The VLDB Journal
DOI 10.1007/s00778-016-0447-0

REGULAR PAPER

PANDA: toward partial topology-based search on large networks in a single machine

Miao Xie[1,2,3] · **Sourav S. Bhowmick**[1] · **Gao Cong**[1] · **Qing Wang**[2]

Received: 23 March 2016 / Revised: 2 September 2016 / Accepted: 1 November 2016

Abstract A large body of research has focused on efficient and scalable processing of subgraph search queries on large networks. In these efforts, a query is posed in the form of a connected query graph. Unfortunately, in practice end users may not always have precise knowledge about the topological relationships between nodes in a query graph to formulate a connected query. In this paper, we present a novel graph querying paradigm called *partial topology-based network search* and propose a query processing framework called PANDA to efficiently process *partial topology query* (PTQ) in a single machine. A PTQ is a disconnected query graph containing multiple connected *query components*. PTQs allow an end user to formulate queries without demanding precise information about the complete topology of a query graph. To this end, we propose an exact and an approximate algorithm called SEN- PANDA and PO- PANDA, respectively, to generate top-k *matches* of a PTQ. We also present a *subgraph simulation-based optimization* technique to further speedup the processing of PTQs. Using real-life networks with millions of nodes, we experimentally verify that our proposed algorithms are superior to several baseline techniques.

Keywords Partially connected query graph · Query evaluation algorithms · Steiner tree · Label propagation · Large network · Single machine

This work was primarily done when the first author was visiting Nanyang Technological University.

Electronic supplementary material The online version of this article (doi:10.1007/s00778-016-0447-0) contains supplementary material, which is available to authorized users.

✉ Sourav S. Bhowmick
assourav@ntu.edu.sg

Miao Xie
0520shui@163.com

Gao Cong
gaocong@ntu.edu.sg

Qing Wang
wq@nfs.iscas.ac.cn

[1] School of Computer Science and Engineering, Nanyang Technological University, Singapore, Singapore

[2] Institute of Software, Chinese Academy of Sciences, Beijing, China

[3] CSI Euler Department, Huawei, Beijing, China

1 Introduction

Networks are of increasing importance in modeling complex structures such as molecular interactions, social relationships, co-purchase behavior, and program dependence. Due to the explosive growth of network data in recent years, querying them has emerged as an important research problem for real-world applications that are centered on large network data. At the core of many of these applications lies a common and important query primitive called subgraph search, where we want to retrieve one or more subgraphs in a network G that *exactly* or *approximately* match a user-specified query graph Q. Exact subgraph search strictly searches for isomorphic subgraphs in G that matches Q [11,25,32]. On the other hand, similar or approximate search allows the topology of the query graph to be mismatched to a certain degree. These approaches utilize edit distance [30], common connected subgraphs [26,33], graph homomorphism [9,17], or graph simulation [19], to retrieve *similar* query results.

1.1 Motivation

Since the last decade, state-of-the-art techniques for subgraph search on large network data have made significant

Published online: 18 November 2016

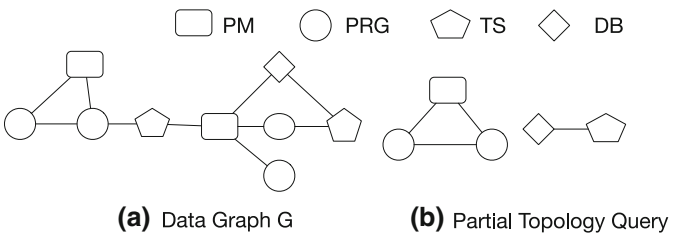


Fig. 1 Collaboration network and a partial topology query. The shapes of nodes represent different professional skills

progress toward building efficient and scalable subgraph query processing framework. A common thread among these innovative techniques is the assumption that the query graph specified by a user is a connected graph. That is, a user precisely knows the topological structure of what it is they are looking for. Unfortunately, due to the topological complexity of the underlying network data, often it is unrealistic to assume that an end user is aware of precise relationships between the nodes in a query graph to formulate a "valid" connected query. There are many instances in which the user has a clear goal in mind but only a vague idea of how the query will be specified. Consequently, she may not always be able to express a query using a single connected graph. To motivate this scenario, consider the following set of user problems.

Example 1 Consider the collaboration network in Fig. 1a where each node represents a person having attributes representing professional skills (e.g., project manager (PM), database developer (DB), programmer (PRG), and software tester (TS)). Each edge indicates whether a pair of persons have collaborated with each other. Suppose a manager Bob wants to organize a team for a new software development project by issuing a subgraph search query on this network. Bob needs five team members (1 PM, 2 PRG, 1 TS and 1 DB) and hopes that the PM and two PRG have collaborated with each other. Furthermore, he wishes that the TS and DB have close collaboration. At the same time, although there is no strict structural requirement between these two subteams, Bob hopes that they can be as close as possible. Additionally, since Bob has sufficient funds to hire five persons, he does not wish to hire a member that can play multiple roles (e.g., PRG as well as DB) as such member may be overloaded with responsibilities leading to delay in completion of his project. Observe that these requirements cannot be expressed using a connected query graph. Instead, Fig. 1b is a query graph that embodies the above requirements of Bob. Observe that it is disconnected and consists of two connected *query components*, each of them shows the topology that needs to be matched exactly in any query result and the distance between these matching components need to be as close as possible. Lastly, matching subgraphs of these two components need to be disjoint as it is not desirable for a member to perform multiple roles.

Example 2 Although many protein complexes have been reported in the literature for various species (e.g., yeast, human, fly, rat, mouse), information of many of their interaction patterns is still missing, especially in rat and mouse [2]. Alice, a biologist, knows that in a rat PPI (protein–protein interaction) network, the interactions of `MCM2`, `MCM4`, and `MCM6` proteins form a triangle and `PCNA` interacts directly with `Ccnd1`. Although she knows that `PCNA` or `Ccnd1` does not directly interact with `MCM` complex, she is unaware of how they are connected. Alice wishes to discover how these proteins are related to the human PPI network. Consequently, similar to the preceding example, her query graph is disconnected containing two connected query components as depicted in Fig. 2. Observe that matches to these query components in a PPI network are nonoverlapping in nature as they involve a distinct collection of proteins.

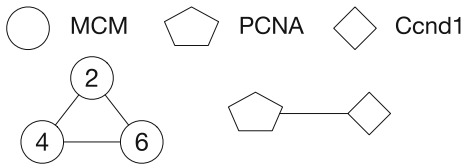


Fig. 2 Partial topology query for Example 2

Unfortunately, the queries posed by Bob and Alice cannot be efficiently or accurately processed by state-of-the-art graph pattern matching algorithms as they demand a connected query graph as input. Consider Example 1 and exact subgraph query matching algorithms. If Bob modifies his query to a connected query graph as shown in Fig. 3a, then there is no matching subgraph in Fig. 1a. Since disconnected query graphs such as Fig. 1b are not supported by existing approaches, it has to be transformed to a set of connected query graphs representing all possible connections between the two subteams. However, the processing cost of such query collection increases dramatically with increase in the size and number of connected components, especially for large networks.

Alternatively, Bob may evaluate the query in Fig. 3a using an approximate subgraph matching algorithm [17,28,30]. Unfortunately, these techniques may retrieve poor quality results that do not match Bob's needs (detailed in Sect. 7). For instance, maximal common connected subgraph-based or edit distance-based techniques may return the result in Fig. 3b. NeMa [17] will retrieve the subgraph in Fig. 3c. Similarly, keyword search on graphs [12] or *topology-free*

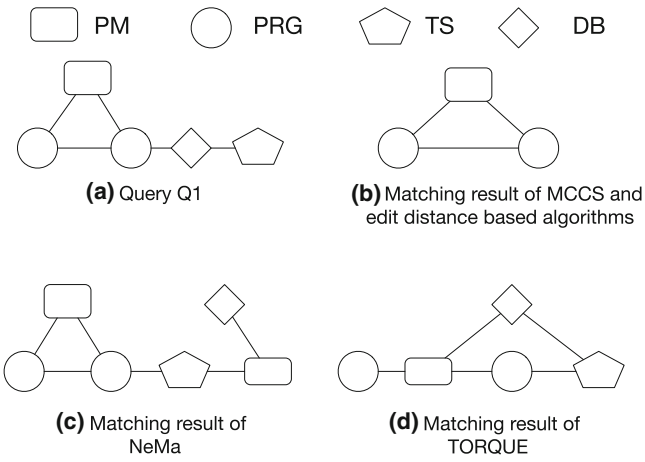


Fig. 3 Connected query graph and results

graph matching techniques that seek subgraphs spanning a set of keywords (such as TORQUE [1]) will return the result in Fig. 3d. Observe that all these results do not capture all the requirements of Bob. In fact, the top-2 connected subgraphs in Fig. 1a that satisfy Bob's requirements are depicted in Fig. 4.

1.2 Our contributions

In this paper, we address the aforementioned limitations of posing connected query graphs by making the following contributions. First, we present a novel graph querying paradigm called *partial topology-based network search* for efficiently evaluating a *partial topology query* (e.g., Fig. 1b). Intuitively, a partial topology query Q_P (PTQ) comprises two or more disjoint *query components*. Each query component is a connected graph. Given an undirected network G, the goal of this problem is to return top-k *matching subgraphs* of Q_P where a match M is a connected subgraph of G. For each query component q_i in Q_P, there exists a subgraph g_i of M such that it is isomorphic to q_i. In addition, the matching subgraphs of any two different query components in M must not overlap, and the distance between them should be as close as possible since the relationship between two entities in a network becomes less relevant as distance between them increases [3]. Note that Examples 1 and 2 highlight the nonoverlapping or disjoint nature of matching subgraphs of a partial topology query.[1]

Second, we show that our proposed problem is NP-hard by reducing it from the well-known Steiner tree problem [16]. To tackle this challenge, we propose a generic PTQ processing framework called PANDA (**PA**rtial Topology-based **N**etwork **D**ata Se**A**rch), which comprises three main components, namely, *matching subgraph generation*, *merged graph construction*, and *partial topology-based matching results generation*. Specifically, the first component finds all isomorphic matching subgraphs for each query components. These matching subgraphs and their locations in G are represented concisely in the form of a *merged graph* by the second component. Finally, the last component extracts top-k matching results of Q_P from the merged graph.

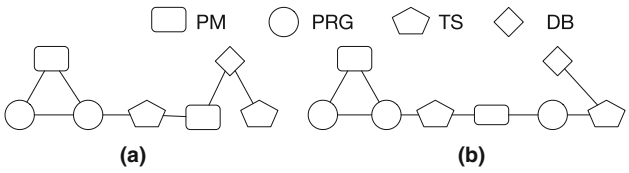


Fig. 4 Top-2 matching subgraphs of the partial topology query in Fig. 1b

Based on this framework, we present two algorithms called SEN- PANDA and PO- PANDA, for networks residing in a single machine. Both our algorithms do not leverage any index as it has been recently reported in [25] that index-based solutions do not scale. SEN- PANDA is an exact algorithm based on group Steiner tree (GST) [6]. It is particularly suitable for applications where the underlying network data are not very large and result quality is paramount. This technique becomes expensive as the number of query components and network size increase due to multiple invocation of the GST algorithm. Hence, we present an algorithm called PO- PANDA that generates approximate results with tight performance guarantee. It avoids utilizing GSTs by exploiting a novel *label propagation*-based technique that runs in polynomial time to implement the third component. Furthermore, since finding matching subgraphs using subgraph isomorphism is an NP-complete problem, we propose a *simulation-based optimization* technique that leverages subgraph simulation [14] in PO- PANDA to judiciously invoke subgraph isomorphism test on a significantly smaller subgraph of G on demand to further improve its efficiency and scalability without compromising on the result quality.

Third, we investigate the performances of the proposed algorithms and optimization techniques on real-world networks of different sizes in a single machine setup. Our experimental study demonstrates clearly that our PANDA framework is significantly more efficient, scalable and generates superior quality results for the partial topology querying problem compared to several baseline approaches.

1.3 Paper organization

The rest of this paper is organized as follows. We formally define the partial topology-based network search problem and the PANDA framework in Sect. 2. We present the merged graph construction component in Sect. 3. The SEN- PANDA

[1] As we shall see later, our solution framework can easily handle overlapping cases by mapping it to a Steiner tree problem.

Table 1 Key notations

Symbol	Definition
$G = (V, E)$	A network
g	A subgraph
$Q = (V_q, E_q)$	A query graph
$Q \prec g$	g matches a query Q via subgraph isomorphism
$Q \prec_s g$	g matches a query Q via subgraph simulation
$Q_P = (q_1, \ldots, q_\ell)$	A partial topology query
$Q_P \prec_p M$	A valid match M of a partial topology query
$cost(M)$	Cost of partial topology-based match M
$\text{SM}_i = \{g_1, \ldots, g_{k_i}\}$	A set of matching subgraphs for the query component q_i
$\mathbb{S} = \{s_1, s_2, \ldots, s_n\}$	A set of merged nodes
I_s	Inner graph of a merged node s
Υ_s	Label of a merged node s
$H = (V_H, E_H)$	Merged graph
$\mathcal{P}_i$	LMQ for label i

and PO- PANDA algorithms are presented in Sects. 4 and 5, respectively. We discuss simulation-based optimization technique in Sect. 6. Experimental results are presented in Sect. 7. We review related work in Sect. 8. Section 9 concludes this paper. Table 1 describes the key notations used in this paper.

2 Partial topology-based network search problem

In this section, we first present the basic graph terminology to facilitate our subsequent discussions. Then, we formally introduce the *partial topology-based network search* problem. Lastly, we provide an overview of the PANDA framework to address it.

2.1 Graph terminology

A graph or a network $G = (V, E, L)$ is an undirected labeled graph where V is a set of nodes, $E \subseteq V \times V$ is a set of edges, L is a labeling function that maps each node $u \in V$ to a set of labels $L(u) \subseteq \Sigma$. The function $L()$ specifies node attributes in a given application (e.g., keywords, blogs, names, emails, professionals, companies) and the label set Σ denotes all such attributes. We denote G as (V, E) when it is clear from the context. We assume that the given data graph (network) is connected.

A graph $G_1 = (V_1, E_1)$ is a *subgraph* of another graph $G_2 = (V_2, E_2)$ if there exists a *subgraph isomorphism* from G_1 to G_2, denoted by $G_1 \subseteq G_2$. We may also say that G_2 *contains* G_1. Given a network $G = (V, E)$ and a query graph $Q = (V_q, E_q)$, the goal of *subgraph isomorphism* is to find every *matching subgraph* $g = (V_g, E_g)$, $g \subseteq G$, denoted by $Q \prec g$, such that there exists a bijection $f : V_q \to V_g$ that satisfies (1)$\forall v \in V_q, L(v) \subseteq L(f(v))$ and (2) $\forall (u, v) \in E_q, (f(u), f(v)) \in E_g$. From the definition of subgraph isomorphism, notice that it strictly requires topology in a matching subgraph to be same as the query graph. It is well-known that the subgraph isomorphism test is an NP-complete problem.

A *path* ρ in G is a sequence of nodes $v_1, \ldots, v_n$ such that $(v_i, v_{i+1}) \in E$ of G for $i \in [1, n-1]$. The *length* of a path ρ, denoted by $length(\rho)$, is the number of edges (*resp.* sum of the weights of edges) in ρ for an unweighted (*resp.* weighted) graph. Specifically, a *path* between two disjoint subgraphs of G, $g_1 = (V_{g_1}, E_{g_1})$ and $g_2 = (V_{g_2}, E_{g_2})$, is a sequence of nodes $v_1, \ldots, v_n$, such that $(v_i, v_{i+1}) \in E$ for $i \in [1, n-1]$ and $v_1 \in V_{g_1}, v_n \in V_{g_2}$ but $v_i \notin V_{g_1} \cup V_{g_2}, \forall i \in [2, n-1]$.

2.2 Partial topology query (PTQ)

A *partial topology query* $Q_P = (q_1, \ldots, q_\ell)$ is a set of connected graphs where $\ell \geq 2$. Each $q_i = (V_{q_i}, E_{q_i})$ is an unweighted and undirected graph, called a *query component*. A query component q_i is a connected graph that comprises a set of labeled nodes V_{q_i} and a set of edges E_{q_i}. Note that the query components are disjoint graphs. We refer to i in q_i as *component index*. Figure 1b shows an example of a PTQ where $\ell = 2$.

2.3 Problem statement

We now formally define the problem addressed in this paper. We begin by defining the notion of *partial topology-based matching*.

Definition 1 *(Partial Topology-based Matching)* Given a network $G = (V, E)$, a partial topology query (PTQ), $Q_P = (q_1, \ldots, q_\ell)$, $M \subseteq G$ is a valid **match** of Q_P, denoted by $Q_P \prec_p M$, if and only if it satisfies the following conditions.

