QUBLE: Blending Visual Subgraph Query Formulation with Query Processing on Large Networks

Ho Hoang Hung[§] Sourav S Bhowmick[§,¶] Ba Quan Truong[§,¶]
Byron Choi[†] Shuigeng Zhou[‡]

[§]School of Computer Engineering, Nanyang Technological University, Singapore
[¶]Singapore-MIT Alliance, Nanyang Technological University, Singapore
[†]Department of Computer Science, Hong Kong Baptist University, Hong Kong
[‡]School of Computer Science, Fudan University, China
hoho0002|assourav|bqtruong@ntu.edu.sg, choi@hkbu.edu.hk, sgzhou@fudan.edu.cn

ABSTRACT

In a previous paper, we laid out the vision of a novel graph query processing paradigm where instead of processing a visual query graph *after* its construction, it *interleaves* visual query formulation and processing by exploiting the latency offered by the GUI [4]. Our recent attempts at implementing this vision [4,6], show significant improvement in the *system response time* (SRT) for subgraph queries. However, these efforts are designed specifically for graph databases containing a large collection of small or medium-sized graphs. Consequently, its *frequent fragment-based action-aware indexing* schemes and query processing strategy are unsuitable for supporting subgraph queries on large networks containing thousands of nodes and edges. In this demonstration, we present a novel system called QUBLE (**QU**ery **B**lender for **L**arge n**E**tworks) to realize this novel paradigm on large networks. We demonstrate various innovative features of QUBLE and its promising performance.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems - Query processing

General Terms

Algorithms, Experimentation, Performance

1. INTRODUCTION

Querying graph databases has emerged as an important research problem for real-world applications that center on large graph 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 set of data graphs that exactly or approximately match a user-specified query graph. Efforts to address this problem can be broadly classified into two streams. One stream focuses on processing subgraph queries on a large number of small or medium-sized graphs (*e.g.,* chemical compounds). The other stream aims to handle query processing on a small number of large graphs (*e.g.,* protein interaction networks, social networks).

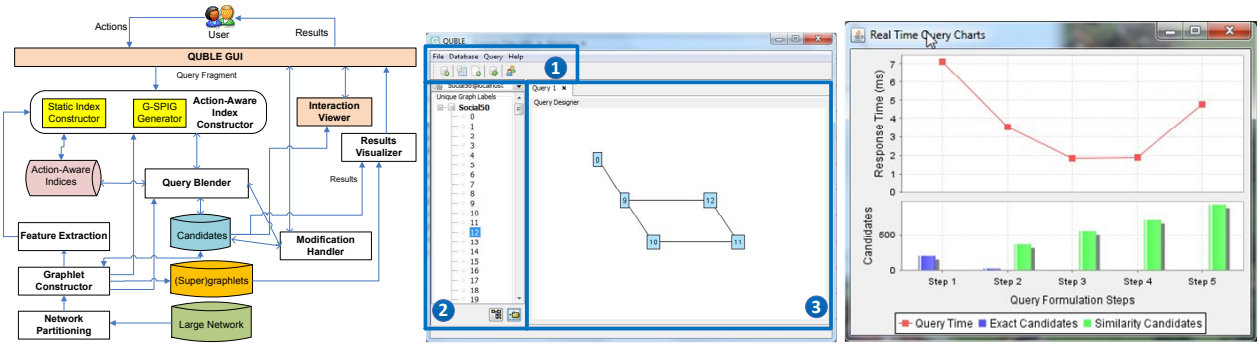
SIGMOD'13, 22 – 27 June, New York, USA

A number of graph query languages (*e.g.,* SPARQL) have been proposed that can be used to formulate subgraph queries. Unfortunately, in many real life domains it is unrealistic to assume that users are proficient in expressing graph queries using these languages. The traditional approach to address this query formulation challenge is to build a user-friendly visual framework on top of a state-of-the-art graph query processing technique. In this traditional visual query processing paradigm, although the final query that a user intends to pose is revealed gradually in a step-by-step manner during query construction, it is not exploited by the query processor prior to clicking of the `Run` icon to execute the query. This often results in slower *system response time* (SRT), which is the duration between the time a user presses the `Run` icon to the time when the user gets the query results, as the query processor remains idle during the entire query formulation process [4,6].

