PANDA: A System for Partial Topology-based Search on Large Networks

Miao Xie[1] Sourav S Bhowmick[1] Hao Su[1] Gao Cong[1] Wook-Shin Han[2]
[1]Nanyang Technological University, Singapore
[2]POSTECH, South Korea
xiemiao|assourav|hsu007|gaocong@ntu.edu.sg, wshan@postech.ac.kr

ABSTRACT

A large body of research on subgraph query processing on large networks assumes that a query is posed in the form of a connected graph. Unfortunately, end users in practice may not always have precise knowledge about the topological relationships between nodes in a query graph to formulate a connected query. In this demonstration, we present a novel graph querying paradigm called *partial topology-based network search* and a query processing system called PANDA to efficiently find *top-k matches* of a *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. We demonstrate various innovative features of PANDA and its promising performance.

PVLDB Reference Format:
Miao Xie, Sourav S Bhowmick, Hao Su, Gao Cong, and Wook-Shin Han. PANDA: A System for Partial Topology-based Search on Large Networks. *PVLDB*, 11 (12): 1966 - 1969, 2018.
DOI: https://doi.org/10.14778/3229863.3236236

1. INTRODUCTION

Subgraph search, which retrieves one or more subgraphs in a network that *exactly* or *approximately* match a user-specified query graph, has many real-world applications. A common assumption among existing subgraph search techniques is that the query graph is connected. That is, users precisely know the topological structure of what they are seeking. Unfortunately, due to the topological complexity of the underlying network data, it is often unrealistic to assume that an end user is aware of the precise relationships between nodes in a query graph, necessary to formulate a "valid" query. There are many instances in which a user has a clear goal in mind but only a vague idea of how a query should be specified. Consequently, it is not always possible to express a query using a connected graph.

As an example, consider the collaboration network (*e.g., LinkedIn*) in Figure 1(a) 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 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. Initially, Bob wishes to form a subteam consisting of three members (1 PM, 2 PRGs) and hopes that the PM and two PRGs have collaborated with each other. Fortunately, he can formulate this requirement as a connected subgraph query as shown in Figure 1(b) and leverage on any state-of-the-art graph query processing system (*e.g.,* [3]) to retrieve matching subgraphs to his query. After few weeks, Bob wishes to expand his team to five members by adding a subteam comprising of a TS and a DB who have collaborated together. 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. Note that 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 the project. Unfortunately, Bob now cannot express these requirements in form of a connected subgraph query as he is unaware of the precise topological connection between these two subteams. Instead, Figure 1(c) is a query graph that embodies Bob's requirements and preferences. Observe that it is disconnected and consists of two non-overlapping connected *query components*, each of which shows the topology that needs to be matched exactly in any query result. Furthermore, the distances between these matching components need to be as close as possible. We refer to such a disconnected query graph as a *partial topology query* [5].

In this demonstration, we present a novel subgraph query processing system called PANDA (**PA**rtial Topology-based **N**etwork **D**ata Se**A**rch) [5] to efficiently process a *partial topology query* (PTQ) in a single machine. A PTQ $Q_P = (q_1, \ldots, q_\ell)$ comprises two or more disjoint *query components* (*i.e.,* $\ell \geq 2$). A *query component* $q_i = (V_{q_i}, E_{q_i})$ is an unweighted and undirected connected graph. We refer to i in

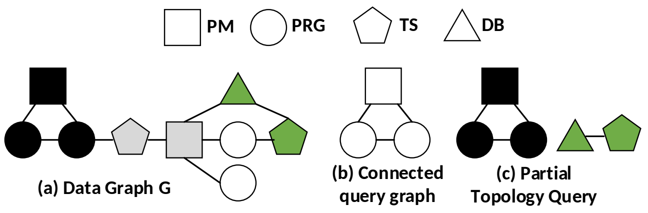

Figure 1: Motivating example.