1. M is a connected graph.
2. For each query component q_i, there exists a subgraph $g_i \subseteq M$, which contains at least one bijection for matching to q_i via **subgraph isomorphism**. That is, it is possible that g_i has many (possibly overlapping) isomorphic matches to q_i.
3. $\forall i, j \in [1, \ldots, \ell] \wedge i \neq j, V_{g_i} \cap V_{g_j} = \emptyset$

We can measure the "quality" of a valid match by computing the *cost* of M as follows:

$$cost(M) = \sum_{e \in E_M \setminus \bigcup_{i \in [1,\ldots,\ell]} E_{g_i}} e.weight \quad (1)$$

In the above equation, E_M denotes the set of edges in M and $e.weight$ is the weight of an edge e. Observe that the cost of a match is the sum of weights of all edges that do not "belong" to matching subgraphs of the query components (i.e., *both* the adjacent nodes of an edge e are not in a matching subgraph). This is because the matching subgraphs of query components are the exact matching results via subgraph isomorphism, satisfying the topological relationships between nodes in each query component in a PTQ. Hence, the weights of edges in these matching subgraphs should not contribute to the cost. Consequently, $cost(M)$ measures the total cost to connect the matching subgraphs. Naturally, our goal is to ensure that $cost(M)$ is as low as possible for a valid match as entities in a network become less relevant as distance between them increases [3].

Example 3 Consider the PTQ in Fig. 1b on the network in Fig. 1a. Figure 4a shows a valid partial topology-based matching M of the query where the two subgraphs g_1 and g_2 with $V_{g_1} = \{\text{PM}_1, \text{PRG}_1, \text{PRG}_2\}$ and $V_{g_2} = \{\text{DB}_1, \text{TS}_2\}$ are isomorphic to the two query components, respectively. Observe that g_1 and g_2 are not overlapping (i.e., $V_{g_1} \cap V_{g_2} = \emptyset$). There are three edges, $(\text{PRG}_2, \text{TS}_1)$, $(\text{TS}_1, \text{PM}_2)$, and $(\text{PM}_2, \text{DB}_1)$ that are not in the matching subgraphs g_1 and g_2. Hence, $cost(M) = \sum_{e \in E_M \setminus (E_{g_1} \cup E_{g_2})} e.weight = 3$.

Remark Observe that Condition 3 of Definition 1 ensures that the matching subgraphs of different query components are disjoint. This is motivated by the example scenarios depicted in Examples 1 and 2. However, Condition 3 makes PTQ more challenging to process due to the following reasons.

- First of all, if we ignore Condition 3 (i.e., overlapping matching subgraphs for different query components are acceptable), the problem of matching PTQ can be reduced to the classic Steiner tree problem after finding all matching subgraphs of the query components. On the contrary, under the aforementioned definition of PTQ, it is not simply a Steiner tree problem as such approach may lead to invalid matches (elaborated in Sect. 4.1).
- Secondly, Condition 3 does not necessarily make it easier to prune many matching candidates since such aggressive pruning cannot guarantee (with performance bound) finding the optimal match to a PTQ as the optimal one may potentially involve these pruned candidates. Observe that in order to search for the optimal match for a PTQ, we need to consider all connections between the matching subgraphs of different query components (as encapsulated by the SEN- PANDA algorithm), preventing us from pruning overlapping matching subgraphs of different query components in advance.

Notably, as we shall see later, our proposed algorithms to handle PTQs can also be easily extended to handle cases where overlapping matching subgraphs for different query components are permissible (when Condition 3 is discarded in Definition 1).

Definition 2 *(Minimal Partial Topology-based Matching)* Given a network $G = (V, E)$, a partial topology query $Q_P = (q_1, \ldots, q_\ell)$, the **minimal partial topology-based matching** of Q_P is the minimal cost subgraph $M_{min} \subseteq G$, which satisfies $M_{min} = argmin_M(cost(M))$ where $M \subseteq G \wedge Q_P \prec_p M$.

Definition 3 *(Partial Topology-based Network Search Problem)* Given a network $G = (V, E)$, a partial topology query $Q_P = (q_1, \ldots, q_\ell)$, and a positive integer k, the goal of **partial topology-based network search problem** is to find the top-k minimal cost partial topology-based matchings of Q_P in G, denoted by $M_1, \ldots, M_k$, where $cost(M_1) \leq cost(M_2) \leq \cdots \leq cost(M_k)$.

The PTQ in Fig. 1b and its result matches in Fig. 4 depict an example of the partial topology-based network search problem for $k = 2$. Note that in the above problem definition, we select subgraph isomorphism for exact subgraph matching. However, other types of subgraph matching techniques such as *strong simulation* [19] can be easily incorporated in Definition 1.

Theorem 1 *The Minimal Partial Topology-based Matching Problem is NP-hard.*

Proof To show the intractability of the problem, we shall prove this theorem by reducing from the Steiner tree problem, which is a classic NP-hard problem [16] when the number of Steiner nodes is larger than 2. Given a graph $G = (V, E)$ and a subset of nodes called Steiner nodes, $S \subseteq V$, the goal of the Steiner tree problem is to find a minimal cost subtree, $T = (V_T, E_T)$, $T \subseteq G$, such that $\forall v \in S, v \in V_T$. Given an instance of the Steiner tree problem, it can be reduced to an instance of the Minimum Partial Topology-based Matching Problem in linear time as follows. Firstly, we assign a unique label as the identifier to each node of S in G. All other nodes $v \notin S$ of G are unlabeled. Secondly, a query can be created as $Q_P = (q_1, \ldots, q_{|S|})$, where q_i contains only one node labeled by the same label of v_i, $v_i \in S$. Here $|S|$ is the number of nodes in S. The solution of the minimal partial topology-based matching problem is a solution to the Steiner tree problem because (a) the solution tree is connected, (b) each q_i can only be isomorphic to a unique node in S, (c) $\forall v_i, v_j \in S \wedge i \neq j$, then $v_i \neq v_j$, and (d) since all query components only contain isolated nodes, the cost of a partial topology-based match is the total cost of edges that connect these matching nodes, which is same as the cost of the Steiner tree. Similarly, we can see that the solution of the Steiner tree problem is a solution to the minimal partial topology-based matching problem. Thus, the minimal partial topology-based matching problem is NP-hard. □

2.4 The PANDA framework

We now provide an overview of the PANDA framework to address the partial topology-based network search problem. The main idea is as follows. Given a network $G = (V, E)$ and a PTQ $Q_P = (q_1, \ldots, q_\ell)$, we find a set of matching subgraphs for each query component q_i, denoted as $\text{SM}_i = \{g_1, \ldots, g_j\}$, such that $\forall g \in \text{SM}_i, q_i \prec g$ and $g \subseteq G$. In total, we can get ℓ sets of matching subgraphs for all query components, $\{\text{SM}_1, \ldots, \text{SM}_\ell\}$ where $\text{SM} = \text{SM}_1 \cup \cdots \cup \text{SM}_\ell$. Then, we *merge* these matching subgraphs in SM into a set of special nodes called *merged nodes*. Using these merged nodes, we generate a new connected graph called *merged graph* from G by "hiding" SM via merged nodes. Finally, we search for paths to connect these merged nodes and return the top-k *valid* partial topology matches. This strategy can be encapsulated in the PANDA framework by utilizing the following three components:

1. **Matching Subgraph Generation:** Find all matching subgraphs, denoted as $\text{SM} = \bigcup_{i=1}^{l} \text{SM}_i$ for $Q_P = \{q_1, \ldots, q_\ell\}$ in G, such that for each q_i, $\forall g \in \text{SM}_i, q_i \prec g$.
2. **Merged Graph Construction:** Based on G and SM, construct a *merged graph* from G, comprising merged nodes that encapsulate the matching subgraphs of the query components.
3. **Partial Topology-based Matching Results Generation:** This step finds shortest paths in the merged graph to connect matching subgraphs (which are contained in merged nodes) for different query components in order to *progressively* search for top-k matching results of Q_P.

Note that realizing the PANDA framework for finding optimal or approximate matches for a PTQ is challenging for the following reasons. Firstly, it is possible that there are overlapping matching subgraphs of different query components that are contained in the same merged node of a *merged graph*. Hence, generating *valid* matches from the merged graph is challenging due to the violation of Condition 3 in Definition 1 (as remarked earlier). Secondly, even if some matching subgraphs of different query components are disjoint, the cost of edges to connect them together in a merged node (i.e., "hidden" cost of a merged node) may largely affect the quality of the match. Hence, in order to realize the third component, we need to consider not only all combination of paths between merged nodes of different query components, but also the hidden cost of merged nodes. It is challenging to realize it efficiently with superior quality guarantee. Lastly, finding all matching subgraphs via subgraph isomorphism for all query components in the first step is prohibitively expensive, especially for large networks.

In order to address the aforementioned challenges, we propose an exact algorithm called SEN- PANDA for finding the optimal solution. It generates a set of *single-label merged graphs* (SMG) and leverages on the *group Steiner tree* (GST) algorithm to address the first two challenges. Although it can find the optimal solution, it can be extremely costly because of exponential number of SMGs and the complexity of GST. So we also propose an approximate algorithm called PO- PANDA, which only needs to generate one merged graph and solve the first two challenges while realizing the third component. It deploys a novel search strategy called *label propagation*, which can reduce the search space significantly by propagating labels along shortest paths from all merged nodes. Finally, for the last challenge, a *simulation-based* optimization strategy is introduced to reduce the cost of isomorphism-based matching subgraph generation without affecting the approximate factor.

Observe that the first step can be addressed by any state-of-the-art subgraph matching algorithm for large networks (e.g., [11]). Hence, in the subsequent sections, we shall focus on the remaining two steps of the PANDA framework.

3 Merged graph construction

Notice that nodes of a matching subgraph $g \subseteq G$ may match nodes in more than one query components of Q_P. In other words, it is possible for two matching subgraphs g_i and g_j, $i \neq j$ and $g_i, g_j \in \text{SM}$ to *overlap*, i.e., $V_{g_i} \cap V_{g_j} \neq \emptyset$. Furthermore, since G is a connected network, the matching subgraphs may be connected through one or more paths. Hence, we need a systematic way to represent SM in order to facilitate efficient search for result matches of Q_P. In this section, we introduce a data structure called *merged graph* toward this goal. We begin by formally introducing the notion of *merged nodes*, which are used to construct a merged graph.

3.1 Merged node and inner graph

Intuitively, a *merged node* $s \notin V$ represents an *aggregation* of one or more matching subgraphs and is annotated with a set of *labels* $\Upsilon_s \nsubseteq \Sigma$. Υ_s represents indexes of the query component(s) whose matching subgraphs are contained in the aggregated subgraph of s. The aggregated subgraph is referred to as the *inner graph* of s, denoted by $I_s = (V_s, E_s)$. Let us elaborate further. Let V_g be the set of nodes, and E_g be the set of edges of a matching subgraph g. Then, if $V_{g_i} \cap V_{g_j} \neq \emptyset$ for $g_i \in \text{SM}_a$ and $g_j \in \text{SM}_b$ where $i \neq j$, $q_a \prec g_i$, $q_b \prec g_j$, and $a, b \in [1, \ell]$, we "merge" them together and represent them with a merged node s whose inner graph is *union* of g_i and g_j. That is, $I_s = (V_s, E_s)$, $V_s = V_{g_i} \cup V_{g_j}$ and $E_s = E_{g_i} \cup E_{g_j}$. The *label set* (or *label* for brevity) of s is $\Upsilon_s = \{a, b\}$. On the other hand, if a matching subgraph

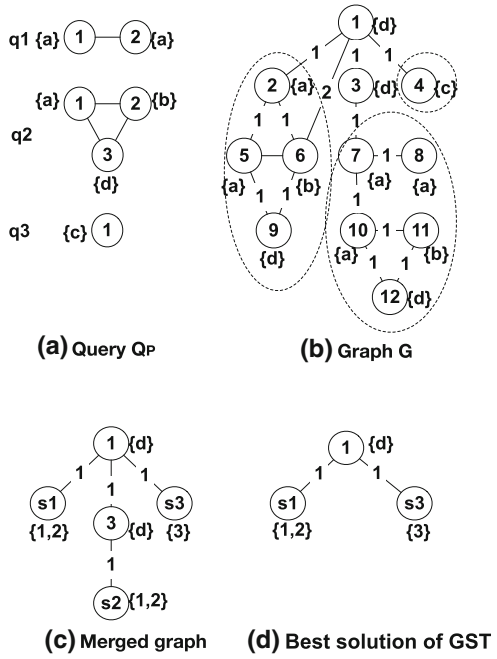


Fig. 5 Merged graph

Algorithm 1: GENERATEMERGEGRAPH

Input: A network $G = (V, E)$, a set of matches in G for ℓ query components: $\{\text{SM}_1, \ldots, \text{SM}_\ell\}$
Output: A merged graph $H = (V_H, E_H)$

```
Initialize H ← G;
Initialize a set SM = ⋃_{i=1}^{l} SM_i;
for g ∈ SM do
    Initialize a merged node s;
    S_o ← g;
    REMOVE(g, SM);
    repeat
        for g_a ∈ SM do
            if ∃g_b ∈ S_o, V_{g_b} ∩ V_{g_a} ≠ ∅ then
                ADD(g_a, S_o);
                REMOVE(g_a, SM);
    until S_o is not changed;
    Create merged node s with label set Υ_s;
    I_s ← ⋃_{g∈S_o} g;
    V_H ← V_H \ V_s;
    E_H ← E_H \ E_s;
    Create s.E_o and E_s;
    E_H ← (E_H ∪ E_s) \ s.E_o;
return H
```

$g \in \text{SM}_a$ is nonoverlapping or disjoint to other subgraphs in SM (i.e., $\forall g' \in \text{SM}, V_g \cap V_{g'} = \emptyset$), it is represented by a merged node s whose inner graph I_s comprises of $V_s = V_g$ and $E_s = E_g$. In this case, $\Upsilon_s = \{a\}$.

Notice that a merged node has at least one label, and it specifies that there is at least one matching subgraph of a query component (whose component index is in Υ_s) in its inner graph. That is, $\exists g \in \text{SM}_i, i \subseteq \Upsilon_s$. Lastly, the *outer edges* of s is an edge set $s.E_o = \{(v_i, v_j) | \exists (v_i, v_j) \in E, v_i \in V_s \wedge v_j \notin V_s\}$.

Example 4 Consider the PTQ Q_P and the network G in Fig. 5a, b, respectively. The identifier of each node is shown within it, and its label is in parenthesis in its vicinity. In the sequel, we shall refer to a node v with identifier i as v_i. Each edge in G is labeled by its weight. Observe that for q_1, there are three matching subgraphs in G. That is, $\text{SM}_1 = \{g_1, g_2, g_3\}$ where $V_{g_1} = \{v_2, v_5\}$, $V_{g_2} = \{v_7, v_8\}$ and $V_{g_3} = \{v_7, v_{10}\}$. Similarly, there are two subgraphs in G that are isomorphic to q_2. Hence, $\text{SM}_2 = \{g_4, g_5\}$ where $V_{g_4} = \{v_5, v_6, v_9\}$ and $V_{g_5} = \{v_{10}, v_{11}, v_{12}\}$. Lastly, q_3 only matches node v_4 in G, i.e., $\text{SM}_3 = \{g_6\}$ where $V_{g_6} = \{v_4\}$. Since $V_{g_1} \cap V_{g_4} \neq \emptyset$, $V_{g_2} \cap V_{g_3} \cap V_{g_5} \neq \emptyset$, and g_6 is disjoint, we can represent these matching subgraphs with three merged nodes s_1, s_2, and s_3, respectively. Thus, $I_{s_1} = g_1 \cup g_4$, $I_{s_2} = g_2 \cup g_3 \cup g_5$, and $I_{s_3} = g_6$. The dotted circles in Fig. 5b show these inner graphs. Furthermore, $\Upsilon_{s_1} = \{1, 2\}$, $\Upsilon_{s_2} = \{1, 2\}$, and $\Upsilon_{s_3} = \{3\}$. Lastly, the outer edges of the merged nodes are as follows: $s_1.E_o = \{(v_2, v_1), (v_6, v_1)\}$, $s_2.E_o = \{(v_7, v_3)\}$ and $s_3.E_o = \{(v_4, v_1)\}$.