In [1, 4], we laid out the vision of a novel visual graph query processing paradigm where instead of processing a query graph after its construction, it *interleaves* the two traditionally orthogonal steps, namely visual query construction and processing, bringing in at least two key benefits. First, it ensures that the query processor does not remain idle during query formulation. Second, it significantly improves the SRT as in this new paradigm it is the time taken to process a part of the query that is yet to be evaluated (if any). Note that from a end user's perspective, the SRT is crucial as it is the time a user has to wait before she can view the results.

Our most recent work that implemented the above vision presented a visual subgraph querying algorithm called PRAGUE [6] designed specifically for graph databases containing a large collection of small or medium-sized graphs. Consequently, its frequent fragment-based indexing schemes and query processing strategy cannot be easily adopted to support subgraph queries on large networks. This is primarily because generating frequent subgraphs is itself a bottleneck here as the time complexity of subgraph isomorphism, the core procedure of any frequent subgraph mining algorithms, grows exponentially with the graph size. Furthermore, small-sized frequent fragments typically may occur many times. As a result, a large number of candidates may be generated against small-sized query fragments. Additionally, visualizing query results is computationally and cognitively challenging issue in the case of large networks. Even if a network contains few thousands of nodes and edges, it imposes significant cognitive burden as the entire network looks like a giant hairball and subgraphs that match the query are lost in the visual maze.

In this demonstration, *we present a novel system called* QUBLE *(**QU**ery **B**lender for **L**arge n**E**tworks) to realize our visual subgraph querying paradigm on large networks*. In particular, the



(a) Architecture. (b) Visual interface. (c) Interaction Viewer.

Figure 1: System architecture and visual interface of QUBLE.

architecture in [5] is modified significantly in the following ways to address the aforementioned challenges. First, we decompose a large network into pieces of small data graphs (called *graphlets*) while ensuring that no structural information is lost during this process. Consequently, the decomposed graph set can be viewed as a collection of small or medium-sized data graphs. Second, we discover *approximate* sets of frequent and infrequent fragments from this collection and *identify* their occurrences in the data graphs. A benefit of this strategy is that it is very storage-efficient as each fragment is now associated with a list of data graph identifiers in lieu of complete location details in the original network. Third, we *redefine* and *build action-aware indexes* of PRAGUE [5, 6] to support subgraph search. Fourth, a *(super)graphlet-at-a-time* interactive results visualization strategy is incorporated to support effective visualization of query results.

2. SYSTEM OVERVIEW

Figure 1(a) shows the system architecture of QUBLE and mainly consists of the following modules.

The GUI module: Figure 1(b) depicts the screenshot of the "edge-at-a-time" visual interface of QUBLE. A user begins formulating a query by choosing a network as the query target and creating a new query canvas using Panel 1. The left panel (Panel 2) displays the unique labels of nodes that appear in the dataset in lexicographic order. In the query formulation process, a user chooses labels from Panel 2 for creating nodes in the query graph. Panel 3 depicts the area for formulating graph queries. A user drags a node that is part of the query from Panel 2 and drops it in Panel 3. Next, she adds another node in the same way. Then, she creates an edge between the added nodes by left and right clicking on them. Additional nodes and edges are added to the query graph by repeating these steps. Finally, the user can execute the query by clicking on the `Run` icon in Panel 1. Figure 2(a) depicts the screenshot for interactive display of the query results (discussed later).