Proceedings of the VLDB Endowment, Vol. 11, No. 12
Copyright 2018 VLDB Endowment 2150-8097/18/8.
DOI: https://doi.org/10.14778/3229863.3236236

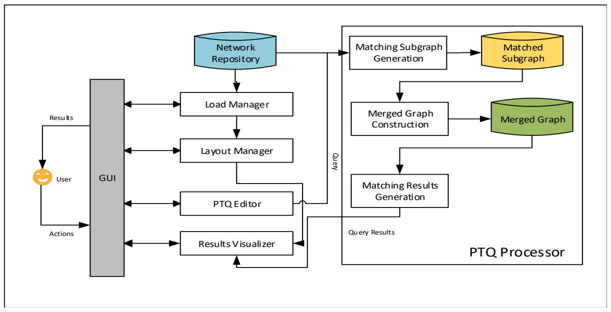

Figure 2: Architecture of PANDA.

q_i as *component index*. Figure 1(c) shows a PTQ where $\ell = 2$. Then, given an undirected network G, PANDA returns top-k *matches* 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. The subgraph depicted using shaded nodes in Figure 1(a) is an example match of the PTQ in Figure 1(c). Observe that it ensures that a member does not play multiple roles.

In this demo, we will first present a walk-through of the PANDA tool, and explain how it facilitates visual formulation of graph queries (*i.e.,* PTQs) that are extremely difficult to formulate using a connected graph. We will then show how it can be used to process PTQs. Specifically, we demonstrate three flavors of PTQ processing algorithms (SEN-PANDA, PO-PANDA and SIMPO-PANDA) [5], designed to handle networks of varying size. We showcase the benefits and tradeoffs of these algorithms w.r.t runtime and result quality. Finally, we will highlight how PANDA can enable *map-based* interactive visualization of PTQ results on large networks.

2. SYSTEM OVERVIEW

Figure 2 shows the system architecture of PANDA and mainly consists of the following modules. The reader may refer to [5] for algorithmic details and performance results.

The GUI module. Figure 3 is a screenshot of the visual interface of PANDA. It consists of four panels. Panel 1 enables us to load a new network dataset to query, trigger visual formulation and processing of a new PTQ, and invoke a configuration panel to set various parameters (*e.g.,* k for top-k matching results) and select a PTQ processing algorithm (*e.g.,* SIMPO-PANDA). Panel 2 is comprised of two subpanels. The top subpanel lists the network data sources that are currently stored in the underlying network data repository. Upon selecting a specific data source (*e.g.,* the citation network *Cit-HepPh*), the bottom part of Panel 2 displays the list of unique labels of the nodes in this selected network. Panel 3 shows the visual layout of the selected network in the form of a *network map* (generated by the *Layout Manager* module). Panel 4 is the PTQ *editor*, which is used to visually formulate a PTQ. Each query component in a PTQ is assigned a unique node color to distinguish one component from another. Once a query is executed, the location of its matching results are shown on the network map using red pins (generated by the *Results Visualizer* module). A user can click on these pins to see further details of the matches.

The Load Manager module. This module enables us to store all relevant information related to a network dataset

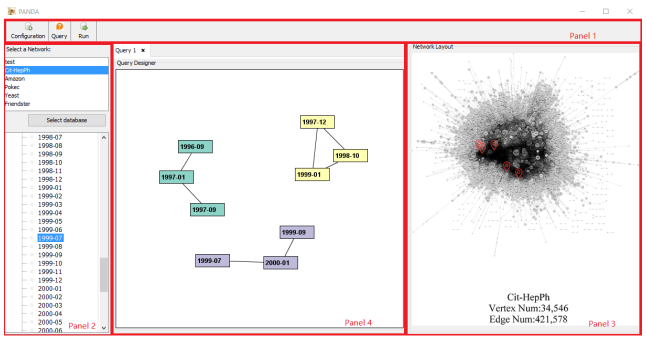

Figure 3: The PANDA GUI.