3.2 Merged graph

Given a network $G = (V, E)$, a PTQ Q_P, and a set of merged nodes $\mathbb{S} = \{s_1, \ldots, s_n\}$, intuitively we transform G to a *merged graph* by substituting the inner graphs with their corresponding merged nodes. For example, consider the inner graphs in the network in Fig. 5b. Figure 5c shows the merged graph by substituting the inner graphs with the merged nodes s_1, s_2, and s_3. Formally, let $V_{inner} = \bigcup_{i=1}^{n} V_{s_i}$, $E_{inner} = \bigcup_{i=1}^{n} E_{s_i}$ and $E_o = \bigcup_{i=1}^{n} s_i.E_o, \forall\, i = [1, \ldots, n]$. Then, a set of edges for a merged node s that connects a node $u \notin I_s \wedge u \in G$ is defined as $E_s = \{(s, u) | \exists (v, u) \in E_o, v \in V_s\}$, where $(s, u).weight = min((v, u).weight, \forall v \in V_s)$. Hence, $E_{st} = \bigcup E_s, \forall s \in \mathbb{S}$. Then, a *merged graph* is defined as $H = (V_H, E_H)$, where $V_H = (V \cup \mathbb{S}) \backslash V_{inner}$, $E_H = (E \cup E_{st}) \backslash (E_{inner} \cup E_o)$.

Note that in a merged graph, we maintain the minimum weight of the edge from a merged node s to another node v in G from all outer edges of s that are connected to v. We shall leverage it later to search for the shortest path between a pair of merged nodes.

Example 5 Reconsider the merged graph in Fig. 5c generated from the network in Fig. 5a and the set of merged nodes $\mathbb{S} = \{s_1, s_2, s_3\}$. Here $E_o = \{(v_2, v_1), (v_6, v_1), (v_7, v_3), (v_4, v_1)\}$ and $E_s = \{(s_1, v_1), (s_2, v_3), (s_3, v_1)\}$. Since $(v_2, v_1).weight = 1$ and $(v_6, v_1).weight = 2$, $(s_1, v_1).weight = min(1, 2) = 1$. Similarly, $(s_2, v_3).weight = 1$ and $(s_3, v_1).weight = 1$.

Algorithm We first initialize the merged graph H as G. Then, for each matching subgraph $g \in$ SM, we fetch all overlapping matching subgraphs in SM and store them in $\mathcal{S}_o$ and subsequently remove them from SM to ensure that these subgraphs can be merged only once. Next, these overlapping matching subgraphs are substituted with a new merged node s and its inner graph and label set. Finally, these overlapping subgraphs in G are substituted with the corresponding merged node by modifying the topology of the merged graph.

The formal description of the algorithm is reported in Algorithm 1. It first initializes the merged graph H as G. Then, for each matching subgraph $g \in$ SM, it fetches all overlapping matching subgraphs in SM and store them in $\mathcal{S}_o$. That is, $\forall g \in \mathcal{S}_o, \exists g' \in \mathcal{S}_o, V_g \cap V_{g'} \neq \emptyset$ (Lines 7–12). It removes $\mathcal{S}_o$ from SM to ensure that these subgraphs can be merged only once. Next, the algorithm substitutes these overlapping matching subgraphs with a new merged node s and its inner graph and label set (Lines 13–14). Finally, it substitutes these overlapping subgraphs in G with the corresponding merged node by modifying the topology of the merged graph (Lines 15–18). The time complexity is $O(|V|^2)$ because inner graphs of all merged nodes at most contain $|V|$ nodes.

4 Group Steiner tree-enabled exact algorithm

In this section, we present an exact algorithm called SEN-PANDA (G**ST-EN**abled PANDA) to solve the top-k partial topology-based network search problem by leveraging the notion of *group Steiner tree* (GST). We begin by briefly introducing the GST problem.

Given a network $G = (V, E)$ and a family of nodes sets $V_1, V_2, \ldots, V_r$, $V_i \subset V$, where r is a positive number (the number of groups), the goal of *group Steiner tree* (GST) problem is to find a minimal cost tree, $T = (V_T, E_T)$ in G that spans at least one node in each group V_i, i.e., $\forall i \in [1, \ldots, r] : V_T \cap V_i \neq \emptyset$. Note that any algorithm for Steiner tree problem can be utilized to solve the GST problem [6].

4.1 Overview

Given a merged graph H, the aim of the final step in our framework (Sect. 2.4) is to search for top-k minimal cost subgraphs of G where each such subgraph connects at least one matching subgraph of each query component of Q_P. Since the matching subgraphs are encapsulated in inner graphs of merged nodes in H, at first glance it may seem that we can address this problem as follows. First, we classify all merged nodes into ℓ groups according to their labels such that all merged nodes that have label i are classified into the same group. Then, we search for a minimal cost tree in H to span at least one merged node from each group by utilizing an existing GST algorithm. Unfortunately, this approach may produce *invalid* matches due to the overlapping nature of the matching subgraphs in inner graphs. For example, reconsider the merged graph in Fig. 5c. If we utilize an existing GST algorithm to span at least one node in groups $V_{s_i}, i \in \{1, \ldots, \ell\}$, where V_{s_i} is the set of merged nodes labeled with i, we will get a minimal tree as shown in Fig. 5d where s_1 is matched to both q_1 and q_2. Although this graph has the minimal cost, it is an invalid solution for our problem because the matching subgraphs with nodes $\{v_2, v_5\}$ for q_1 and $\{v_5, v_6, v_9\}$ for q_2 are not disjoint, which violates the third condition in Definition 1. In fact, the correct solution is the tree involving the nodes $\{v_1, v_3, s_2, s_3\}$ and its cost is 4.

Main idea of the SEN-PANDA algorithm. If a merged node s has multiple labels (i.e., $|\Upsilon_s| > 1$) in a merged graph H (referred to as *multi-label merged node*), it indicates that the inner graph I_s contains several matching subgraphs, possibly overlapping, for different query components. Furthermore, even if there exists at least one nonoverlapping matching subgraphs for these query components, these matches in I_s may impact the cost associated with a query result as GST algorithms do not consider the length of paths to connect these nonoverlapping matches across different query components. For example, in Fig. 5c., the inner graph of s_2 contains nonoverlapping matching subgraphs of q_1 (nodes v_7 and v_8) and q_2 (nodes v_{10}, v_{11}, and v_{12}). However, the path between v_7 and v_{10} increases the cost of the match. Thus, our idea is to generate a set of *single-label merged graphs* (SMG) from H, where every merged node in each SMG has only one label (i.e., $|\Upsilon_s| = 1$). That is, it matches exactly one query component. These SMGs represent all *valid label combinations* (detailed below) of merged nodes with multiple labels. This enables us to impose the restriction that the inner graph of a merged node in *each* SMG can only have matching subgraphs of a single query component. Consequently, we can find the top-k matches in each SMG by using an existing GST algorithm and rank them globally to generate the final results.

4.2 The SEN-PANDA algorithm

Algorithm 2 outlines the formal procedure for finding top-k matches to a PTQ Q_P. First, for every query component, it utilizes an existing subgraph matching algorithm to search for matching subgraphs in G (Lines 2–3). Next, it generates the merged graph H from these matching subgraphs by invoking Algorithm 1 (Line 4). Lastly, it generates a set of SMGs from H. Based on these graphs, it finds top-k matches to Q_P by reformulating the partial topology-based network search problem to the GST problem on these SMGs (Lines 5–18). We now elaborate on the last step.

Note that a merged node s with multiple labels has $2^{|\Upsilon_s|} - 1$ possible combinations of component index sets. For exam-

Algorithm 2: The SEN- PANDA algorithm

Input: A network $G = (V, E)$, a partial topology query $Q_P = (q_1, \ldots, q_\ell)$, a positive integer k.
Output: Top-k partial topology-based matches of Q_P.

1 Initialize a heap $TreesHeap \leftarrow \emptyset$
2 **for** $i \in \{1, \ldots, \ell\}$ **do**
3 $\text{SM}_i \leftarrow \text{ISOMORPHIC}(q_i, G)$
4 $H \leftarrow \text{GENERATEMERGEGRAPH}(G, \{\text{SM}_1, \ldots, \text{SM}_\ell\})$
5 Initialize a set of SMGs $\mathbb{H} \leftarrow \emptyset$
6 **for** *each label combination* $D = \{\langle s, \Upsilon_s \rangle\}$ *of multi-label merged nodes* **do**
7 **if** D *is valid* **then**
8 **if** $\forall \langle s, \Upsilon_s \rangle \in D, |\Upsilon_s| = 1$ **then**
9 Generate a SMG H_s from H according to D
10 $\mathbb{H} \leftarrow \mathbb{H} \cup H_s$
11 **else**
12 Generate a set of SMGs $\mathbb{H}'$ from H according to D.
13 $\mathbb{H} \leftarrow \mathbb{H} \cup \mathbb{H}'$
14 **for** *each* $H_s \in \mathbb{H}$ **do**
15 **for** $i \in \{1, \ldots, \ell\}$ **do**
16 $V_i \leftarrow \{s | s \in H_s \wedge i \in \Upsilon_s\}$
17 $\text{INSERT}(\text{GST}(TreesHeap, H_s, \{V_1, \ldots, V_\ell\}, k))$
18 $R \leftarrow \text{TOPK}(TreeHeap, k)$
19 **return** R

ple, the merged node s_1 in Example 4 has two labels. Hence, it can represent 3 possible combinations of component index sets (i.e., $\{1\}$, $\{2\}$, $\{1, 2\}$). The algorithm iterates through each *valid* label combination D of multi-label merged nodes, which is a set of *valid merged node-label pairs* (denoted by $\langle s, \Upsilon_s \rangle$), and *updates* H by ensuring that the inner graph of a merged node is updated to contain matching subgraphs of query components whose component indices are in D. A label combination $D = \{\langle s, \Upsilon_s \rangle\}$ for all merged nodes is *valid* if there is at least one merged node s with label i for each query component q_i. Specifically, H is updated as follows (Lines 8–13).

– Consider the scenario where s is assigned one label, i.e., $|\Upsilon_s| = 1$ (Line 8). That is, the merged node represents the matching subgraphs of only one query component. Consequently, only matching subgraphs of the corresponding query component in the merged node's inner graph need to be maintained and remaining nodes and edges can be reassigned as outer edges to the updated merged graph. For example, suppose the multi-label merged nodes s_1 and s_2 in Fig. 5c are assigned the labels `"1"` and `"2"`, respectively, (s_3 has only one label). Observe that this label combination $D = \{\langle s_1, \{1\}\rangle, \langle s_2, \{2\}\rangle, \langle s_3, \{3\}\rangle\}$ is valid because there is at least one merged node containing matching subgraphs of each query component. Consequently, by labeling s_1 to `"1"`, it maintains the induced graph of nodes v_2 and v_5 as inner graph and reassign nodes v_6 and v_9 back to H. Similarly, the induced graph of nodes v_{10}, v_{11} and v_{12} is maintained as inner graph of s_2, and nodes v_7 and v_8 are reassigned to H. Figure 6a depicts the updated merged graph.
– In the second scenario, a multi-label merged node s is replaced by a set of connected single-label merged nodes if and only if there exists nonoverlapping matching subgraphs for their corresponding query components in the inner graph of s. For example, reconsider Fig. 5c. Let s_1 and s_2 are assigned labels $\{1\}$ and $\{1, 2\}$, respectively. Then, we replace s_2 by aggregating two nonoverlapping induced graphs involving nodes $\{v_7, v_8\}$ and $\{v_{10}, v_{11}, v_{12}\}$ into two new connected merged nodes labeled s_2 and s_4, respectively, as shown in Fig. 6b. Observe that since the matching subgraphs involving nodes $\{v_7, v_{10}\}$ and $\{v_{10}, v_{11}, v_{12}\}$ are overlapping, we do not consider it.

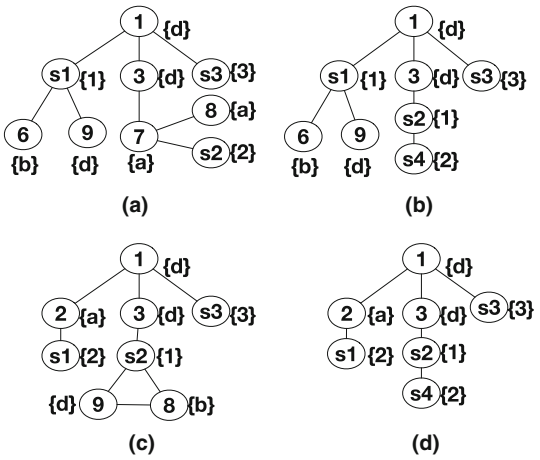


Fig. 6 The set of SMGs for the merged graph in Fig. 5

These scenarios lead to the generation of a set of SMGs $\mathbb{H}$ (Lines 8–13). Figure 6 depicts the SMGs for the merged graph in Fig. 5c. Consequently, the partial topology-based network search problem can be now addressed by reformulating it as a GST problem (Lines 14–17). Specifically, for each SMG, we classify all merged nodes into ℓ groups according to their labels. Then, our search problem is reduced to finding top-k Steiner trees from all SMGs. We can use any state-of-the-art GST algorithm [5,13,15] (Line 17) to this end. A partial topology-based matching M can be obtained by restoring all merged nodes with their corresponding matching subgraphs in their inner graphs. After it gets all top-k result matches for all D, it ranks these results based on $cost(M)$ (Eq. 1) to return the top-k results of Q_P (Lines 18–19). Figure 7 shows the top-1 matches from the SMGs in Fig. 6. The matches in Fig. 7a, b are retrieved from the SMGs in Fig. 6a, b, respectively. The two matches shown in Fig. 7c, d are obtained from Fig. 6c because they have the same cost. The match from Fig. 6d is

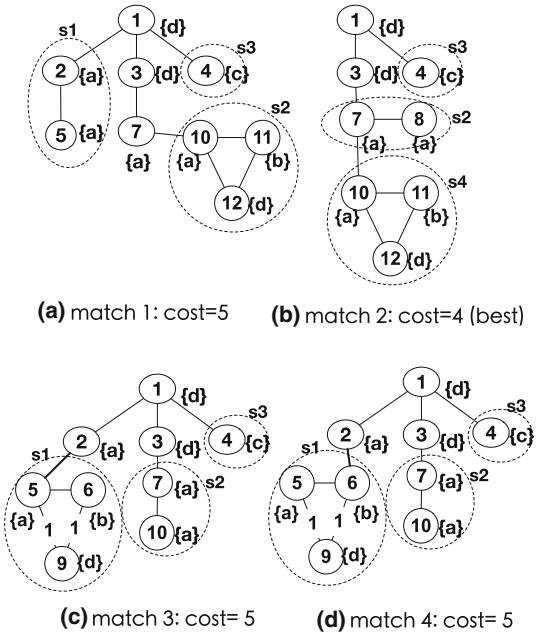


Fig. 7 The top-1 matches from the SMGs in Fig. 6

identical to that in Fig. 6b. Hence, the best solution is Fig. 7b, having the lowest cost 4.

Lemma 1 *Given a merged graph with* $|\mathbb{S}|$ *merged nodes, the maximum number of* SMG*s generated from it is* $\prod_{i\in[1,\ldots,|\mathbb{S}|]}(|\Upsilon_{s_i}| + \sum_{j=2}^{|\Upsilon_{s_i}|} C^j_{|\Upsilon_{s_i}|}\beta^j)$*, where* β *is the maximal number of matching subgraphs for a query component and* j *is the number of multiple labels.*

Proof For each multi-label case, there are at most β^j cases of nonoverlapping matching subgraphs for given j. Hence, we need to consider at most $(|\Upsilon_s| + \sum_{j=2}^{|\Upsilon_s|} C^j_{|\Upsilon_s|}\beta^j)$ cases for a merged node s. If there are $|\mathbb{S}|$ merged nodes, there are at most $\prod_{i\in[1,\ldots,|\mathbb{S}|]}(|\Upsilon_{s_i}| + \sum_{j=2}^{|\Upsilon_{s_i}|} C^j_{|\Upsilon_{s_i}|}\beta^j)$ SMGs. □

Theorem 2 *Let* $\mathbb{G}$ *and* $\mathbb{I}$ *are the time complexities of algorithms for group Steiner tree and subgraph matching problems, respectively. The time complexity of* SEN- PANDA *algorithm is* $O(\ell \times \mathbb{I} + |V|^2 + \mathbb{G}\prod_{i\in[1,\ldots,|\mathbb{S}|]}(\ell + \sum_{j=2}^{\ell} C^j_\ell \beta^j))$.