The Network Partitioning module: This module decomposes a large network to pieces of small data graphs by exploiting METIS [7], a fast and widely used minimum cut-based graph partitioning algorithm. Note that the task of such graph partitioning algorithm is to assign a single partition number to each node of the input network based on the required number of nodes in one partition. Edges that connect nodes that have different partition numbers are "cut" away. The goal is to minimize edge-cut while trying to achieve the required number of nodes in a partition.

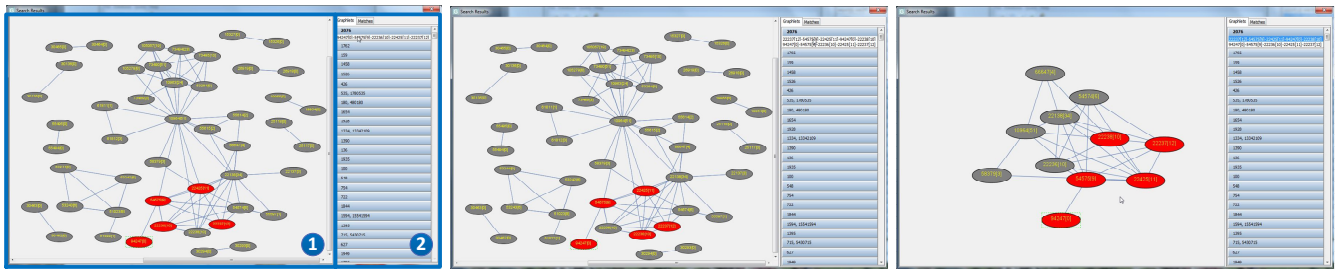
The Graphlet Constructor module: The goal of this module is to construct *graphlets* and *supergraphlets* from the partitions and cut edges. A *graphlet* is either a *partition graph* or a *bridge*. Informally, *partition graphs* are partitions generated by the graph partitioning algorithm on the original network. On the other hand, *bridges* are graphs that are constructed from cut edges that link certain pairs of partition graphs. Each graphlet (denoted by G_ℓ) is identified by a unique identifier, denoted by $gid(G_\ell)$. Clearly, any edge in the original network can only belong to exactly one graphlet. Two graphlets are *adjacent iff* they share some common nodes in the original network but not edges. A set of graphlets is considered as *adjacent set* (denoted as Δ) *iff* each graphlet is adjacent to at least one other graphlet in the set.

A *supergraphlet* is a graph generated by *merging* a set of adjacent graphlets. Formally, let $\Delta = \{G_{\ell_1}, G_{\ell_2}, \ldots, G_{\ell_n}\}$ be an adjacent set and $n \geq 2$. Then a *supergraphlet* $G_\Delta = (V_\Delta, E_\Delta)$ of Δ is a graph satisfying the followings: (a) $\forall\ v \in V_\Delta$, $v \in V_{\ell_i}$ where $0 < i \leq n$ and (b) $\forall\ e \in E_\Delta$, $e \in E_{\ell_j}$ where $0 < j \leq n$. Each supergraphlet G_Δ is assigned a *supergraphlet identifier*, denoted by $sgId(G_\Delta)$ ($sgId$ for brevity), which is a concatenation of identifiers of all graphlets in the adjacent set of the supergraphlet in ascending order. For example, let $gid(G_1) = 1$ and $gid(G_2) = 2$. Then if $\Delta = \{G_{\ell_1}, G_{\ell_2}\}$, $sgId(G_\Delta) = 1\text{-}2$. Since a gid can be considered as a special case of supergraphlet identifier containing only a single identifier, we shall use $sgId$ to denote a gid as well.

Obviously, constructing all possible supergraphlets is prohibitively expensive. However, some supergraphlets are needed to identify all occurrences of *frequent fragments* and SIFs (discussed below). Hence, this module constructs them *selectively* only when subgraph verification needs to be performed (*i.e.,* to verify if a supergraphlet actually contains a (sub)graph).