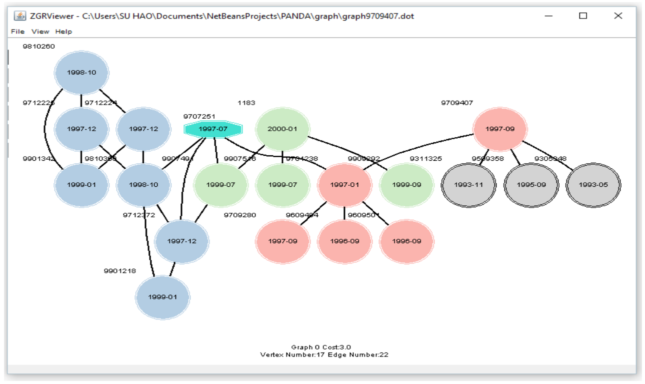

Figure 4: Local level visualization of a match.

such as the edge relations, node labels, and precalculated unique labels within each network. When a user selects a network to query, it loads the network data and associated information in Panels 2 and 3.

The Layout Manager module. The key goal of this module is to generate the "map-view" of a network dataset by utilizing a graph layout algorithm and to store the coordinates of each node in Panel 3 off-line for fast generation of the network map during query formulation. In this demonstration, we use the *sfdp* of *GraphViz* for generating the layout. This module also leverages a linear-time *k-core* algorithm [2] to calculate the *coreness* of each node in order to adjust the size of the nodes accordingly to facilitate visualization of the network structure. Given the result matches of a PTQ, the layout manager is also responsible for pinning the location of each match on the network map (Panel 3) to display their relative positions. When a user rolls over the mouse on a pin, it shows the *cost* and *rank* of the corresponding match. If she clicks on a pin, it will display the details of the matching result in a separate window (Figure 4).

The PTQ Editor module. This module facilitates visual formulation of a PTQ in Panel 4. Specifically, a user may drag a node label from Panel 2 and drop it in Panel 4 to create a node of a query component. An edge between a pair of query nodes is constructed by left and right clicking on them. Once she clicks the `"Run"` button (Panel 1), the number of query components in the PTQ is checked, and unique colors are assigned to them. For example, in Figure 3, the PTQ contains three query components. Hence, three unique colors are assigned to them.

The PTQ Processor module. Given a PTQ formulated using the GUI, this module aims to process it efficiently.

That is, given a network $G = (V, E)$, a PTQ $Q_P = (q_1, \ldots, q_\ell)$ where q_i is a query component, and a positive integer k, the goal of this module 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 $cost(M_1) \leq cost(M_2) \leq \ldots \leq cost(M_k)$. Intuitively, the *cost* of a match (*i.e., cost*(M)) is the sum of weights of all edges that do not "belong" to matching subgraphs of the query components. Consequently, it measures the total cost to connect the matching subgraphs. This problem is NP-hard [5]. To tackle this challenge, the PTQ processor implements three key submodules, namely, *matching subgraph generation*, *merged graph construction*, and *matching results generation*, which we shall elaborate below. Based on this framework, this module implements three variants of the PTQ processing algorithm, namely, SEN-PANDA, PO-PANDA, and SIMPO-PANDA [5].

SEN-PANDA is an exact algorithm based on Group Steiner Tree (GST) [1] and is particularly suitable for applications where the underlying network data is not very large and result quality is paramount. On the other hand, PO-PANDA exploits a novel *label propagation*-based technique to generate approximate results with a tight performance guarantee. Both of these algorithms find matching subgraphs of the query components using subgraph isomorphism, which is an NP-complete problem. To tackle this challenge, SIMPO-PANDA extends PO-PANDA by leveraging subgraph simulation [4], which runs in cubic time, to enhance the efficiency and scalability of the PTQ processor without compromising on the result quality.

The Matching Subgraph Generation submodule. Given G and a PTQ $Q_P = (q_1, \ldots, q_\ell)$, this submodule finds 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 *matches* g and g is a subgraph of G. In total, we can get ℓ sets of matching subgraphs for all query components, $\{\text{SM}_1, \ldots, \text{SM}_\ell\}$ where $\text{SM} = \text{SM}_1 \cup \cdots \cup \text{SM}_\ell$. For SEN-PANDA and PO-PANDA, the current version of PANDA implements TurboIso [3], a state-of-the-art technique to find exact subgraph matches via subgraph isomorphism. Note that nodes of a matching subgraph g may match nodes in more than one query components of Q_P. For example, consider the PTQ Q_P and the network G in Figures 5(a) and 5(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. 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 $V_{g_6} = \{v_4\}$.