Proof The time complexity computation can be divided into three phases. In the first phase, we need to invoke subgraph matching algorithm ℓ times for all query components to find matching subgraphs, whose complexity is $O(\ell \times \mathbb{I})$. The time complexities of overlap testing for two node sets and merged graph generation (Algorithm 1) are $O(1)$ and $O(|V|^2)$, respectively. In the final phase, the Steiner tree algorithm will be invoked for each SMG. For a merged node, it has at most ℓ labels in a merged graph. There are at most ℓ single-label cases and $\sum_{j=2}^{\ell} C^j_\ell$ multi-label cases. So, based on Lemma 1 there are at most $\prod_{i\in[1,\ldots,|\mathbb{S}|]}(\ell + \sum_{j=2}^{\ell} C^j_\ell \beta^j)$ SMGs. Putting them together, the complexity of Algorithm 2 is $O(\ell \times \mathbb{I} + |V|^2 + \mathbb{G}\prod_{i\in[1,\ldots,|\mathbb{S}|]}(l + \sum_{j=2}^{\ell} C^j_\ell \beta^j))$. □

Theorem 3 *The minimal cost Top-*1 *matching result returned by the* SEN- PANDA *algorithm is optimal when the selected group Steiner tree algorithm returns the exact solution.*

Proof Suppose M_o is the optimal (that is minimal cost) partial topology-based matching result of a given problem, and M is the top-1 matching result return by Algorithm 2. We suppose that $cost(M_o) < cost(M)$. M is obtained from a merged graph where all merged nodes have one and only one label because of operations in Lines 5–11 in Algorithm 2. Since the inner graph of every merged node contains isomorphic match of one query component, there is no additional cost in the inner graph of every merged node. Since $cost(M_o) < cost(M)$, the paths between matching subgraphs for each query component in M are costlier than those of M_o. It is contradictory to the fact that M is the minimal cost Steiner tree returned by an optimal GST algorithm after considering all possible label combinations of multi-label merged nodes. □

Remark Note that the SEN- PANDA algorithm can easily be extended to handle scenarios where overlapping matching subgraphs for different query components are permissible (removal of Condition 3 in Definition 1). Specifically, in this case there may exist some merged nodes with multiple labels in the merged graph because they may contain overlapping matches for more than one query components. Hence, instead of generating a set of SMGs, we can simply generate a merged graph with multi-label merged nodes and find top-k Steiner trees by invoking an optimal GST algorithm and replacing the merged nodes by their inner graphs in these trees. The formal description of this modified algorithm is given Supplementary Material A.

5 Propagation-based algorithm

Although the SEN- PANDA algorithm can find the best solution for partial topology-based network search problem, it becomes inefficient as the number of query components and network size increase. As the number of query components increases, it is inevitable that there are potentially many overlapping matching subgraphs for these components, leading to the generation of exponential number of SMGs (Lemma 1). Consequently, the GST algorithm will be invoked exponential number of times, which can be prohibitively expensive. In this section, we present a novel algorithm called PO- PANDA (**PrO**pagation-based PANDA) that addresses this limitation of SEN- PANDA.

5.1 Overview

The PO- PANDA algorithm returns approximate results with a performance guarantee. In contrast to SEN- PANDA, it avoids generating an exponential number of SMGs and performing exhaustive search on each SMG. The main idea of PO- PANDA can be outlined as follows. We devise a *label propagation* scheme that "propagates" labels (i.e., component indexes) from the merged nodes in the merged graph to other nodes along the shortest paths. Whenever a node receives all ℓ labels of a PTQ Q_P, it can be used to generate a candidate *matching tree* to Q_P, which spans a set of at most ℓ different merged nodes to cover all query components. The algorithm will check whether the candidate tree is *valid* to be a match of the PTQ and terminates after finding top-k valid matches (if any). For example, consider the PTQ and the network in Fig. 5. PO- PANDA searches the results of the query directly on the merged graph in Fig. 5c. During the label propagation process, merged nodes s_1, s_2 and s_3 propagate their labels in the merged graph to other nodes along their shortest paths. When node 3 receives all ℓ labels (i.e., $\{1, 2, 3\}$), a candidate matching tree is generated based on this node (Fig. 8h). The top-1 matching result (Fig. 8i) is retrieved from the candidate trees. Specifically, we design a novel exploration method that guarantees to obtain a valid match for a PTQ. It efficiently ensures that a multi-label merged node and its inner graph do not lead to an invalid candidate match generation. This strategy of searching for top-k matches of a PTQ along the shortest paths from merged nodes enables us to prune many inferior candidates, thus reducing the search space significantly. Our approach guarantees that each node is visited at most twice during the propagation process. Therefore, the propagation can be achieved in polynomial time. Note that PO- PANDA returns approximate solution without exhaustively exploring all candidate matching trees.

Algorithm 3: The PO- PANDA Algorithm

Input: A network $G = (V, E)$, a partial topology query $Q_P = (q_1, \ldots, q_\ell)$, a positive integer k.
Output: Top-k partial topology-based matches of Q_P.

```
1  Initialize a heap TreesHeap ← ∅;
2  for i ∈ {1, ..., ℓ} do
3      SM_i ← ISOMORPHIC(q_i, G);
4  H ← GENERATEMERGEGRAPH(G, {SM_1, ..., SM_ℓ});
5  𝕊 ← GETMERGENODES(H);
6  Initialize ℓ LMQs {P_1, ..., P_ℓ}, P_i ← ∅;
7  for s ∈ 𝕊 do
8      for i ∈ Υ_s do
9          ENQUEUE(s, 0, P_i);
10         s.parent(i) ← s;
11         s.source(i) ← s;
12     if |Υ_s| > 1 then
13         add Υ_s to s.sharedlabels;
14 while ∃P_i ≠ ∅ do
15     for i ∈ ℓ, P_i ≠ ∅ do
16         (v, d_s) ← DEQUEUE(P_i);
17         if i ∉ Υ_v then
18             ADD(i, Υ_v);
19         if |Υ_v| = ℓ then
20             if v.sharedlabels = ∅ or all v's source nodes are
               safe then
21                 Generate a result tree T using v;
22                 Compute cost(T);
23                 ADD(T, cost(T), TreesHeap);
24                 if SIZE(TreeHeap) ≥ k then
25                     δ ← INNERGRAPHCOST(TreeHeap, k);
26                     T' ← FINDVALIDTREES({P_1, ..., P_ℓ}, δ);
27                     if T' ≠ ∅ then
28                         ADD(T', TreeHeap);
29                     R ← GETTOPK(TreeHeap);
30                     Replace merged nodes in R with the
                       corresponding inner graphs;
31                     return R
32         NEIGHBOREXPLORATION(v, i) /* Algorithm 4 */;
```

5.2 The PO-PANDA algorithm

Algorithm 3 outlines the PO- PANDA algorithm. Lines 1–4 realize the first two steps of the PANDA framework, which are identical to Algorithm 2. Lines 5–33 realize the third step of the framework, which differs from SEN- PANDA. For ease of exposition, hereafter we shall denote the label(s) (component index) of *any* node $v \in V_H$ by Υ_v. We begin by introducing some terminology and data structure utilized by the algorithm.

5.2.1 Terminology

Given a merged graph $H = (V_H, E_H)$, a node $v \in V_H$'s *source node* for a label i, $0 < i \leq \ell$, is a merged node s which *propagates* the label $i \in \Upsilon_s$ to v in H. In other words, the source node s represents the source of the label when v owns it. We denote v's source node for its label i as $v.source(i)$. For example, consider the merged graph in Fig. 5c. Assume that the merged node s_1 propagates the labels $\{1, 2\}$ to v_1. Then, $s_1 = v_1.source(1, 2)$. A node v's *parent node* for a label i is a neighbor of v who directly propagates i to v, denoted by $v.parent(i)$. For instance, s_1 is also the parent node of v_1 for labels $\{1, 2\}$.

If v receives more than one label from the same source node s, then we refer to these labels as *shared labels* of v w.r.t. s, denoted as $v.sharedlabels(s)$. For instance, since s_1 propagates two labels to v_1 in the above example, $v_1.sharedlabels(s_1) = \{1, 2\}$. Observe that the shared labels indicate that the matching subgraphs in the inner graph of s for different query components perhaps overlap, leading to an invalid match for our problem. Hence, we call the source node s to be *unsafe* when it propagates more than one

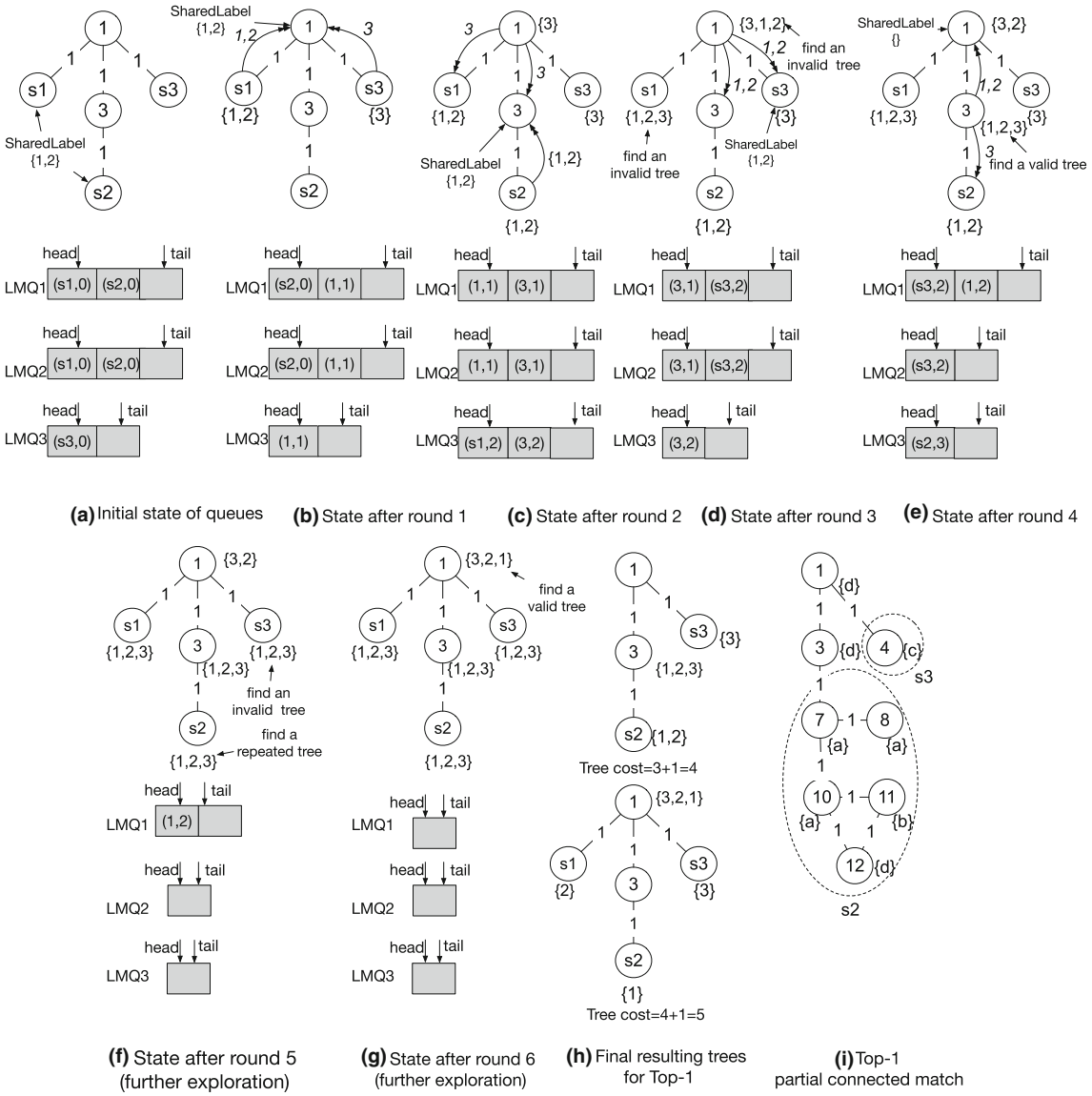


Fig. 8 Label propagation process for Fig. 5. The *double-arrow* edge with the propagated label shows the propagation direction

label to a node v. Otherwise, s is *safe* w.r.t v. Lastly, given a query Q_P with ℓ query components, if a node $v \in V_H$ has all ℓ labels, then v is called a ℓ-*node*.

Label-based Merged Node Queue (LMQ). To facilitate label propagation, we utilize a data structure called *label-based merged node queue* (LMQ). Informally, a LMQ, denoted as $\mathcal{P}_i$, is a priority queue for a specific label i. Each element of $\mathcal{P}_i$ is a 2-tuple $(v, dist(v))$ where $v \in V_H$ and $dist(\cdot)$ is its distance to its source node for the label i. All elements in $\mathcal{P}_i$ are sorted by increasing order of their distance. For example, Fig. 8a depicts three LMQs for the three labels in the merged graph in Fig. 5c.

Given a merged graph H, ℓ LMQs are created (Line 6) to facilitate propagation of these labels from merged nodes through shortest paths to other nodes in H iteratively. Each LMQ is responsible for propagating one label, thus ℓ LMQs in total. All merged nodes with label i will be enqueued to a LMQ $\mathcal{P}_i$ initially with zero distance (i.e., $dist(s) = 0$). Hence, a multi-label merged node will be enqueued in several LMQs. Before adding a merged node to a LMQ, we do the following: (1) For each merged node s, we assign the parent node and source node to be itself (Lines 10–11). (2) If s is a multi-label merged node, then the labels are added as shared labels of s (Lines 12–13).

Example 6 Consider the merged graph in Fig. 5c. Assume that the weights of all edges in G are equal to 1 and $k = 1$. Then, three LMQs are generated as $\ell = 3$. Consequently, $\mathcal{P}_1 = [(s_1, 0), (s_2, 0)]$, $\mathcal{P}_2 = [(s_1, 0), (s_2, 0)]$ and $\mathcal{P}_3 = [(s_3, 0)]$ after Line 11. Furthermore, the label set $\{1, 2\}$ is added to $s_1.sharedlabels$ and $s_2.sharedlabels$ (Line 13) as

Algorithm 4: NEIGHBOREXPLORATION procedure.

```
   Input: A current node v ∈ H, an index i of LMQ.
1  for (u, (v, u).weight) ∈ v.edgesList() and u ≠ v.parent(i)
   do
2      if i ∉ Υ_u then
3          if u ∈ P_i then
4              if (v, u).weight + dist(v) < dist(u) then
5                  u.source(i) ← v.source(i);
6                  u.parent(i) ← v;
7                  UPDATE(P_i, u, (v, u).weight + dist(v));
8                  update u.sharedlabels;
9          else
10             u.source(i) ← v.source(i);
11             u.parent(i) ← v;
12             if v.source is unsafe for u then
13                 add labels from v.source to u.sharedlabels;
14             ENQUEUE(P_i, u, (v, u).weight + dist(v));
15     else if i ∈ u.sharedlabels then
16         if v.source(i) is safe for u then
17             update u.sharedlabels;
18             u.source(i) ← v.source(i);
19             u.parent(i) ← v.id;
20             REMOVE(i, Υ_u);
21             ENQUEUE(P_i, u, (v, u).weight + dist(v));
```

s_1 and s_2 are multi-label merged nodes. Figure 8a depicts the overall state of the merged nodes and LMQs after Line 13.

5.2.2 LMQ-driven label propagation

After the construction of LMQs, the algorithm utilizes them to facilitate the label propagation process, which occurs in several rounds until all LMQs are empty or top-k matching results of Q_P are obtained. The key challenge here is to design a polynomial time exploration algorithm which can produce valid matches for the PTQ with tight performance guarantee.

The overall idea of this step is to sequentially visit each LMQ only once in each round. When a node v is dequeued from a LMQ $\mathcal{P}_i$, first the corresponding label i is added to the label set of the visited node v (Lines 17–18). Next, if the visited node v has ℓ labels (i.e., ℓ-node) and satisfies *certain conditions*, a *result tree* is generated and stored in a heap (Lines 19–23). Lastly, v's neighbors are explored, and a neighbor u is enqueued to $\mathcal{P}_i$ for further visit if it satisfies certain *exploration conditions* (Line 32). We now discuss these steps in detail. For ease of exposition, we begin with the *neighborhood exploration* step.

Neighborhood exploration. Algorithm 4 outlines the procedure for exploration of neighbors of a visited node v. For all neighbors of v except $v.parent(i)$, it checks whether a neighbor u should be enqueued into the LMQ $\mathcal{P}_i$ based on the following three cases.

- **Case 1.** If u is not in $\mathcal{P}_i$ (Lines 9–14), then it indicates that u does not have label i ($i \notin \Upsilon_u \wedge u \notin \mathcal{P}_i$). Hence, we record u's parent node as node v and the source node as $v.source(i)$. Before adding it to $\mathcal{P}_i$, we check whether the new source node of u is safe. If the source node has propagated other labels to u before (i.e., unsafe), we add these labels to u's shared labels. Finally, u is enqueued to $\mathcal{P}_i$ with its distance as the sum of the weight of the edge (v, u) and $dist(v)$.
- **Case 2.** Consider the scenario when u is already in $\mathcal{P}_i$ (Lines 3–8). Then, if a shorter path is discovered for u from v to reach a source node labeled i (Line 4), we replace u's parent node with v and source node with $v.source(i)$. The shared labels of u are updated if the new source node can replace an unsafe source node for label i. Note that since all ℓ LMQs will dequeue and visit elements based on their distance one by one, we can guarantee that a node will be visited. Hence, we can always set the label i only for nodes in the shortest path from all source nodes labeled i.
- **Case 3.** Lastly, if u already has the label i, u can still be explored and added to $\mathcal{P}_i$ again if $u.source(i)$ is unsafe for u, but $v.source(i)$ is safe for u (Lines 15–21). The algorithm utilizes $u.sharedlabels$ to check whether $v.source(i)$ is safe. Specifically, u is enqueued to $\mathcal{P}_i$ again, and its shared labels, parent and source nodes for the label i are updated to ensure that the new source node can eliminate the risk of an invalid candidate match generation.