The Feature Extraction module: This module generates *frequent fragments* and SIFs from the graphlets and supergraphlets. Let g be a connected subgraph of G_ℓ or G_Δ in the set of graphlets and supergraphlets (denoted by $\mathcal{D}_\Delta$) and has at least one edge. Then, g is a *fragment* in $\mathcal{D}_\Delta$. Given a fragment $g \subseteq G_i$ and $G_i \in \mathcal{D}_\Delta$, G_i is called a *fragment support graph* (FSG) of g. We denote a set of supergraphlet identifiers of FSGs of g as $fsgId(g)$. The *support* of g (denoted as $sup(g)$) is the number of *graphlets* (not supergraphlets) that are FSGs of g. For example, let $fsgId(g) = \{3\text{-}7, 1, 6\}$. Then, $sup(g) = 2$ as we only count the graphlets.

A fragment g is *frequent* if $sup(g) \geq \alpha|\mathcal{D}_\ell|$ where α is the minimum support threshold, $0 < \alpha < 1$ and $\mathcal{D}_\ell \subseteq \mathcal{D}_\Delta$ is the set of graphlets. Otherwise, the fragment is *infrequent*. Since the number of infrequent fragments can be prohibitively large, we only index the *small infrequent fragments* (SIFs). Given an infrequent fragment g, g is a SIF if (a) $|g| = 1$ or (b) $|g| = 2$ and g is a *maximal cover graph* (MCG) of at least one adjacent set. A graph Q is called *maximal cover graph* (MCG) of a supergraphlet G_Δ where $\Delta = \{G_{\ell_1}, G_{\ell_2}, \ldots, G_{\ell_n}\}$ if Q is isomorphic to $G' = (V', E')$, $G' \subseteq G_\Delta$ and $\forall G_{\ell_i} \in \Delta$, $\exists e \in E'$ s.t $e \in E_{\ell_i}$. Q is said to have a *cover match* in the original network G. For distinction, we refer to an infrequent fragment that is not a SIF as *non-small infrequent*



(a) Result visualizer GUI. (b) Viewing approximate matches. (c) Drill-down "localized" view.

Figure 2: Results visualization in QUBLE.

fragment (NIF). Observe that that all size-one fragments that are not frequent are SIFs. Also, as $sup(g) \leq |fsgId(g)|$, g may be a frequent subgraph in the original network but not in $\mathcal{D}_\ell$. In this case if $|g| \leq 2$ then it is classified as a SIF. Otherwise, it is a NIF. Note that such "miscategorization" does not adversely impact the performance of QUBLE. Interestingly, it is not necessary to identify *all* frequent fragments accurately to support efficient visual subgraph query processing in our paradigm.

We now summarize the frequent fragment and SIF generation process. First, it uses an existing frequent graph mining algorithm (*gSpan* [8] in our case) to generate frequent fragments from the graphlet set. This step can identify *all* FSGs of size-one fragments as an edge can only belong to at most one graphlet. However, FSG sets of frequent fragments with size two or more are *incomplete* as a fragment can not only be subgraph of a graphlet but also a subgraph of a supergraphlet. Consequently, QUBLE takes a *two-phase* approach to resolve this issue. In the first phase, it identifies all the cover matches of all frequent fragments. It consists of two key steps. It identifies all the cover matches of frequent fragments whose size is two, *i.e.,* identify all supergraphlets that contain size-two frequent fragments. During this step, it also identifies size-two SIFs and some of their cover matches in the supergraphlets. Next, it identifies all the cover matches for frequent fragments having size greater than two. In the second phase, it completes identification of FSG sets of *all* SIFs.

The Action-Aware Index Constructor module: This module is responsible for constructing the following two types of indexes.