Observe that both PO-PANDA and SEN-PANDA utilize expensive subgraph isomorphism to implement this submodule. SIMPO-PANDA, on the other hand, replaces the subgraph isomorphism-based strategy with subgraph simulation-based one. Note that all matching results of subgraph isomorphism are contained in the results of subgraph simulation. Specifically, it finds all *connected simulation subgraphs* using the simulation matching pairs.

The Merged Graph Construction submodule. Due to overlapping nature of matching subgraphs, we need a systematic way to represent SM in order to facilitate efficient search for result matches of Q_P. This submodule addresses this issue by *merging* the matching subgraphs in SM into a set of special nodes called *merged nodes.* Using these merged nodes, it constructs a new connected graph called *merged graph* from G by "hiding" SM via merged nodes.

Intuitively, a *merged node* $s \notin V$ represents an *aggregation* of one or more matching subgraphs, and is annotated with a set of *labels* Υ_s that has different meaning with the label set of nodes. 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 $I_s = (V_s, E_s)$. Note that if a matching subgraph g is non-overlapping or disjoint to other subgraphs in SM, it is represented by a merged node s whose inner graph I_s is identical to g. For example, reconsider Figure 5. 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 Figure 5(b) show these inner graphs. Furthermore, $\Upsilon_{s_1} = \{1, 2\}$, $\Upsilon_{s_2} = \{1, 2\}$, and $\Upsilon_{s_3} = \{3\}$. The *outer edges* of the merged nodes s_1, s_2, and s_3 are $\{(v_2, v_1), (v_6, v_1)\}$, $\{(v_7, v_3)\}$, and $\{(v_4, v_1)\}$, respectively.

Note that in SIMPO-PANDA a set of connected simulation subgraph can be represented by merged nodes similar to the aforementioned way isomorphic matching subgraphs are represented in PO-PANDA and SEN-PANDA. However, now the inner graph of a merged node may not necessarily contain isomorphic matches to a query component.

Given the set of merged nodes $\mathbb{S} = \{s_1, \ldots, s_n\}$, it transforms G to a *merged graph* $H = (V_H, E_H)$ by substituting the inner graphs with their corresponding merged nodes. In H, it maintains 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. For example, consider the inner graphs in Figure 5(b). Figure 5(c) shows the merged graph by substituting the inner graphs with the merged nodes s_1, s_2, and s_3 and maintaining the minimum-weight outer edges. Note that the outer edges will be utilized by the next submodule to search for the shortest path between a pair of merged nodes.

The Matching Results Generation submodule. It 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. It implements three different strategies that correspond to the three algorithms.

SEN-PANDA generates 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 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, it can find the top-k matches in each SMG by using an existing GST algorithm [1] and rank them globally.

Although the matching results generation strategy of SEN-PANDA can find the best solution, it becomes prohibitively expensive as the number of query components and network size increase due to the invocation of the GST algorithm exponential number of times for potentially exponential number of SMGs. The matching results generation strategy of PO-PANDA addresses this limitation by only searching the merged graph without generating any SMGs. It leverages 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. It checks 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, PO-PANDA searches the results of the query directly on the merged graph in Figure 5(c). 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. The top-k matching results are retrieved from the candidate trees. Specifically, this strategy guarantees that the propagation can be achieved in polynomial time by visiting each node at most twice.

In SIMPO-PANDA, as the inner graph of a merged node may not contain any matching subgraph via subgraph isomorphism, the label exploration process of PO-PANDA is extended 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, it performs 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.

The Results Visualizer module. This module is responsible for *two-level* visualization of matching results of a PTQ in real-time. At the *global level*, it sends locations of the matches in the network to the *Network Layout Manager* to display them on the network map using red pins (as discussed earlier). At the *local level*, it displays each match visually in response to the click on corresponding pin by a user. Specifically, nodes of a matching query component are highlighted with a circular shape, and colors of these nodes are consistent with the colors of the corresponding query components. In contrast, nodes connecting query components are depicted with green octagons. When a user clicks on a node in a matching result, the neighbors that are not in it are displayed as gray circular nodes to enable exploration of nearby nodes of the match. Figure 4 depicts an example of local level visualization of a match of the PTQ in Figure 3.