Note that except for the above three cases, a neighbor u cannot be inserted into an LMQ. Observe that all nodes may be visited by ℓ LMQs at most twice during propagation, which enables the propagation process being conducted in polynomial time complexity. Also, in our algorithm we do not need to store the visited nodes because a node may be enqueued and dequeued more than once from a LMQ.

Example 7 Reconsider Example 6. In the first round of label propagation (depicted in Fig. 8b), the visited node in $\mathcal{P}_1$ is s_1 (Line 16 in Algorithm 3) and its label is set to 1. The neighbor of s_1, node v_1, is added to $\mathcal{P}_1$, and its weight is set to 1 (by executing Lines 9–14 in Algorithm 4). Similarly, the algorithm will iterate $\mathcal{P}_2$ and $\mathcal{P}_3$ and add the corresponding neighbor v_1. Hence, at the end of first round, $\mathcal{P}_1 = [(s_2, 0), (v_1, 1)]$, $\mathcal{P}_2 = [(s_2, 0), (v_1, 1)]$, $\mathcal{P}_3 = [(v_1, 1)]$, s_1 is labeled $\{1, 2\}$ and s_3 is labeled $\{3\}$. Note that since s_1 propagates two different labels to v_1, it is an unsafe source node for v_1. Hence, $v_1.sharedlabels = \{1, 2\}$.

In the next round, the visited nodes in these three LMQs are s_2 and v_1. Hence, at the end of second round (see Fig. 8c), $\mathcal{P}_1 = [(v_1, 1), (v_3, 1)]$, $\mathcal{P}_2 = [(v_1, 1), (v_3, 1)]$, $\mathcal{P}_3 =$

$[(s_1, 2), (v_3, 2)]$, $s_2.sharedlabels = \{1, 2\}$, and the label 3 is added to v_1.

Similarly, Fig. 8d, e illustrate the label propagation process and the content of the LMQs in the next two rounds. Observe that in the third round, v_1 and s_1 have all the three labels (i.e., they are ℓ-nodes). Furthermore, as a neighbor of v_1, s_3 will be explored by $\mathcal{P}_1$ and $\mathcal{P}_2$ for the first time. Because these two related labels are all from s_1, $v_3.sharedlabels = \{1, 2\}$. Also, v_3 ignores the exploration from v_1 because it is already in $\mathcal{P}_1$ and $\mathcal{P}_2$ with a shorter distance (Line 4). Observe that in the fourth round, v_3 is visited by all LMQs, i.e., it is a ℓ-node.

Theorem 4 *The time and space complexities of the label propagation process in* PO- PANDA *algorithm are* $O(2\ell(|V_H| log|V_H| + |E_H|))$ *and* $O(\ell|V_H|)$*, respectively.*

Proof We need only ℓ LMQs to propagate ℓ labels to all nodes from the set of merged nodes. During the propagation process, a node can be added into a LMQ $\mathcal{P}_i$ if and only if the node has no corresponding label i or the label is a shared label for a source node. Thus, each node is inserted into a LMQ at most twice. In other words, a LMQ will traverse the whole graph at most twice. Each traversal for shortest paths from a node costs $O(|V_H|log|V_H| + |E_H|)$, and there are ℓ LMQs in total. Hence, the total time complexity of the label propagation process is $O(2\ell(|V_H|log|V_H| + |E_H|))$.

For each node in a LMQ, we will store the parent node, source node, and shared labels information. Suppose it takes $O(1)$ space for this information. Then, the space complexity is $O(\ell|V_H|)$ because there are at most $|V_H|$ nodes in a LMQ. □

Top-k matching results generation. During label propagation, when a visited node v is a ℓ-node (Line 19 in Algorithm 3), it means that at least one merged node in each S_i is reachable from v. Recall that we record all source nodes for all labels of v. Hence, a *result tree* T can be constructed based on the ℓ-node v and comprises all its source nodes along with their label propagation paths. The key issue here is to determine whether T is *valid* w.r.t partial topology-based matching of Q_P. Informally, a result tree is *valid* iff every source node's inner graph contains at least one nonoverlapping matching subgraph for every multi-label source node of the tree. Specifically, we can detect it by checking if v does not have any shared label (i.e., $v.sharedlabels = \emptyset$), indicating that each source node is matched to only one query component. Otherwise, for each unsafe source node, we need to check whether there exists at least one nonoverlapping matching subgraph for corresponding query component in its inner graph. For example, consider the third round of label propagation in Fig. 8d. Observe that both v_1 and s_1 are ℓ-nodes and $v_1.sharedlabels = s_1.sharedlabels = \{1, 2\}$. Hence, s_1 and v_1 are unsafe (Line 20 of Algorithm 3). After checking I_{s_1}, we observe that all matching subgraphs of q_1 and q_2 are overlapping. Hence, the result tree rooted at v_1 or s_1 is invalid. Now consider Fig. 8e. In this round, v_3 is a ℓ-node and the corresponding result tree is valid because $s_2 = v_3.source(1) = v_3.source(2)$ and I_{s_2} contains nonoverlapping matching subgraphs for q_1 and q_2.

During the construction of a valid result tree T, the algorithm keeps track of its *total cost*, denoted as $cost(T)$, which is sum of the total cost of edges connecting the merged nodes in T and the cost of inner graphs of these merged nodes. Formally, let there be n merged nodes in T. Then,

$$cost(T) = \sum_{e \in E_T} e.weight + \sum_{a=1}^{n} cost(I_a) \quad (2)$$

The first component of the equation computes the total cost of edges in T, and the second one computes the total cost of the inner graphs of all merged nodes. Note that a merged node has a nonzero cost of its inner graph in two situations. First, if a ℓ-node of a valid tree has shared labels, then for each unsafe source node, there exists at least one nonoverlapping matching subgraph for corresponding query component in its inner graph. These matching subgraphs for different query components are nonoverlapping and connected. Hence, cost of the shortest path connecting them is the cost of the merged node's inner graph. Second, if a ℓ-node does not have shared labels, it means that each merged node of the tree is to be matched for only one query component. But if a merged node of the tree is originally a multi-label merged node, then its inner graph may contain some edges that do not belong to the corresponding matching subgraphs. Then the cost of the inner graph is the shortest path involving these edges that connect the matching subgraphs. Observe that if we replace the merged nodes of the tree by its matching subgraphs with their connected shortest paths, $cost(T)$ is equal to $cost(M)$ in Eq. 1.

After the total cost of T is computed, it is added into the heap $TreeHeap$ sorted by increasing order of its cost (Lines 22–23 of Algorithm 3). Note that T may contain a set of matching results of Q_P if and only if there is at least one merged node in T having more than one nonoverlapping matching subgraphs in its inner graph.

Observe that $cost(I_a)$ influences the ranking of result trees as it may not always be zero. Hence, we need to ensure that the cost of inner graphs of merged nodes cannot adversely influence the "quality" of the result trees. To this end, we continue exploration of H by label propagation from all LMQs (Lines 25–28 of Algorithm 3). Specifically, we record $cost(I_a)$ of the worst trees (k-th tree) in the $TreeHeap$. Let the cost be δ. Then, all LMQs are allowed to propagate its label to nodes up to δ distance. If there are other valid result trees found during this process, we insert it in the $TreesHeap$. Finally, we return the top-k result matches by retrieving top-k result

trees from $TreesHeap$ and replace their merged nodes with corresponding inner graphs (Lines 29–31 of Algorithm 3).

Example 8 Reconsider Fig. 8. In the fourth round of label propagation, we found a valid result tree T as depicted in Fig. 8 (h, top). Here $cost(T) = 3 + cost(I_{s_2}) + cost(I_{s_3}) = 3 + 1 + 0 = 4$. Since $cost(I_{s_2}) = 1$, all three LMQs are further explored by label propagation for $\delta = 1$ (i.e., explore all nodes with distance 2 in $\mathcal{P}_1$ and $\mathcal{P}_2$, and with distance 3 in $\mathcal{P}_3$) as shown in Fig. 8f, g. In this process, another valid result tree T' is discovered as shown in Fig. 8 (h, below) with $cost(T') = 4 + cost(I_{s_1}) + cost(I_{s_2}) + cost(I_{s_3}) = 4+1+0+0 = 5$. Note that the inner graph of s_1 needs a path from v_2 to v_5 or from v_2 to v_6 for the matching subgraph of q_2 (Fig. 5b). Since $cost(T) < cost(T')$, T is returned as the top-1 result after replacing the merged nodes with corresponding inner graphs (Fig. 8i).

Theorem 5 *The* PO- PANDA *algorithm guarantees that all results generated by it for the partial topology-based network search problem are correct.*

Proof We suppose that M is an incorrect match produced by the PO- PANDA algorithm. In the definition of partial topology-based matching problem (Definition 1), incorrectness of M can be attributed to three possible reasons. Firstly, M is disconnected which contradicts the definition of a tree because M is a tree spanning several source nodes. Secondly, there is no isomorphic match in M for a query component. Any source node in M for a label of ℓ-node must contain at least one subgraph which is isomorphic to the label's corresponding query component. For this reason, there exists a label which is not contained in the label set of the ℓ-node of M. But this is contradictory to the precondition of producing a result tree in Line 19 of Algorithm 3. Lastly, all isomorphic matches are overlapping in M for a certain pair of query components. In this case, the ℓ-node of M must have shared labels for a source node. That is, the source node is unsafe, which is contradictory to the precondition of producing a result tree in Line 20 of Algorithm 3. Hence, all matches generated by PO- PANDA are correct. □

Corollary 1 *The time complexity of* PO- PANDA *algorithm is* $O(2\ell(|V_H|log|V_H| + |E_H|) + \ell\mathbb{I} + |V|^2)$, *where* $\mathbb{I}$ *is the time complexity of the subgraph matching algorithm.*

Proof The time complexity can be divided into three steps. In the first step, we need to compute the candidate matching subgraphs using a subgraph matching algorithm. Hence, the complexity is $O(\ell\mathbb{I})$ for ℓ query components. The second step is to generate the merged graph using Algorithm 1, which takes $O(|V|^2)$. The third step is the label propagation process which takes $O(2\ell(|V_H|log|V_H| + |E_H|))$ (Theorem 4). Hence, the overall complexity is $O(2\ell(|V_H|log|V_H| + |E_H|) + \ell\mathbb{I} + |V|^2)$. □

Theorem 6 *The* PO- PANDA *algorithm returns an approximate solution for Minimal Partial Topology-based Matching problem with an approximation factor of* $O(\ell)$, *when* $\ell \geq 3$.

Proof Suppose $Tree_o$ is the minimal one for all valid result trees for a partial topology-based matching problem, and $Tree_{st}$ is the minimal cost tree returned by Algorithm 3. Therefore, $\exists(v_1, \ldots, v_\ell) \in (S_1, \ldots, S_\ell)$ and $v_i \in Tree_o$ where S_i is the set of all merged nodes whose inner graph is isomorphic to query component i. Suppose $root_o$ is the root node of $Tree_o$. Further, $\forall i \in \{1, \ldots, \ell\}$ there exists exactly one path $p(root_o, v_i) \subset Tree_o$, denoted by $p^o_{root_o, v_i}$. For each $p^o_{root_o, v_i} \in Tree_o$, $\gamma(p^o_{root_o, v_i}) \leq \gamma(Tree_o)$ where γ is a cost function. For all $Tree_u \in TreeHeap$, Algorithm 3 computes a tree $Tree_u$ with u as the root node and $\gamma(Tree_{st}) \leq \gamma(Tree_u)$. For $Tree_{st}$, $\exists(v_1, \ldots, v_\ell) \in (S_1, \ldots, S_\ell)$ and $v_i \in Tree_{st}$ where S_i is the set of all merged nodes whose inner graph is isomorphic to the query component i. The cost of inner graph of all merged nodes cannot affect the quality because of Line 26 in Algorithm 3. Also, $\forall i \in \{1, \ldots, \ell\}$ there exists exactly one path $p(u, v_i) \subset Tree_{st}$, denoted by p_{u,v_i}. To build $Tree_u$, Lines 4–8 of Algorithm 4 find ℓ-node u as root node by searching along shortest paths from $S_i, \forall i \in \{1, \ldots, \ell\}$. Thus, $\exists j \in \{1, \ldots, \ell\}$ such that $\forall i, \gamma(p_{u,v_i}) \leq \gamma(p^o_{root_o, v_j})$. Hence, $\gamma(Tree_{st}) \leq \gamma(Tree_v) \leq \sum_{i\in\{1,\ldots,\ell\}} \gamma(p_{u,v_i}) \leq \ell\gamma(p^o_{root_o, v_j}) \leq \ell\gamma(Tree_o)$. □

Remark Similar to SEN- PANDA, the PO- PANDA algorithm can easily be reduced to handle scenarios where overlapping matching subgraphs for different query components are allowed. The formal description of this modified algorithm is given Supplementary Material B.

6 Simulation-based optimization

Although the PO- PANDA algorithm improves the efficiency of searching a merged graph for top-k matches, matching subgraph generation in the first phase is expensive due to subgraph isomorphism test. In this section, we present an optimization strategy based on *subgraph simulation* to alleviate this problem.

We begin by introducing the concept of *subgraph simulation*. Given a graph $G = (V, E)$ and a query graph $Q = (V_q, E_q)$, the goal of *subgraph simulation* [14] is to find each matching subgraph $g = (V_g, E_g)$, $g \subseteq G$, denoted by $Q \prec_s g$, such that there exists a binary match relation $R \subseteq V_q \times V_g$ satisfying the following conditions: (a) For each $(u, v) \in R$, v contains all labels of u, i.e., $L(u) \subseteq L(v)$; (b) for each node $u \in V_q$, there exists $v \in V_g$ such that $(u, v) \in R$; and (c) for each edge $(u, u') \in E_q$, there exists an edge $(v, v') \in E_g$ such that $(u', v') \in R$.

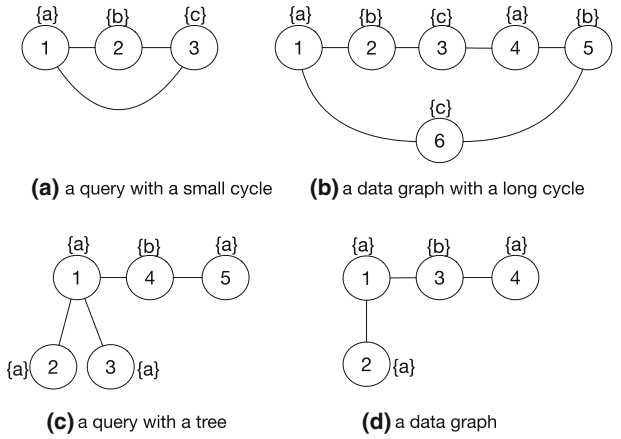


Fig. 9 Differences between simulation and isomorphism. The identifier of a node is shown inside the node and its label set in its vicinity

Example 9 Consider the query in Fig. 9a on the network in Fig. 9b. Based on subgraph simulation, there is a binary match relation $R = \{(1, 1), (1, 4), (2, 2), (2, 5), (3, 3), (3, 6)\}$ for this query. Similarly, a binary match relation $R = \{(1, 1), (2, 2), (3, 2), (4, 3), (5, 4)\}$ in Fig. 9d can be found for the query in Fig. 9c. However, observe that there does not exist any isomorphic match to these queries.

Compared to subgraph isomorphism, subgraph simulation only needs to find a binary match relation instead of a bijection mapping. Consequently, the requirements of subgraph simulation are weaker than subgraph isomorphism and can be solved in cubic time. In fact, all matching results of subgraph isomorphism are contained in the results of subgraph simulation [19] (i.e., if $q \prec G$, then $q \prec_s G$). Hence, we propose a *simulation-based pruning strategy* to improve the performance of PO- PANDA without compromising on the quality of results. Specifically, in the matching subgraph generation step, we replace subgraph isomorphism-based matching with subgraph simulation (Line 3 in Algorithm 3) and enhance the exploration algorithm (Algorithm 4) in the third step to realize our simulation-based pruning strategy. We now elaborate on these two enhancements.

6.1 Simulation-based connected subgraph generation

First, we compute the candidate matches of each query components via subgraph simulation. Since the topology of simulation-based matches can be significantly different from a connected query component, next we find all *connected simulation subgraphs* by breadth-first search using the simulation matching pairs (v, v_{sim}), where $v \in V_{q_i}$ and $v_{sim} \in V$. Note that a set of connected simulation subgraph (possibly overlapping) can be represented by merged nodes similar to the way isomorphic matching subgraphs are represented in PO- PANDA. However, now the inner graph of a merged node may not necessarily contain isomorphic matches to a query component.