Action-aware static index. We use the two action-aware static indices of PRAGUE [4–6] for indexing frequent fragments and SIFs. The *action-aware frequent index* (A^{2}F) is a graph-structured index having a *memory-resident* and a *disk-resident* components called *memory-based frequent index* (MF-index) and *disk-based frequent index* (DF-index), respectively. The MF-index indexes all frequent fragments having size less than or equal to β (fragment size threshold) whereas frequent fragments with size larger than β reside in the DF-index. Specifically, the DF-index is an array of directed graphs called *fragment clusters*. Each vertex v in a fragment cluster is a frequent fragment g (represented by its CAM code [3]) and points to a set of FSG identifiers of g. There exists an edge (v', v) *iff* g' is a proper subgraph of g and $|g| = |g'| + 1$. The structure of MF-index is similar to that of a fragment cluster. In addition, *leaf* vertices representing frequent fragments of size β is associated with a *fragment cluster list* where each entry points to a fragment cluster in the DF-index that contains the fragment as a subgraph. The *action-aware infrequent index* (A^{2}I) consists of an array of SIFs arranged in ascending order of their sizes. Each entry stores the CAM code of a SIF g and a list of FSG identifiers of g.

Action-aware dynamic index. The G-SPIG *Generator* module generates a dynamic index called ***g**raphlet-based **spi**ndle-shaped **g**raph* (G-SPIG) on-the-fly during visual query construction by *extending* the idea of spindle-shaped graphs (SPIG) in PRAGUE [5,6]. For each *new* edge e_m created by the user, QUBLE creates a G-SPIG. Each edge is assigned a unique identifier according to their formulation sequence. That is, the *m-th* edge constructed by a user is denoted as e_m where m is the *label* of the edge.

Similar to a SPIG, a G-SPIG is also a directed graph where each vertex represents a subgraph g of the query fragment containing the new edge e_m. There is a directed edge from vertex v' to vertex v if $g' \subset g$ and $|g| = |g'| + 1$. The *source* vertex (vertex with no incoming edge) in the first level represents e_m and the *target* vertex (vertex with no outgoing edge) in the last level represents the entire query fragment at a specific step. *The content of v in a* G-SPIG *is different from* SPIG. Specifically, each v is associated with the CAM code of the corresponding g, a list of labels of edges of g, a list of identifier set called *Indexed Fragments List* (IFL) to capture information related to frequent or infrequent nature of g or its subgraphs, and a set of identifiers $\Omega(g)$ called *supergraphlet id set* to hold the $sgId$s of candidate graphlets and supergraphlets that may contain g, if g is not indexed by action-aware static indices (*i.e., g* is a NIF). An IFL contains two attributes, namely *frequent id* and SIF *id*. If g is in the A^{2}F-index or A^{2}I-index, then the identifier of the vertex or entry v representing g in the corresponding index is stored in *frequent id* or SIF *id* attribute, respectively. If g is neither in the A^{2}F-index nor in the A^{2}I-index, then $\Omega(g)$ stores the new $sgId$s created by "joining" (called *fragment join*) common graphlets in $fsgId$s of g_{v1} and g_{v2}, which are any two fragments associated with two different parents of v. For example, let $fsgId(g_{v1}) = \{4\text{-}7, 3\text{-}6, 2\}$ and $fsgId(g_{v2}) = \{3\text{-}7, 1, 6\}$. Then $\Omega(g) = \{3\text{-}4\text{-}7, 3\text{-}6\text{-}7, 3\text{-}6\}$. If g_{v1} or g_{v2} is in the A^{2}F-index or A^{2}I-index, then their corresponding FSG identifiers are retrieved from these indices to compute $\Omega(g)$. Otherwise, supergraphlet id sets of the two parents ($\Omega(g_{v1})$ and $\Omega(g_{v2})$) are used.

