_{FC}(G_q, G_t, lp_2)$ with $lp_1 < lp_2$. Thus, $R_{AC}(G_q, G_t) \subseteq R_{FC}(G_q, G_t, 2) \subseteq R_{FC}(G_q, G_t, lp > 2)$. □

Appendix C: Statistical significance

We tested the statistical significance of the difference between nd the competitors. To do so, we applied the empirical non-parametric paired test described in Katari et al. (2021). Given a set of n queries, we construct two vectors, v_1 and v_2, of length n reporting the corresponding running time of two algorithms. The running time corresponding to the ith query is reported at position i of both vectors. Let's suppose v_1 to be the algorithm with the highest average value. We want to test whether algorithm 2 (which has the lowest average value) is statistically significantly faster. Let $d = \sum_{i=1}^{n}(v_1[i] - v_2[i])$, where $v[i]$ is the ith value of v. For a given number N of iterations, we build two vectors v'_1 and v'_2 such that, for each position $1 \le i \le n$, $v'_1[i] = v_1[i]$ and $v'_2[i] = v_2[i]$ with probability 0.5, and with probability 0.5 $v'_1[i] = v_2[i]$ and $v'_2[i] = v_1[i]$. Namely, with probability 0.5 we exchange the values of the two vectors. Thus, we count how many times $\sum_{i=1}^{n}(v'_1[i] - v'_2[i]) > d$. We chose N equal to 1000. The resultant p value is the count divided by N. (If the count value is 0, then we say that the p value is less than $1/N$ (1/1000 in our case).) If the p value is small, then it's unlikely that algorithm 2 was shown to be faster than algorithm 1 by chance.

Appendix D: Memory scalability

We computed the average memory footprints of the compared algorithms on the three synthetic benchmarks in order to analyse the scalability of the approaches. The analysis was conducted by varying the number of target vertices, the number of vertex labels and the number of query vertices. Results for the Barabási–Albert, Erdos and Forest Fire models are shown in Figs. 18, 19 and 20, respectively.

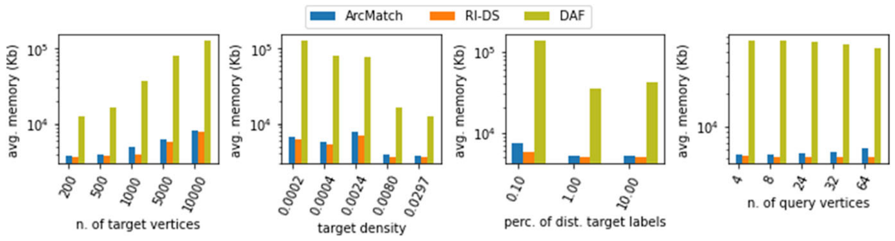

Fig. 18 Memory scalability analysis for the synthetic benchmark built by means of the Barabási–Albert model. Charts are drawn by grouping the memory footprints of each algorithm according to the properties of the target and query graphs

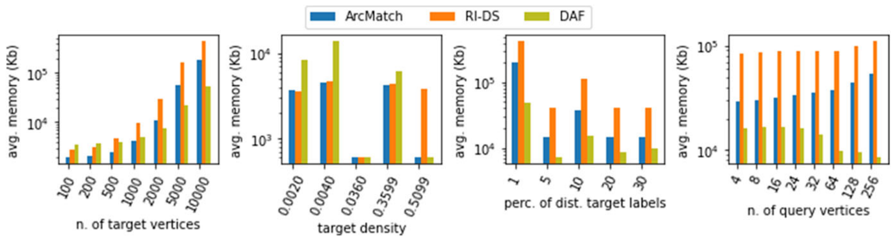

Fig. 19 Memory scalability analysis for the synthetic benchmark built by means of the Erdos model. Charts are drawn by grouping the memory footprints of each algorithm according to the properties of the target and query graphs

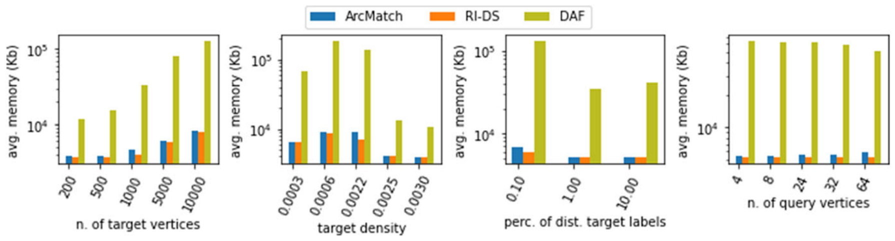

Fig. 20 Memory scalability analysis for the synthetic benchmark built by means of the Forest Fire model. Charts are drawn by grouping the memory footprints of each algorithm according to the properties of the target and query graphs

Funding Open access funding provided by Università degli Studi di Parma within the CRUI-CARE Agreement.