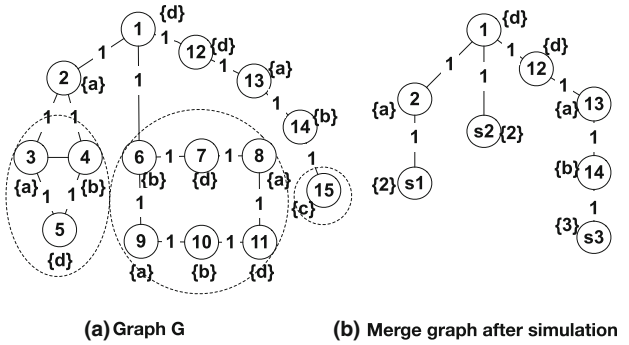


Fig. 10 Simulation-based optimization

Example 10 Consider the network in Fig. 10a. Suppose a PTQ Q_P comprises of the components q_2 and q_3 in Fig. 5a. Observe that there are two connected simulation subgraphs in Fig. 10a (highlighted by dotted cycle) that match q_2 via subgraph simulation but only one of them is isomorphic to it. The resultant merged graph is shown in Fig. 10b. Observe that the inner graph of s_2 does not have any isomorphic match to q_2.

6.2 Enhanced label propagation

As the inner graph of a merged node may not contain any matching subgraph via subgraph isomorphism, we need to enhance the exploration process in Algorithm 4 to ensure that the merged nodes of a result tree are valid. This requires that an inner graph should contain at least one subgraph which is isomorphic to the corresponding query component. Specifically, we perform subgraph isomorphism test on the inner graph of a merged node on demand during the label propagation process. Note that the inner graph of a merged node is usually significantly smaller than the original network, and in practice only a subset of all merged nodes need to be tested. Hence, as we shall see in Sect. 7, the cost of such test is significantly less in practice.

Recall from Algorithm 4, given a LMQ $\mathcal{P}_i$, a path from a source node s_a will terminate its label propagation when it confronts a node u satisfying any one of the following cases. (1) $i \notin \Upsilon_u$, $u \in \mathcal{P}_i$ and the path from s_a to u is longer than that from $u.source(i)$. (2) u has already owned label i and $u.source(i)$ is safe (i.e., $i \in \Upsilon_u$ and $i \notin u.sharedlabels$). (3) u has already owned i, $u.source(i)$ is unsafe, and s_a is also an unsafe source node for u. Hence, we need to check whether the source node contains isomorphic matches to corresponding query component q_i before terminating propagation of i based on the above three cases.

Suppose s_b is the current source node ($s_b = u.source(i)$) for label i, and s_a is the new source node from v. We enhance

Algorithm 5: Enhanced NEIGHBOREXPLORATION procedure for SIMPO- PANDA.

```
   Input: A current node v ∈ H, an index i of LMQ.
1  for (u, (v, u).weight) ∈ v.edgesList() and u ≠ v.parent(i) do
2      if i ∉ Υ_u then
3          if u ∈ P_i then
4              if (v, u).weight + dist(v) < dist(u) then
5                  u.source(i) ← v.source(i);
6                  u.parent(i) ← v;
7                  UPDATE(P_i, u, (v, u).weight + dist(v));
8                  update u.sharedlabels;
9              else if SIMCHECK(v.source(i), u.source(i), i) then
10                 u.source(i) ← v.source(i);
11                 u.parent(i) ← v;
12                 UPDATE(P_i, u, (v, u).weight + dist(v));
13         else
14             u.source(i) ← v.source(i);
15             u.parent(i) ← v;
16             if v.source is unsafe for u then
17                 add labels from v.source to u.sharedlabels;
18             ENQUEUE(P_i, u, (v, u).weight + dist(v));
19     else if i ∈ u.sharedlabels then
20         if v.source(i) is safe for u then
21             update u.sharedlabels;
22             u.source(i) ← v.source(i);
23             u.parent(i) ← v.id;
24             REMOVE(i, Υ_u);
25             ENQUEUE(P_i, u, (v, u).weight + dist(v));
26         else if SIMCHECK(v.source(i), u.source(i), i) then
27             u.source(i) ← v.source(i);
28             u.parent(i) ← v;
29             ENQUEUE(P_i, u, (v, u).weight + dist(v));
30     else if SIMCHECK(v.source(i), u.source(i), i) then
31         u.source(i) ← v.source(i);
32         u.parent(i) ← v;
33         ENQUEUE(P_i, u, (v, u).weight + dist(v));
```

the aforementioned three exploration cases to ensure that propagation from v will continue by enqueuing or updating u in $\mathcal{P}_i$ if and only if (a) the shortest path from u to connect a matching subgraph in the inner graph of s_a is shorter than the path to connect s_b or (b) s_b does not have any isomorphic matches but s_a has. We check s_a before s_b so that if there is no isomorphic matches in s_a, we can directly ignore it. Otherwise, if s_b does not contain an isomorphic match or the shortest path from the matching subgraph in the inner graph of s_a to u is shorter than that of s_b, we replace $u.source(i)$ with s_a, update its parent node based on the path from s_a, and enqueue u to $\mathcal{P}_i$ again.

Additionally, we also need to check all source nodes via subgraph isomorphism when we find a ℓ-node (Line 21 in Algorithm 3). To avoid duplication of checking a source node, we store all previously checked results of source nodes in a hash map in the memory. The formal procedure of this enhanced label exploration is outlined below.

Algorithm 6: The SIMCHECK procedure.

```
   Input: A merged node s_a, A merged node s_b, The propagated
          label i.
1  if I_{s_a} contains subgraph isomorphism for q_i then
2      if I_{s_b} contains subgraph isomorphism for q_i then
3          if the path from the isomorphism in s_a is shorter than s_b
           then
4              return True;
5      else
6          return True;
7  return False;
```

Algorithm The enhanced neighborhood exploration procedure to realize simulation-based optimization is outlined in Algorithm 5. It enhances Algorithm 4 by performing subgraph isomorphism test on the inner graphs of source nodes for the three cases discussed in Sect. 6. Specifically, Lines 9–12, 30–33, and 26–29 realize Cases 1, 2, and 3, respectively.

Suppose s_b is the current source node ($s_b = u.source(i)$) for label i, and s_a is the new source node from v. Algorithm 6 shows the subgraph isomorphism test on the inner graphs of these two source nodes. It will return `True` if and only if s_b does not have any isomorphic matches but s_a has or the shortest path from u to connect a matching subgraph in the inner graph of s_a is shorter than the path to connect s_b. Observe that it checks s_a before s_b (Lines 1–2) so that if there is no isomorphic matches in s_a, we can directly ignore it. Otherwise, if s_b does not contain an isomorphic match for q_i (Line 6) or the shortest path from the matching subgraph in the inner graph of s_a to u is shorter than that of s_b (Lines 3–4), it returns `True`. Observe that if it returns `True`, the propagation from v will continue by enqueuing or updating u in $\mathcal{P}_i$ (in Algorithm 5). This enhancement ensures that all merged nodes of a result match are valid, which requires that the inner graph should contain at least one subgraph which is isomorphic to the corresponding query component.

Example 11 Reconsider the merged graph in Fig. 10b. During the label propagation process in $\mathcal{P}_2$, label 2 will be first propagated from s_2 to v_1. Then, when it is propagated from s_1, v_1 already has the label 2 (Case 2). However, before terminating label 2's propagation along the paths from s_1, we shall test whether there are isomorphic matches in the inner graphs of s_1 and s_2. Observe that s_1 has but s_2 does not. So we enqueue v_1 to $\mathcal{P}_2$ again. These two labels are propagated until they confront at v_{12} and a valid tree will be generated at this node.

Table 2 Datasets

Dataset	Nodes	Edges	Distinct labels	Diameter
Yeast	2361	11,855	13	11
Cit-HepPh	34,546	420, 899	124	12
Amazon	403,394	2443K	22,926	21
Pokec	1632K	22,301K	61,342	11
Friendster	65,608K	1,806,067K	957,154	32

Remark Note that the simulation-based optimization algorithm has the same approximation bound as PO- PANDA (Theorem 6). In fact, this strategy may increase the possibility of finding better quality result trees compared to PO- PANDA because the inner graphs in this case are usually larger than those of isomorphic-based matches, which will shorten the distance between merged nodes. Lastly, similar to PO- PANDA, SIMPO- PANDA can easily be extended to handle scenarios where the matching subgraphs of different query components are overlapping.

7 Performance study

In this section, we report the performance of our proposed algorithms. All algorithms are implemented in Java. All experiments are performed on a machine with Intel four Core i7 3.7Ghz CPU and 32 GB RAM memory, running Ubuntu Linux 12.04.

7.1 Experimental setup

Datasets. We experimented on five publicly available real datasets from different fields with different characteristics as listed below. Table 2 summarizes their characteristics. These datasets are downloaded from [18].

1. **Citation Network:** High-energy physics theory citation network (*Cit-HepPh*). The data cover papers in the period from January 1993 to April 2003 (124 months). Each paper is assigned a label of the publication time (month-year).
2. **Co-purchase Network:** It was collected by crawling *Amazon* website on June 01, 2003. All nodes are labeled by their product categories such as business, history, and general. A node can have many different labels if it belongs to many categories. The average number of products in a category in the dataset is 41, and the average number of labels for a node is 3.
3. **Social Network:** *Pokec* is the most popular on-line social network in Slovakia. Profile data contain gender, age, hobbies, interest, profession etc. We use profession information of each user as labels.
4. **PPI Network:** *Yeast* protein interaction network[2] is a public available large PPI network with 13 known protein cliques. This classification is used as labels.
5. **Communities-based Social Network:** *Friendster* is an on-line gaming network. We use community ground truth of a node as labels.

Query set. Similar to [17,19], we generate partial topology queries by extracting nonoverlapping subgraphs from the target networks randomly and ensuring sufficient diversity in topological structures of the query components (simple path, tree, cycle, and long cycle). We use the templates of the benchmark in [20] to simulate real queries. The query generation is controlled by:

- Number of query components, denoted by ℓ;
- Maximum number of nodes in each query component q, denoted by $|V_q|$;
- Maximum diameter of each query component q, denoted by D_q. Specifically, we randomly select D_q within $(0, |V_q|)$ for each q to generate queries with different diameters.

Unless specified otherwise, each query workload refers to 100 randomly generated queries with different topologies. We run each query 5 times and report the average running time. If a query takes more than one hour to execute, we terminate it.

Algorithm To the best of our knowledge, no existing technique addresses the partial topology-based query processing problem. Hence, we enhance two state-of-the-art network querying efforts that are most germane to our problem in order to address our partial topology-based network search problem. Specifically, we study the following algorithms. The subgraph isomorphism is implemented using [4] for all relevant algorithms.

1. SEN- PANDA: Algorithm 2 as proposed in Sect. 4. We use [5] for GST implementation.
2. PO- PANDA: Algorithm 3 as proposed in Sect. 5.

[2] http://vlado.fmf.uni-lj.si/pub/networks/data/bio/Yeast/yeast.zip.

3. SIMPO- PANDA: Simulation-based optimized PO- PANDA algorithm as presented in Sect. 6.
4. NEMA+: The approximate graph search algorithm in [17], which is similar to LOOPBP [28]. Note that NEMA demands a connected query as input and returns the top-1 matching subgraph. Hence, we extend it so that a new query is created from a partial topology query by inserting a set of implicit edges between query components, where each implicit edge bridges a pair of nodes from different query components [28]. We also enhance it to return top-k results.
5. TORQUE+: A topology-free protein network matching algorithm using integer programming [1]. The goal of this algorithm is to find matching sets of proteins (keywords) for the input set that span connected regions in the network. We enhance it by implementing it on top of our PANDA framework (i.e., adding matching subgraph generation and merged graph generation steps).
6. SP+: Based on the PANDA framework, we first find all isomorphic subgraphs of each query component and build a merged graph. Then, we utilize classic Dijkstra's algorithm to connect merged nodes for all query components by calculating shortest paths between these merged nodes.

Performance metrics for ranked results. We adopt the following metrics to evaluate the quality of ranking.

1. *Precision at* k ($P@k$): The percentage of top-k answers that satisfies all requirements of a partial topology-based matching.
2. *Mean average precision* ($MAP@k$): It is measured as

$$MAP@k = \frac{\sum_{i=1}^{|Q_P|} AveP(Q_{P_i})}{|Q_P|}$$

where $AveP = \frac{\sum_{i=1}^{k} P(i) \times rel(i)}{R}$, rel is an indicator function whose value is 1 if the answer at rank i is a correct answer, zero otherwise, and R is the number of the answers.

3. *Normalized Discounted Cumulative Gain* ($NDCG$): It is measured as

$$NDCG = \left(\sum_{i=1}^{|Q_P|} \frac{DCG_{Q_{P_i}}}{iDCG_{Q_{P_i}}} \right) / |Q_P|$$

where

$$DCG = \sum_{i=1}^{k} \frac{2^{r_i} - 1}{log_2(i+1)}$$

$$iDCG = \sum_{i=1}^{k} \frac{2^{ir} - 1}{log_2(i+1)}$$

$r_i = 1/c$, c is the cost of the returned answer, and $ir = \frac{1}{\ell - 1}$, which is the ideal cost of a match for a query Q_{P_i} with ℓ query components. Observe that $NDCG$ can capture not only the validity of the results, but also their cost, whereas $MAP@K$ and $P@K$ only measure the results' validity.

7.2 Experimental results

We now investigate the performance of our proposed algorithms from a variety of aspects and report the results here.

7.2.1 Effect of k

First, we investigate the efficiency and effectiveness of the algorithms with the increase in k. We set $\ell = 3$, $|V_q| = 4$, and vary k from 1 to 100. The results are reported in Fig. 11 and Table 3. We can make the following observations.

First, SIMPO- PANDA demonstrates the best efficiency and is at least an order of magnitude faster than others for larger networks. This is because it avoids the time-consuming subgraph isomorphism check for every query component, and its search process prunes a lot of simulation results. Even if there is a need to check isomorphism in an inner graph of a merged node, as mentioned earlier, its size is significantly smaller than the whole network.

Second, observe that SEN- PANDA is worse than other isomorphism-based techniques because it needs to invoke the GST algorithm several times. In contrast, PO- PANDA is more efficient because it can find top-k matches in a progressive manner by invoking the search process only once, highlighting the benefit of our label propagation mechanism. There is only a little quality loss by PO- PANDA and SIMPO- PANDA compared to SEN- PANDA. That is, both these techniques do not adversely affect the result quality significantly.

Third, for the relatively small *Yeast* network, TORQUE+ can efficiently find matches with high quality as it is specifically designed for PPI networks. However, it is costlier compared to other algorithms for the *Citation* and *Amazon* networks due to expensive linear programming-based approach. Besides, the quality of its results is lower than our algorithms, especially for large networks and k because TORQUE+ does not allow a node (merged node) to match to multiple query nodes (query components).