The Query Blender module: This module exploits the action-aware indices and the latency offered by the GUI actions to blend visual query formulation and query processing. When a user adds a new edge e_m to query q, it computes the identifiers of candidate graphlets and supergraphlets that contain q using the action-aware indexes. Specifically, if q represents a frequent fragment or SIF, it retrieves FSG id set of q from the A^{2}F-index or A^{2}I-index, respectively, and use it as the candidate set. Otherwise, q represents a NIF and the supergraphlet id set of q is used as the candidate set. Next, given a *subgraph distance threshold* σ, the G-SPIG set is exploited to identify relevant subgraphs of q that need to be matched for retrieving *approximate* candidates. We retrieve all the *connected common subgraphs* (CCS) such that the *subgraph distance*[1] is within σ . Specifically, these subgraphs are query fragments represented by the vertices at levels $|q| - 1$ to $|q| - \sigma$ in the G-SPIG set. In order to reduce the verification cost for a large

[1]The *subgraph distance* measures the number of edges that are allowed to be missed in q in order to match a (super)graphlet.

candidate set, the candidate set is separated into two parts, identifiers of verification-free candidate graphs (R_{free}) and identifiers of candidates that need verification (R_{ver}). For each vertex in the *i-th* level, if it is a frequent fragment or SIF, then the candidates are computed using the aforementioned exact matching procedure and combined with existing R_{free}. Otherwise, it is a NIF and requires verification. Consequently, supergraphlet id sets are exploited to compute the candidates and combined with existing R_{ver}. Lastly, candidates that exist in both R_{free} and R_{ver} are removed from R_{ver}.

The above steps are repeated for each new edge to incrementally update the candidate identifiers until the user clicks on the `Run` icon when the final set of graphlets and supergraphlets containing at least one matches (exact or approximate) are generated. Subgraphs that exactly match the query are verified, if necessary, from candidate graphlets and supergraphlets. Next, candidates that match the query approximately are added to the results (we extend VF2 [2] to handle CCS-based similarity verification).

The Interaction Viewer module: This module provides a real-time graphical view of the working of the visual query evaluation paradigm. It depicts the effect of *each* visual query formulation step, the size of candidate (super)graphlets as well as the time taken by QUBLE to compute them. Consider the construction of the query in Figure 1(b). The query evaluation process at every step is depicted in Figure 1(c). The bottom part of the screen displays sizes of candidate (super)graphlets at different steps. The top part of the display plots the time taken by the *Query Blender* module to compute and maintain candidate graphlets at every step.

The Modification Handler module: This module assists a user to modify the formulated query appropriately. A user is free to delete any edge that has been previously constructed by her. After a modification by the user, it updates the G-SPIG set by removing irrelevant vertices and updates the candidate set.

The Results Visualizer module: This module addresses the challenge mentioned in Section 1 to facilitate visualization of query results. Specifically, results are viewed in "supergraphlet-at-a-time" mode where one supergraphlet or graphlet containing result matches is displayed on the results screen one at a time. This enables viewing a small "piece" of the original network containing a match. Figure 2(a) depicts the results visualization of the query in Figure 1(b). Panel 2 shows the set of (super)graphlet identifiers containing at least one results match. Each (super)graphlet occupies a section of the panel and its matches are listed as list. By default, only matches with minimum subgraph distance to the query is displayed. For example, the graphlet with $sgId$ *2076* has a single exact match *94247[0]-54575[9]-22236[10]-22425[11]-22237[12]*[2]. Panel 1 displays the corresponding (super)graphlet (*e.g., 2076*). When the mouse is hovered on a match in Panel 2, the corresponding subgraph is highlighted in Panel 1 in red color. Moreover, when a vertex is double-clicked in Panel 1, neighbors of this vertex are automatically retrieved and displayed, further facilitating graph exploration. If a user wants to retrieve all matches (exact and approximate) in a particular (super)graphlet, she can right-click on the (super)graphlet identifier in Panel 2 to initiate computation of all matches. For example, the list associated with the graphlet *2076* in Panel 2 in Figure 2(b) shows all matches in the graphlet. Note that since the size of a (super)graphlet is significantly smaller than the original network, the remaining matches to the query graph can be quickly computed. Clicking on a match in Panel 2, presents a "localized" view of the matching subgraph in the network in Panel 1 (Figure 2(c)). Additionally, one can also view the results sorted in ascending order of subgraph distance by clicking on the tab `Matches` in Panel 2.