References

Aparo A, Bonnici V, Micale G, Ferro A, Shasha D, Pulvirenti A, Giugno R (2019) Fast subgraph matching strategies based on pattern-only heuristics. Interdiscip. Sci.: Comput. Life Sci. 11(1):21–32

Archibald B, Burns K, McCreesh C, Sevegnani M (2021) Practical bigraphs via subgraph isomorphism. In: 27th international conference on principles and practice of constraint programming (CP 2021). Schloss Dagstuhl-Leibniz-Zentrum für Informatik

Avellaneda F, Alikacem E-H, Jaafar F (2019) Using attack pattern for cyber attack attribution. In: 2019 International conference on cybersecurity (ICoCSec). IEEE, pp 1–6

Balaban AT (1985) Applications of graph theory in chemistry. J Chem Inf Comput Sci 25(3):334–343

Barabási A-L, Albert R (1999) Emergence of scaling in random networks. science 286(5439):509–512

Bi F, Chang L, Lin X, Qin L, Zhang W (2016) Efficient subgraph matching by postponing cartesian products. In: Proceedings of the 2016 international conference on management of data, pp 1199–1214

Bing R, Yuan G, Zhu M, Meng F, Ma H, Qiao S (2023) Heterogeneous graph neural networks analysis: a survey of techniques, evaluations and applications. Artif Intell Rev 56(8):8003–8042

Bonnici V, Giugno R (2017) On the variable ordering in subgraph isomorphism algorithms. IEEE/ACM Trans Comput Biol Bioinform 14(1):193–203

Bonnici V, Ferro A, Giugno R, Pulvirenti A, Shasha D (2010) Enhancing graph database indexing by suffix tree structure. In: IAPR international conference on pattern recognition in bioinformatics. Springer, Berlin, pp 195–203

Bonnici V, Giugno R, Pulvirenti A, Shasha D, Ferro A (2013) A subgraph isomorphism algorithm and its application to biochemical data. BMC Bioinform 14(7):1–13

Cao J, Hall D (2021) Module library development via graph mining. In: Proceedings of the 2021 European conference on computing in construction. University College Dublin, Dublin, pp 285–292

Carletti V, Foggia P, Vento M (2013) Performance comparison of five exact graph matching algorithms on biological databases. In: International conference on image analysis and processing. Springer, Berlin, pp 409–417

Carletti V, Foggia P, Saggese A, Vento M (2017a) Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with vf3. IEEE Trans Pattern Anal Mach Intell 40(4):804–818

Carletti V, Foggia P, Saggese A, Vento M (2017b) Introducing vf3: a new algorithm for subgraph isomorphism. In: International workshop on graph-based representations in pattern recognition. Springer, Berlin, pp 128–139

Carletti V, Foggia P, Greco A, Saggese A, Vento M (2020) Comparing performance of graph matching algorithms on huge graphs. Pattern Recogn Lett 134:58–67

Chaturvedi A, Gupta M, Gupta SK (2018) DPVO: design pattern detection using vertex ordering a case study in jhotdraw with documentation to improve reusability. In: International conference on communication, networks and computing. Springer, Berlin, pp 452–465

Clark NM, Nolan TM, Wang P, Song G, Montes C, Valentine CT, Guo H, Sozzani R, Yin Y, Walley JW (2021) Integrated omics networks reveal the temporal signaling events of brassinosteroid response in Arabidopsis. Nat Commun 12(1):1–13

Comyn-Wattiau I, Akoka J (2017) Model driven reverse engineering of NoSQL property graph databases: the case of Neo4j. In: 2017 IEEE international conference on big data (big data). IEEE, pp 453–458