Fourth, although NEMA+ is relatively more efficient than several techniques, its result quality is the worst. Majority of them are invalid because topology of several query components are mismatched. NEMA+ does not find isomorphic

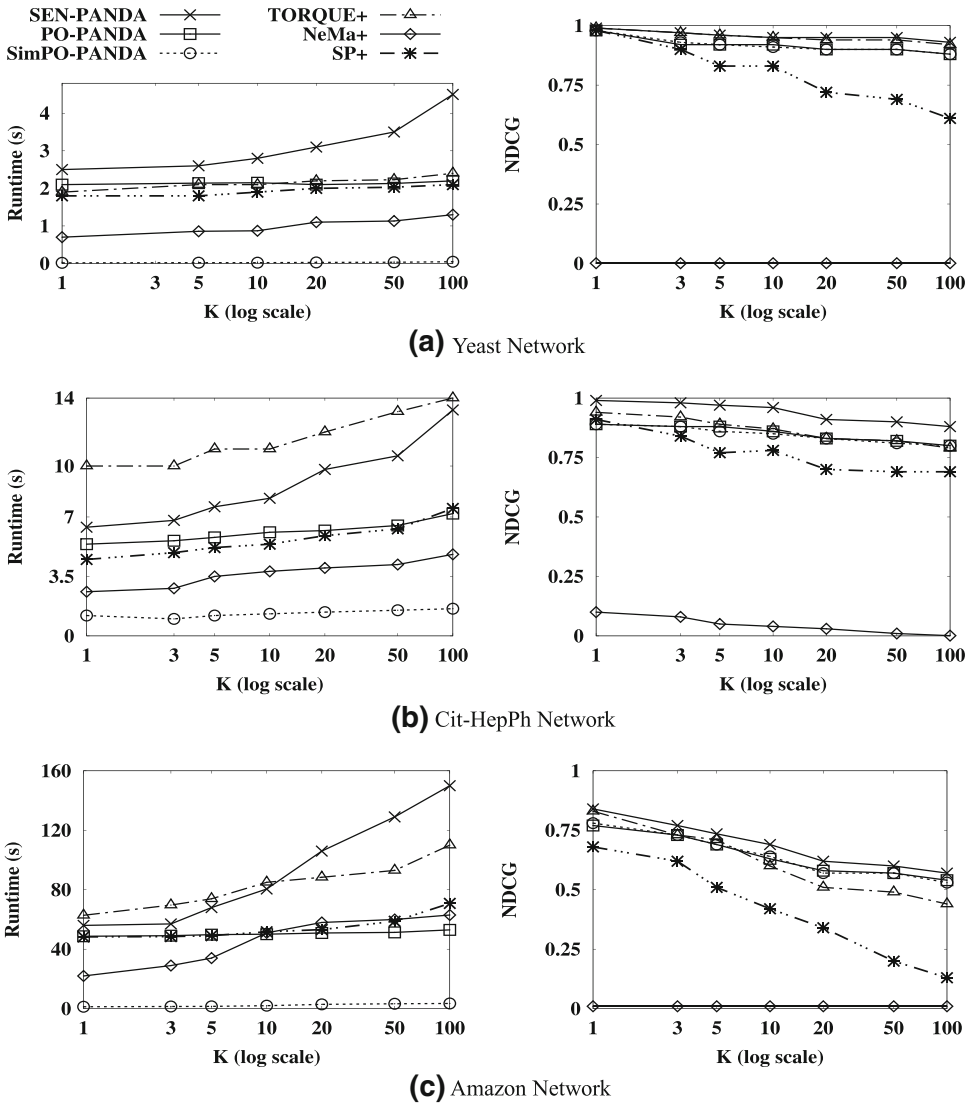


Fig. 11 Effect of k on runtime and result quality

Table 3 The variance (minimum, maximum) of running time (in second) of different techniques on the *Amazon* network for different k

	$k=1$	$k=3$	$k=5$	$k=10$	$k=20$	$k=50$	$k=100$
SEN- PANDA	(21, 323)	(22, 400)	(30, 440)	(35, 450)	(43, 480)	(55, 610)	(60, 720)
PO- PANDA	(5, 130)	(5, 131)	(5, 131)	(6, 132)	(6, 134)	(7, 135)	(8, 140)
SIMPO- PANDA	(0.5, 40)	(0.5, 41)	(0.5, 41)	(0.5, 45)	(0.6, 50)	(0.8, 60)	(0.9, 80)
TORQUE+	(20, 420)	(24, 430)	(30, 433)	(38, 440)	(48, 456)	(53, 500)	(65, 610)
NEMA+	(10, 32)	(12, 40)	(15, 55)	(20, 70)	(22, 75)	(25, 80)	(27, 108)
SP+	(4, 110)	(4, 110)	(5, 118)	(7, 125)	(9, 140)	(18, 178)	(30, 220)

matches for each query component. Instead, it searches for approximate matches based on neighbors' information. In fact, we shall further validate poor result quality of NEMA+ by undertaking a user study in Sect. 7.2.5. Besides, although SP+ is more efficient than SEN- PANDA and TORQUE+, its NDCG scores are relatively poor, especially for larger k.

Fifth, the maximum $P@k$ and $MAP@k$ values of NEMA+ are 0.05 and 0.02, respectively, for all values of k in all datasets. On the other hand, result quality returned by other algorithms is significantly superior (i.e., $MAP@k = 1$, $P@k = 1$ for all k).

Lastly, Table 3 reports the variance of running times (minimum and maximum time taken for all runs) for different techniques on the *Amazon* network. Similar to existing subgraph matching algorithms [1,4,19], the variance is large as the query nodes' labels, the structure of query

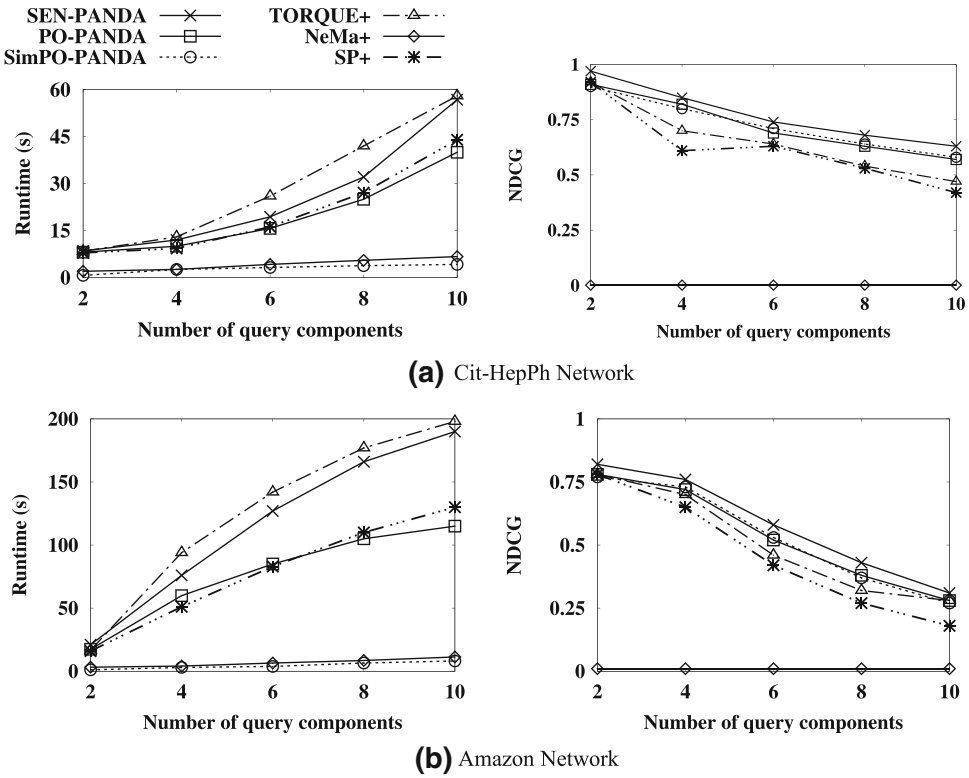


Fig. 12 Effect of ℓ

components, and the processing order of the query nodes have large impact on the running time. Nevertheless, consistent with the aforementioned results, SIMPO- PANDA is superior to other approaches due to its ability to prune a large number of candidate paths and reduction in the number of subgraph isomorphism test. As we shall see below, this same phenomenon holds in subsequent experiments.

7.2.2 Effect of ℓ

Next, we investigate the impact of number of query components ℓ in Q_P. Observe that if $\ell = 1$, all algorithms reduce to a subgraph matching algorithm (i.e., $VF2$ [4] in our case). Hence, we vary ℓ from 2 to 10 and set $k = 3$. Other parameters remain the same as preceding experiments. Figure 12 reports the running times and result quality. Obviously, all algorithms consume more time as more number of query components needs to be matched. We can make the following observations.

First, TORQUE+ has the worst running time for both datasets. Second, although SEN- PANDA has the best result quality, its performance suffers with the increase in ℓ as the cost of subgraph isomorphism-based matching increases and there are potentially more overlapping matching subgraphs. In fact, there are nearly 70% to 80% cases for $\ell > 4$ when these two algorithms fail to finish evaluation within one hour. So we only report the average running time of queries that can finish within an hour. Third, SIMPO- PANDA has the best efficiency and is several orders of magnitude faster than its counterparts for larger network. Lastly, SP+ is slightly more efficient than PO- PANDA, but it gets worse with increasing ℓ as more shortest paths need to be computed.

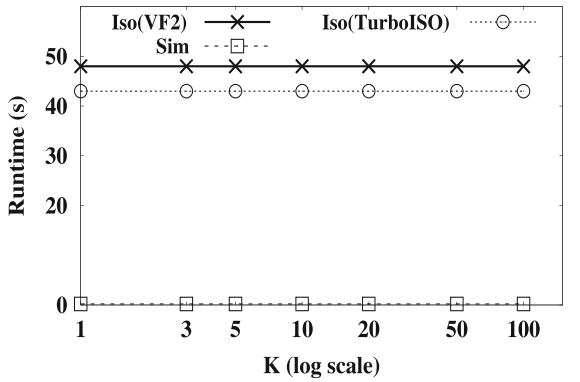
As far as result quality is concerned, the NDCG scores of all algorithms decrease with increasing ℓ. Observe that there is little difference between PO- PANDA, SIMPO- PANDA, and SEN- PANDA w.r.t result quality. However, the result qualities of SP+ and TORQUE+ are consistently worse than our algorithms as the number of query components increases. Also, NEMA+ hardly finds valid matches for PTQs, emphasizing the fact that it cannot be deployed effectively to solve the partial topology-based network search problem.

7.2.3 Performance of different components

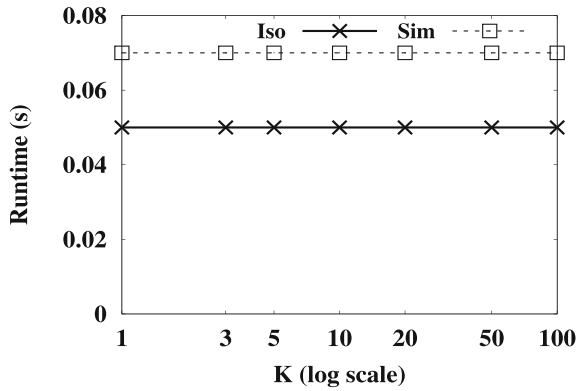
Figure 13 plots the performances of the three components in the PANDA framework on the *Amazon* dataset. We do not include NEMA+ in this set of experiments as it does not follow our framework. Also, in order to study the impact of the choice of subgraph isomorphism technique on our framework, we include a recently proposed subgraph isomorphism algorithm called TURBOISO [11] for the first and third components. Specifically, TURBOISO has superior performance than VF2 as it has a better strategy for pruning embedding candidates.

For the matching subgraph generation component, as expected subgraph simulation is significantly faster than subgraph isomorphism (Fig. 13a). Although TURBOISO has better performance than VF2, both are significantly worse than the subgraph simulation algorithm. Hence, the first component of SIMPO- PANDA is significantly faster than that of SEN- PANDA or PO- PANDA. For the merged graph generation component (Fig. 13b), subgraph simulation-based generation (SIMPO- PANDA) is slightly costlier than its isomorphism-based counterpart (VF2-based) because simulation-based matches are usually larger than isomorphic matches. Note that the running time of this component is not affected by the choice of isomorphism algorithm because the resultant matching subgraphs are same. Furthermore, the response times of these two components are unaffected by k as they are independent of top-k result generation.

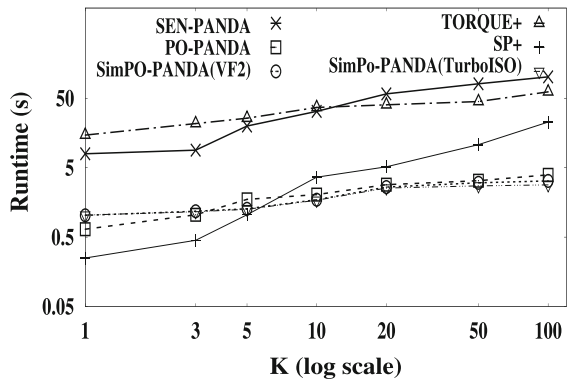
In contrast, the final component is sensitive to the parameter k, especially for SEN- PANDA and TORQUE+. SEN- PANDA generates a set of SMGs which increases the search time dramatically for some cases. Poor efficiency of TORQUE+ is attributed to the increasing cost of linear programming. Although SP+ uses shortest path as a heuristic for finding matches, the time to compute it increases with k. It invokes the shortest path algorithm $\ell - 1$ times in order to find a match. Our PO- PANDA and SIMPO- PANDA are faster than other techniques because they find all matching candidates by searching the merged graph only once using the proposed label propagation technique, which is not very sensitive to k. Observe that SIMPO- PANDA is slightly worse than PO- PANDA for small k due to subgraph isomorphism validation time required for connected simulation subgraphs during label propagation, which are typically larger in size. Also, since SIMPO- PANDA is the only algorithm that needs to conduct isomorphism checking for inner graphs of some candidate merged nodes, different subgraph isomorphism algorithms may affect its running time. Fortunately, Fig. 13c shows that its running time is similar for both TURBOISO-based and VF2-based solutions because the number of candidate merged nodes that are checked for subgraph isomorphism is often small in practice.



(a) Matching subgraph generation.



(b) Merged graph construction.

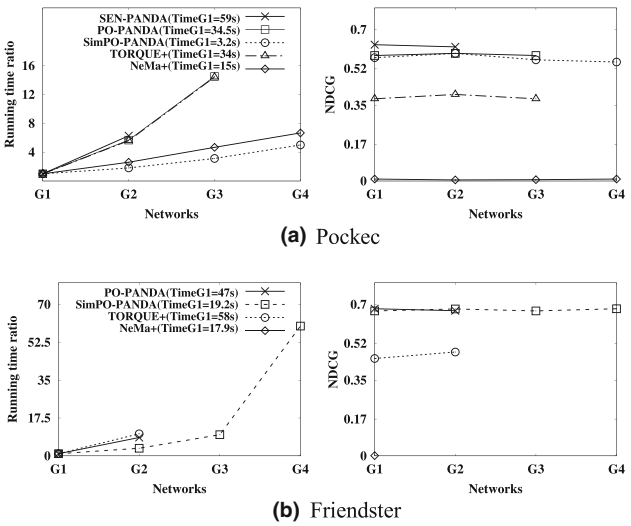


(c) Partial topology-based matching results generation.

Fig. 13 Performance of different components of PANDA framework

7.2.4 Scalability

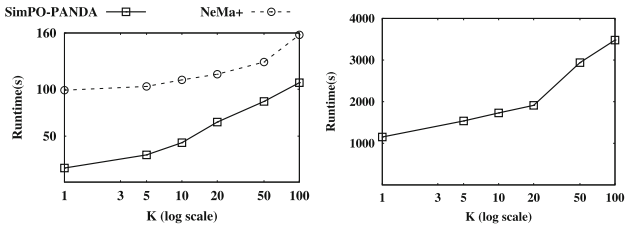
Figure 14 plots the running time and quality of results on larger *Pokec* and *Friendster* networks. Specifically,



(a) Pockec

(b) Friendster

Fig. 14 Scalability of different algorithms



(a) Pockec G_4 **(b)** Friendster G_4

Fig. 15 Effect of k on scalability

we randomly initialize a subgraph G_1 with size ($|V| = 400K, |E| = 800K$) from *Pockec* and create three more instances of this network, namely G_2, G_3, G_4, by gradually growing it to networks with sizes ($|V| = 800K, |E| = 2.4M$), ($|V| = 1.2M, |E| = 6.9M$), and ($|V| = 1.6M, |E| = 22M$), respectively. Similarly, we generate four instances of *Friendster* G_1, G_2, G_3, and G_4 with ($|V| = 4M, |E| = 10M$), ($|V| = 8M, |E| = 36M$), ($|V| = 12M, |E| = 80M$), and ($|V| = 16M, |E| = 110M$), respectively. We set $k = 1$, $\ell = 3$, $|V_q| = 3$ and test the algorithms in a streaming mode. The Y-axis in Fig. 14 plots the *running time ratio* of G_i and G_1 where $0 < i \leq 4$. The actual running times of all algorithms for G_1 are shown in their corresponding legends. If an algorithm did not finish generating results in one hour or run out of memory, we do not show it in the chart.

We can observe that simulation-based algorithm SIMPO- PANDA has the best scalability and outperforms all other algorithms. Its result quality is also superior to others (except SEN- PANDA). On the other hand, subgraph isomorphism-based SEN- PANDA cannot finish execution on G_3 and G_4. Similarly, PO- PANDA can handle up to G_3 for the *Pokec* network and G_2 for the *Friendster* network. The reasons are as discussed earlier. Although TORQUE+ is more efficient than SEN- PANDA, it has similar weakness and cannot handle larger networks. Furthermore, PO- PANDA is more scalable than TORQUE+. Note that NEMA+ failed to scale beyond G_2 of *Friendster* because it ran out of memory during index construction.

Figure 15 shows the running times of SIMPO- PANDA on G_4 for different values of k. Observe that it outperforms NEMA+ for all values of k. It is worth noting that although SIMPO- PANDA is a sequential algorithm running on a single machine, it can successfully execute PTQs on real-world large networks with millions of nodes and billion of edges.