[2]Here each node is represented as $id[label]$ where id is a globally unique identifier and $label$ is the node's label which is also used in the query.

3. RELATED SYSTEMS AND NOVELTY

Recently, there have been a number of studies on subgraph query evaluation over large networks (*e.g.,* [9]). In contrast to QUBLE, none of these strategies address subgraph search problem by partitioning a large network or by exploiting frequent patterns for candidate pruning. More importantly, all these efforts follow the conventional query processing paradigm.

More germane to this work is our previous research in [4–6]. Firstly, we focus on querying large networks here instead of a large set of small or medium-sized graphs. Secondly, we use SIFs instead of DIFs (*discriminative* infrequent fragments) as representative infrequent fragments. Thirdly, the generation of frequent fragments and SIFs are much more involved in QUBLE. Note that in [4, 6], frequent fragments and DIFs can be directly generated by using *gSpan* [8]. Fourthly, although the topological structure of a SPIG [6] and G-SPIG is identical, the vertex content is different. In G-SPIG, each vertex stores a set of (super)graphlet identifiers of ($\Omega(g)$) and SIF id among other features, which are irrelevant in a SPIG. Due to these differences, the candidate generation process during visual query formulation also differs. Lastly, the visualization of result matches is much more challenging in QUBLE.

4. DEMONSTRATION OBJECTIVES

QUBLE is implemented in Java JDK 1.7. Our demonstration will be loaded with synthetic and real networks with different sizes (up to 100K nodes). Example query graphs will be presented. Users can also write their own ad-hoc queries through our GUI.

One of the key objectives of the demonstration is to enable the audience to interactively experience the proposed query processing paradigm in real-time on large networks. During the visual construction of a subgraph query, the *Interaction Viewer* module shall be enabled to assist users in gaining such experience. Through this module, one will be able to view the generation of candidate (super)graphlets at each visual step and the time taken by QUBLE at each step for fetching candidates (Figure 1(c)) and appreciate the fact that the latency offered by the GUI at each step is sufficient to finish this prefetching task. Furthermore, she will be able to interactively view result matches on a large network by "drilling down" into specific parts of the network in real-time using the *Results Visualizer* (Figure 2). Lastly, we shall interactively demonstrate how QUBLE efficiently handle query modification.

Acknowledgement: Shuigeng Zhou was supported by Research Innovation Program of Shanghai Municipal Education Committee under grant No. 13ZZ003.

5. REFERENCES

[1] S. S. Bhowmick, B. Choi, S. Zhou. VOGUE: Towards A Visual Interaction-aware Graph Query Processing Framework. *In CIDR*, 2013.
[2] L.P. Cordella, P. Foggia, C. Sansone, M. Vento. An improved algorithm for matching large graphs. *In 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, 2001.
[3] J. P. Huan, W. Wang. Efficient Mining of Frequent Subgraph in the Presence of Isomorphism. *In ICDM*, 2003.
[4] C. Jin, et al. GBLENDER: Towards Blending Visual Query Formulation and Query Processing in Graph Databases. *In ACM SIGMOD*, 2010.
[5] C. Jin, et al. GBLENDER: Visual Subgraph Query Formulation Meets Query Processing. *In ACM SIGMOD*, 2011.
[6] C. Jin, et al. PRAGUE: A Practical Framework for Blending Visual Subgraph Query Formulation and Query Processing. *In ICDE*, 2012.
[7] G. Karypis, V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. on Scientific Computing*, 20(1), 1999.
[8] X. Yan, J. Han. gSpan: Graph-based Substructure Pattern Mining. *In ICDM*, 2002.
[9] G. Zhu, X. Lin, K. Zhu et al. TreeSpan: Efficiently Computing Similarity All-Matching. *In SIGMOD*, 2012.