Cook SA (1971) The complexity of theorem-proving procedures. In: Proceedings of the third annual ACM symposium on theory of computing, pp 151–158

Cordella LP, Foggia P, Sansone C, Vento M (2001) An improved algorithm for matching large graphs. In: 3rd IAPR-TC15 workshop on graph-based representations in pattern recognition, pp 149–159

Dahm N, Bunke H, Caelli T, Gao Y (2015) Efficient subgraph matching using topological node feature constraints. Pattern Recogn 48(2):317–330

Dechter R, Cohen D et al (2003) Constraint processing. Morgan Kaufmann, San Francisco

Erdos P, Rényi A (1959) On random graph. Publ Math 6:290–297

Giugno R, Bonnici V, Bombieri N, Pulvirenti A, Ferro A, Shasha D (2013) Grapes: a software for parallel searching on biological graphs targeting multi-core architectures. PLoS ONE 8(10):76911

Han W-S, Pham M-D, Lee J, Kasperovics R, Yu JX (2011) igraph in action: performance analysis of disk-based graph indexing techniques. In: Proceedings of the 2011 ACM SIGMOD international conference on management of data, pp 1241–1242

Han W-S, Lee J, Lee J-H (2013) Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In: Proceedings of the 2013 ACM SIGMOD international conference on management of data, pp 337–348

Han M, Kim H, Gu G, Park K, Han W-S (2019) Efficient subgraph matching: harmonizing dynamic programming, adaptive matching order, and failing set together. In: Proceedings of the 2019 international conference on management of data, pp 1429–1446

Haralick RM, Elliott GL (1980) Increasing tree search efficiency for constraint satisfaction problems. Artif Intell 14(3):263–313

He H, Singh AK (2008) Graphs-at-a-time: query language and access methods for graph databases. In: Proceedings of the 2008 ACM SIGMOD international conference on management of data, pp 405–418

Hoksza D, Jelínek J (2015) Using neo4j for mining protein graphs: a case study. In: 2015 26th international workshop on database and expert systems applications (DEXA). IEEE, pp 230–234

Huang C-H, Zaenudin E, Tsai JJ, Kurubanjerdjit N, Ng K-L (2022) Network subgraph-based approach for analyzing and comparing molecular networks. PeerJ 10:13137

Katari MS, Tyagi S, Shasha D (2021) Statistics is easy: case studies on real scientific datasets. Synthesis lectures on mathematics and statistics, vol 13, no 3. Springer, Berlin, pp 1–74

Katsarou F, Ntarmos N, Triantafillou P (2015) Performance and scalability of indexed subgraph query processing methods. Proc VLDB Endow 8(12):1566–1577

Katsarou F, Ntarmos N, Triantafillou P (2017) Hybrid algorithms for subgraph pattern queries in graph databases. In: 2017 IEEE international conference on big data (big data). IEEE, pp 656–665

Kim H, Choi Y, Park K, Lin X, Hong S-H, Han W-S (2021) Versatile equivalences: speeding up subgraph query processing and subgraph matching. In: Proceedings of the 2021 international conference on management of data, pp 925–937

Kim H, Choi Y, Park K, Lin X, Hong S-H, Han W-S (2022) Fast subgraph query processing and subgraph matching via static and dynamic equivalences. VLDB J 32:1–26

Lee J, Han W-S, Kasperovics R, Lee J-H (2012) An in-depth comparison of subgraph isomorphism algorithms in graph databases. Proc VLDB Endow 6(2):133–144

Leskovec J, Kleinberg J, Faloutsos C (2005) Graphs over time: densification laws, shrinking diameters and possible explanations. In: Proceedings of the eleventh ACM SIGKDD international conference on knowledge discovery in data mining, pp 177–187

Mackworth AK (1977) Consistency in networks of relations. Artif Intell 8(1):99–118

McCreesh C, Prosser P, Solnon C, Trimble J (2018) When subgraph isomorphism is really hard, and why this matters for graph databases. J Artif Intell Res 61:723–759

McCreesh C, Prosser P, Trimble J (2020) The Glasgow subgraph solver: using constraint programming to tackle hard subgraph isomorphism problem variants. In: International conference on graph transformation. Springer, Berlin, pp 316–324

Milo R, Shen-Orr S, Itzkovitz S, Kashtan N, Chklovskii D, Alon U (2002) Network motifs: simple building blocks of complex networks. Science 298(5594):824–827