7.2.5 User study

Lastly, we undertake a user study to evaluate the result quality of different algorithms. We select 10 test queries from all benchmark queries for each dataset. Ten unpaid male volunteers (ages from 21 to 27) participated in evaluating the resulting matches of all algorithms. Given a PTQ, they rate each algorithm from 1 to 5 where 5 represents the highest quality. We compute the *average rating* of each algorithm for all test queries across all datasets. The rank of these algorithms in descending order of average rating is as follow: SEN- PANDA > PO- PANDA > SIMPO- PANDA > TORQUE+ > SP+ > NEMA+. Observe that the results are consistent with our discussion in Sect. 7.2.1.

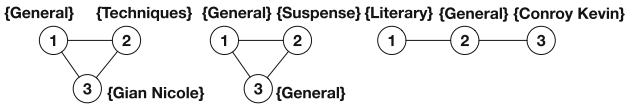
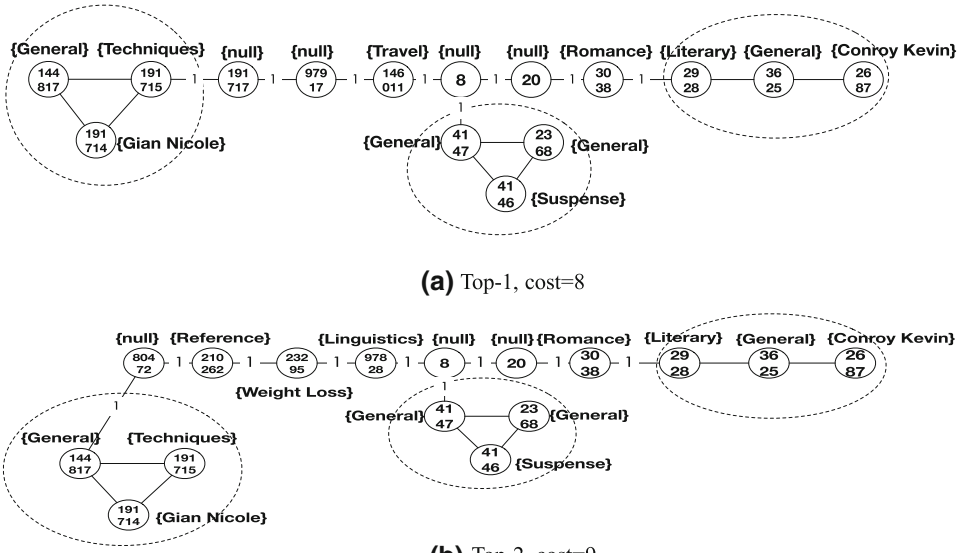


Fig. 16 A PTQ

To further validate the superiority of our proposed techniques, we briefly present a case study on the *Amazon* dataset to compare result qualities of PO- PANDA, NEMA+, and SP+. Consider the PTQ in Fig. 16 with three query components. The top-2 matches returned by PO- PANDA, NEMA+, and SP+ are shown in Figs. 17, 18, and 19, respectively. The matching subgraphs for the three query components are highlighted using dotted lines. Observe that NEMA+ has the worst quality consistent with our previous discussions. On the other hand, PO- PANDA has the best quality. Although all matching subgraphs are found by SP+, the quality is worse than that of PO- PANDA as the former is a heuristic algorithm based on shortest paths between merged nodes.

8 Related work

There has been considerable work on subgraph query processing over large networks. *Subgraph containment* query processing techniques such as GADDI [32], CP-index [27], and TurboISO [11] focus on efficiently finding all or a subset of exact matches of a given query graph in the network. Recently, there has also been effort to scale exact subgraph matching to billion-node graphs on a distributed framework [25]. *Subgraph similarity* queries, on the other hand, seek for *approximate* or *inexact* matches to the query graph by allowing missing edges or nodes. The core component of the evaluation mechanism of several techniques for such queries, such as TALE [26], SAPPER [30], TreeSpan [33], is the notion of *graph similarity*, which is computed using a variety of measures such as *graph edit distance* [30,31] and *maximum connected common subgraph* [10,24]. In the former approach, the similarity of two graphs is defined by the least edit operations (insertion, deletion, and relabeling) used to transform one graph into another. The latter approach detects the largest connected subgraph of the query graph that is subgraph-isomorphic to a subgraph in the network. Also, subgraph similarity search has recently been realized on top of a distributed platform [29].



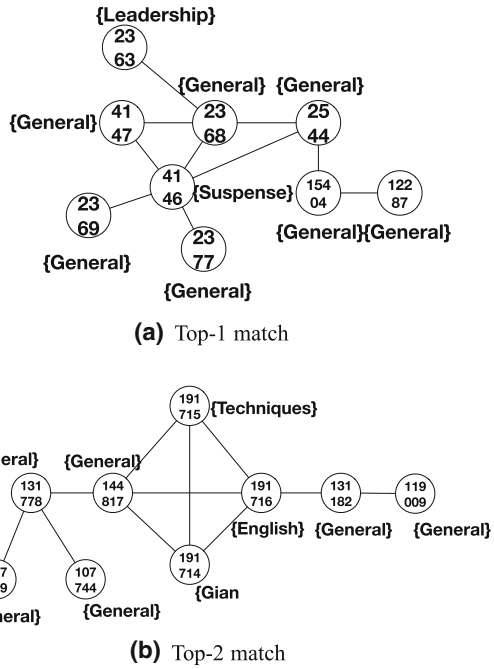


Fig. 18 Top-2 matches returned by NEMA+

Other efforts on inexact subgraph matching include *homomorphism-based* subgraph matching [9], which aims to find all the subgraphs where edges of the query graph are mapped to paths of a given maximum length and the label differences are within a specified threshold. More recently, Khan et al. [17] proposed a more flexible homomorphism-based subgraph matching framework called NEMA that allows for ambiguity in both topology and node labels. The matches to a query are driven by a *cost metric*, which considers both the cost of matching node labels and their neighborhoods within a certain hops. Specifically, it transforms the homomorphism problem to an equivalent inference problem in graphical models [21] and apply an inference algorithm to identify the optimal matches. Note that NEMA only extracts the most probable result, i.e., top-1 match.

In contrast to the proposed partial topology query, all the aforementioned efforts assume that the query graph is connected. Hence, these efforts are complimentary to our approach as they are not designed to address the partial topology-based network search problem. In fact, as highlighted in Sect. 1, the exact and approximate subgraph matching algorithms cannot be easily adapted to address such queries. First, it becomes prohibitively expensive to evaluate a large set of connected query graphs, which is generated by considering all possible connections between the query components of a PTQ, using existing exact subgraph matching algorithms. Second, as shown in Sect. 7, existing approximate subgraph matching methods cannot be trivially extended to process PTQ as they often lead to poor quality results due to violation of various topological constraints in a PTQ. Specifically, existing approximate subgraph matching methods usually design a cost metric to incorporate nodes' and edges' mismatch or matching cost. In contrast, PANDA's cost metric is designed to measure the paths between isomorphic matches of different query components in a result match. Lastly, the single-label merged graph generation, the GST-based matching process, and the label propagation process in a merged graph with multi-label merged nodes are novel as they are not deployed in any existing exact or approximate subgraph matching techniques for large networks.

Graph simulation has also been adopted for pattern matching. Several recent work extend the definition of graph simulation to *strong simulation* [19] and *bounded simulation* [8] to generate superior result quality while maintaining the same algorithmic complexity. Bounded simulation extends

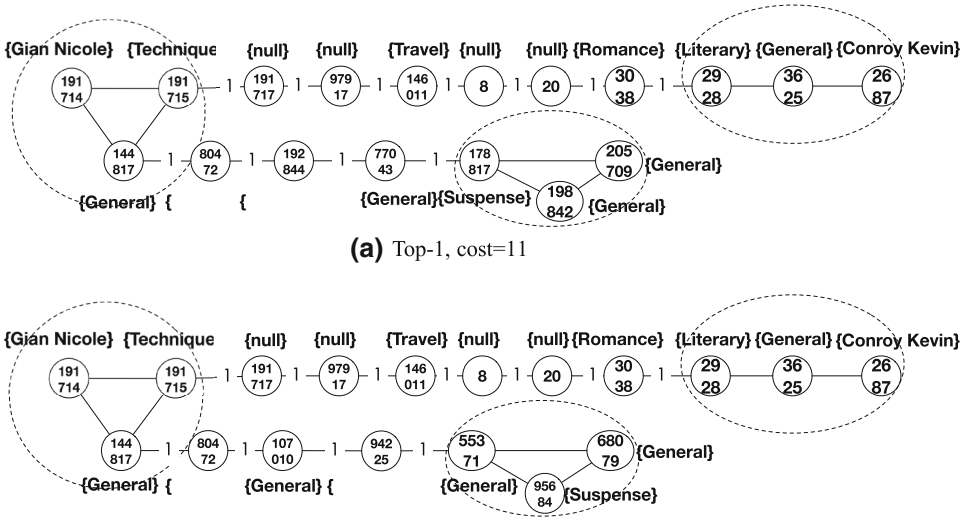


Fig. 19 Top-2 matches returned by SP+

simulation by allowing bounds on the number of hops in pattern graphs and then incorporating regular expressions as edge constraints on pattern graphs using [7]. Strong simulation enforces the *locality* (i.e., restrict match within a "ball") and *duality* (consider both child and parent relationships) properties in matching graphs.

Yang et al. [28] recently proposed a generalized graph search problem that uses a metric combining an extensible set of *transformation functions* to automatically map keywords and connectivities from a query graph to their matches in a network. A key feature of this work is that it automatically learns a *ranking model* and utilizes it to return top-k matches by leveraging graph sketches and belief propagation. Specifically, it supports *partial connected* queries comprising a connected graph query and/or a set of single node components representing keyword queries. In contrast, our partial topology query is more generic as it allows a set of query components where each connected component can be of any size and not simply a node representing a keyword query. Furthermore, this approach processes these queries by inserting a set of implicit edges where each edge connects a pair of nodes from different query components. However, such strategy is expensive as shown in Sect. 7.

In [1,2], *topology-free matching* is performed on a PPI network by considering the network as a vertex-colored graph and utilizing linear programming to solve it. The work in [22,23] extends the topology-free matching problem to partial information network queries for biological networks. These matching problems are different from ours, and their proposed algorithms mainly focus on discovering the direct relation instead of a latent path to connect different query components. Consequently, these algorithms become infeasible when query component matches are relatively far from each other. Furthermore, as shown in Sect. 7, they do not scale to large networks as they are designed for relatively small biological networks. In fact, topology-free matching query is a special case of our problem when each query component in a PTQ contains only one node.

9 Conclusions and future work

In this paper, we present a novel graph querying framework called PANDA for processing partial topology queries (PTQs) in a single machine. In contrast to traditional query graphs, a PTQ is disconnected and consists of two or more connected query components. We prove that PTQ evaluation is an NP-hard problem and propose two algorithms, namely SEN- PANDA and PO- PANDA, to find top-k matches of a PTQ. SEN- PANDA generates results of highest quality but is not suitable for very large networks. On the other hand, PO- PANDA is more efficient and scalable while slightly sacrificing result quality. We further enhance PO- PANDA with a simulation-based optimization technique to improve its scalability and efficiency. Our experimental results demonstrate the superiority of these algorithms over several *extended* state-of-the-art graph querying techniques.

Observe that PANDA embodies a sequential algorithm framework designed to handle large networks (i.e., containing tens of millions of nodes) in a single machine. Hence, it is a non-trivial challenge to extend it on a distributed platform to support networks with billions of nodes. As part of future work, we intend to investigate this challenging problem.

Acknowledgements Qing Wang is supported by the National Natural Science Foundation of China under grants 61432001, 91318301, 91218302.

References

1. Bruckner, S., Huffner, F., Karp, R.M., Shamir, R., Sharan, R.: Torque: topology-free querying of protein interaction networks. Nucl. Acids Res. **37**(2), 106–108 (2009)
2. Bruckner, S., Huffner, F., Karp, R.M., Shamir, R., Sharan, R.: Topology-free querying of protein interaction networks. J. Comput. Biol. **17**(3), 237–252 (2010)
3. Buchan, N., Croson, R.: The boundaries of trust: own and others actions in the US and china. J. Econ. Behav. Organ. **55**(4), 485–504 (2004)
4. Cordella, L., Foggia, P., Sansone, C., Vento, M.: A (sub)graph isomorphism algorithm for matching large graphs. Pattern Anal. Mach. Intell. IEEE Trans. **26**(10), 1367–1372 (2004)
5. Ding, B., Xu Yu, J., Wang, S., Qin, L., Zhang, X., Lin, X.: Finding top-k min-cost connected trees in databases. In: ICDE, pp. 836–845 (2007)
6. Duin, C., Volgenant, A., Voß, S.: Solving group steiner problems as steiner problems. Eur. J. Oper. Res. **154**(1), 323–329 (2004)
7. Fan, W., Li, J., Ma, S., Tang, N., Wu, Y., Wu, Y.: Graph pattern matching: from intractable to polynomial time. VLDB **3**(1–2), 264–275 (2010)
8. Fan, W., Li, J., Ma, S., Tang, N., Wu, Y.: Adding regular expressions to graph reachability and pattern queries. In: ICDE (2011)
9. Fan, W., Li, J., Ma, S., Wang, H., Wu, Y.: Graph homomorphism revisited for graph matching. In: PVLDB (2010)
10. Fernández, M.-L., Valiente, G.: A graph distance metric combining maximum common subgraph and minimum common supergraph. Pattern Recognit. Lett. **22**(6–7), 753–758 (2001)
11. Han, W.-S., Lee, J., Lee, J.-H.: TurboISO: towards ultrafast and robust subgraph isomorphism search in large graph databases. In: SIGMOD (2013)
12. He, H., Wang, H., Yang, J., Yu, P.S.: Blinks: ranked keyword searches on graphs. In: SIGMOD, pp. 305–316 (2007)
13. Helvig, C.S., Robins, G., Zelikovsky, A.: An improved approximation scheme for the group steiner problem. Networks **37**(1), 8–20 (2001)
14. Henzinger, M.R., Henzinger, T., Kopke, P.: Computing simulations on finite and infinite graphs. In: FOCS (1995)
15. Ihler, E.: Bounds on the quality of approximate solutions to the group steiner problem. In: Graph-Theoretic Concepts in Computer Science, pp. 109–118 (1991)
16. Karp, R.M.: Reducibility Among Combinatorial Problems. Springer, Berlin (1972)
17. Khan, A., Wu, Y., Aggarwal, C.C., Yan, X.: NeMa: fast graph search with label similarity. VLDB **6**(3), 181–192 (2013)
18. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection (2014)
19. Ma, S., Cao, Y., Fan, W., Huai, J., Wo, T.: Strong simulation: Capturing topology in graph pattern matching, vol. 39. In: TODS (2014)
20. Morsey, M., Lehmann, J., Auer, S., Ngomo, A.-C.N.: DBpedia SPARQL benchmark-performance assessment with real queries on real data. In: ISWC, volume 7031 of LNCS, pp. 454–469. Springer, Berlin (2011)
21. Pearl, J.: Reverend Bayes on inference engines: a distributed hierarchical approach. In: AAAI (1982)
22. Pinter, R.Y., Shachnai, H., Zehavi, M.: Partial information network queries. J. Discrete Algorithms **31**, 129–145 (2015)
23. Pinter, R.Y., Shachnai, H., Zehavi, M.: Improved parameterized algorithms for network query problems. In: Parameterized and Exact Computation, pp. 294–306. Springer (2014)
24. Shang, H., Lin, X., Zhang, Y., Yu, J. X., Wang, W.: Connected substructure similarity search. In: SIGMOD, pp. 903–914 (2010)
25. Sun, Z., Wang, H., Wang, H., Shao, B., Li, J.: Efficient subgraph matching on billion node graphs. In: PVLDB (2012)
26. Tian, Y., Patel, J.M.: TALE: a tool for approximate large graph matching. In: ICDE, pp. 963–972 (2008)
27. Xie, Y., Yu, P.S.: CP-index: on the efficient indexing of large graphs. In: CIKM (2011)
28. Yang, S., Wu, Y., Sun, H., Yan, X.: Schemaless and structureless graph querying. VLDB **7**(7), 565–576 (2014)
29. Yuan, Y., Wang, G., Xu, J. Y., Chen, L.: Efficient distributed subgraph similarity matching. VLDB J. **24**(3), 369–394 (2010)
30. Zhang, S., Yang, J., Jin, W.: SAPPER: subgraph indexing and approximate matching in large graphs. VLDB **3**, 1185–1194 (2010)
31. Zeng, Z., Tung, A. K. H., Wang, J., Feng, J., Zhou, L.: Comparing stars: on approximating graph edit distance. In: VLDB (2009)
32. Zhang, S., Li, S., Yang, J.: GADDI: distance index based subgraph matching in biological networks. In: EDBT (2009)
33. Zhu, G., Lin, X., Zhu, K., Zhang, W., Yu, J.X.: TreeSpan: efficiently computing similarity all-matching. In: SIGMOD, pp. 529–540 (2012)