3. RELATED SYSTEMS AND NOVELTY

Existing exact and approximate subgraph search techniques assume that a query graph is connected. Consequently, they cannot be easily adapted to address PTQs. Yang *et al.* [6] recently proposed a technique to process partial connected queries comprising a connected graph query and/or a set of single node components representing keyword queries. In contrast, our PTQ 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 [5].

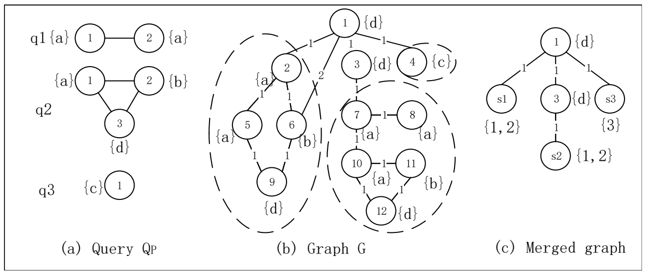

Figure 5: Merged graph.

4. DEMONSTRATION OBJECTIVES

PANDA is implemented in Java JDK 1.8. Our demonstration will be loaded with a few real datasets (*e.g.,* citation network, Amazon co-purchase network, Yeast PPI network) with different sizes (thousands to millions of nodes). Example PTQs on each of these networks will be presented. Users can also write their own ad-hoc queries.

The audience will be requested to draw a query graph for a query given to them in natural language (English). Through this interaction, they will appreciate the difficulty in formulating a connected query graph and the need for PTQ processing framework. Specifically, the GUI of PANDA shall assist users in gaining such experience. Through the PTQ *Editor* and *Results Visualizer* modules, one will be able to interactively formulate a PTQ due to the difficulty in formulating connected query graphs and view the result matches at global and local levels as well as interactively explore neighborhoods of specific parts of result matches in real-time (Figure 4). The audience shall also be able to gain differential experience on runtime and result quality of exact and approximate PTQ algorithms (SEN-PANDA, PO-PANDA, and SIMPO-PANDA) by formulating a variety of PTQs (having different number of query components) on networks with different sizes. Specifically, they will be able to experience that SIMPO-PANDA can efficiently generate good quality results on large networks. Finally, we shall also demonstrate how PANDA can be utilized to address the *keyword search* problem as a keyword query is a special case of PTQ (*i.e.,* each query component contains a single vertex).

A short video to illustrate the aforementioned features of PANDA is available at `https://youtu.be/xplZ6SjDRmU`.

Acknowledgments. Sourav S. Bhowmick and Hao Su are supported by the AcRF Tier-2 Grant MOE2015-T2-1-040. Wook-Shin Han was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF), funded by the Ministry of Science, ICT (No.NRF-2017M3C4A7065887).

5. REFERENCES

[1] C. Duin, et al. Solving Group Steiner Problems as Steiner Problems. *Euro. J. of Operational Res.*, 154(1), 2004.
[2] J. A.-Hamelin, L. Dall'Asta, A. Barrat, A. Vespignani. Large Scale Networks Fingerprinting and Visualization Using the k-core Decomposition. In *NIPS*, 2005.
[3] W.-S. Han, J. Lee, J.-H. Lee. TurboISO: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases. In *SIGMOD*, 2013.
[4] M. R. Henzinger, T. Henzinger, P. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995
[5] M. Xie, S. S. Bhowmick, G. Cong, Q. Wang. PANDA: Towards Partial Topology-based Search on Large Networks in a Single Machine. In *The VLDB Journal*, 26(2), 2017.
[6] S. Yang, Y. Wu, H. Sun, X. Yan. Schemaless and Sructureless Graph Querying. *PVLDB* 7(7): 565-576, 2014. .