Petković M, Ceci M, Pio G, Škrlj B, Kersting K, Džeroski S (2022) Relational tree ensembles and feature rankings. Knowl-Based Syst 251:109254

Piccolboni L, Menon A, Pravadelli G (2017) Efficient control-flow subgraph matching for detecting hardware trojans in RTL models. ACM Trans Embed Comput Syst (TECS) 16(5s):1–19

Pourhabibi T, Ong K-L, Kam BH, Boo YL (2020) Fraud detection: a systematic literature review of graph-based anomaly detection approaches. Decis Support Syst 133:113303

Sakr S, Al-Naymat G (2010) Graph indexing and querying: a review. Int J Web Inf Syst 6:101–120

Shang H, Zhang Y, Lin X, Yu JX (2008) Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. Proc. VLDB Endow. 1(1):364–375

Solnon C (2010) Alldifferent-based filtering for subgraph isomorphism. Artif Intell 174(12–13):850–864

Stelzl U, Worm U, Lalowski M, Haenig C, Brembeck FH, Goehler H, Stroedicke M, Zenkner M, Schoenherr A, Koeppen S et al (2005) A human protein–protein interaction network: a resource for annotating the proteome. Cell 122(6):957–968

Strandberg PE, Ostrand TJ, Weyuker EJ, Sundmark D, Afzal W (2018) Automated test mapping and coverage for network topologies. In: Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis, pp 73–83

Sun S, Luo Q (2020) In-memory subgraph matching: an in-depth study. In: Proceedings of the 2020 ACM SIGMOD international conference on management of data, pp 1083–1098

Sun Z, Wang H, Wang H, Shao B, Li J (2012) Efficient subgraph matching on billion node graphs. Proc VLDB Endow 5(9):788–799

Ullmann JR (1976) An algorithm for subgraph isomorphism. J ACM (JACM) 23(1):31–42

Ullmann JR (2011) Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. J. Exp. Algorithmics (JEA) 15:1–1

Weich A, Flamann C, Berges J, Singh KP, Chambers D, Lai X, Wolkenhauer O, Berking C, Kroenke G, Gupta S et al (2024) The integration of network biology and pharmacophore modeling suggests repurposing clindamycin as an inhibitor of pyroptosis via caspase-1 blockage in tumor-associated macrophages. bioRxiv

Yang J, Leskovec J (2015) Defining and evaluating network communities based on ground-truth. Knowl Inf Syst 42(1):181–213

Zampelli S, Deville Y, Solnon C (2010) Solving subgraph isomorphism problems with constraint programming. Constraints 15(3):327–353

Zeng L, Dong Z-K, Yu J-Y, Hong J, Wang H-Y (2019) Sketch-based retrieval and instantiation of parametric parts. Comput-Aided Des 113:82–95

Zheng Q, Skillicorn D (2017) Social networks with rich edge semantics. Taylor & Francis, London

Authors and Affiliations

Vincenzo Bonnici[1] · Roberto Grasso[2] · Giovanni Micale[3] · Antonio di Maria[3] · Dennis Shasha[4] · Alfredo Pulvirenti[3] · Rosalba Giugno[5]

✉ Vincenzo Bonnici
vincenzo.bonnici@unipr.it

Roberto Grasso
roberto.grasso@phd.unict.it

Giovanni Micale
gmicale@dmi.unict.it

Antonio di Maria
antoniodm@unict.it

Dennis Shasha
shasha@cims.nyu.edu

Alfredo Pulvirenti
apulvirenti@dmi.unict.it

Rosalba Giugno
rosalba.giugno@univr.it

[1] Department of Mathematical, Physical and Computer Sciences, University of Parma, Parco area delle Scienze, 53A, 43124 Parma, Italy

[2] Department of Physics and Astronomy, University of Catania, Via S. Sofia, 64, 95123 Catania, Italy

[3] Department of Clinical and Experimental Medicine, University of Catania, Via Santa Sofia, 89, 95123 Catania, Italy

[4] Computer Science Department, Courant Institute of Mathematical Sciences, 251 Mercer street, New York, NY 10012, USA

[5] Department of Computer Science, University of Verona, Strada le Grazie, 15, 37134 Verona, Italy