Top-K Interesting Subgraph Discovery in Information Networks

Manish Gupta*, Jing Gao†, Xifeng Yan‡, Hasan Cam§ and Jiawei Han¶
*Microsoft, India. Email: gmanish@microsoft.com
†State University of New York at Buffalo. Email: jing@buffalo.edu
‡University of California, Santa Barbara. Email: xyan@cs.ucsb.edu
§US Army Research Lab. Email: hasan.cam.civ@mail.mil
¶University of Illinois at Urbana-Champaign. Email: hanj@cs.uiuc.edu

***Abstract*—In the real world, various systems can be modeled using heterogeneous networks which consist of entities of different types. Many problems on such networks can be mapped to an underlying critical problem of discovering top-K subgraphs of entities with rare and surprising associations. Answering such subgraph queries efficiently involves two main challenges: (1) computing all *matching* subgraphs which satisfy the query and (2) *ranking* such results based on the rarity and the interestingness of the associations among entities in the subgraphs. Previous work on the matching problem can be harnessed for a naïve ranking-after-matching solution. However, for large graphs, subgraph queries may have enormous number of matches, and so it is inefficient to compute all matches when only the top-K matches are desired. In this paper, we address the two challenges of matching and ranking in top-K subgraph discovery as follows. First, we introduce two index structures for the network: topology index, and graph maximum metapath weight index, which are both computed offline. Second, we propose novel top-K mechanisms to exploit these indexes for answering *interesting subgraph* queries online efficiently. Experimental results on several synthetic datasets and the DBLP and Wikipedia datasets containing thousands of entities show the efficiency and the effectiveness of the proposed approach in computing *interesting subgraphs*.**

I. Introduction

With the ever-increasing popularity of entity-centric applications, it becomes very important to study the interactions between entities, which are captured using edges in the entity-relationship (or information) networks. Entity-relationship networks with multiple types of entities are usually referred to as heterogeneous information networks. For example, bibliographic networks capture associations like 'an author wrote a paper' or 'an author attended a conference'. Similarly, social networks, biological protein-enzyme networks, Wikipedia entity network, etc. also capture a variety of rich associations.

In these applications, it is critical to detect novel connections or associations among objects based on some subgraph queries. Two example problems are shown in Figures 1 and 2, and are described as follows.

P1: Team Selection: Organization networks consist of person and object nodes where two persons are connected if they have worked together on a successful mission in the past, and a person is linked to an object if the person has a known expertise in using that object. For example, US army network which consists of ∼2-3M persons and much more linked objects. A manager in such an organization may have a mission which can be defined by a query graph of objects and persons. For example, the left half of Figure 1 shows an organization network while the right half of the figure shows a sample mission query graph consisting of three persons and a laptop, a microscope and a gun. The network has edges with weights such that a high weight implies higher compatibility between the nodes connected by the edge. The manager is interested in selecting a team to accomplish the mission with the person-person and person-object compatibilities as specified in the mission query. Using the historical compatibility based organization network, how can we find the best team for this mission?

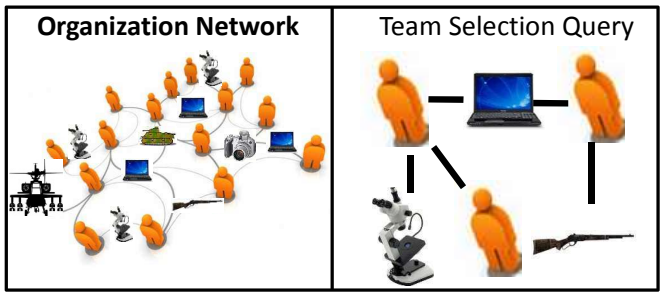

Fig. 1. Team Selection Problem (Interestingness = Highest Historical Compatibility)

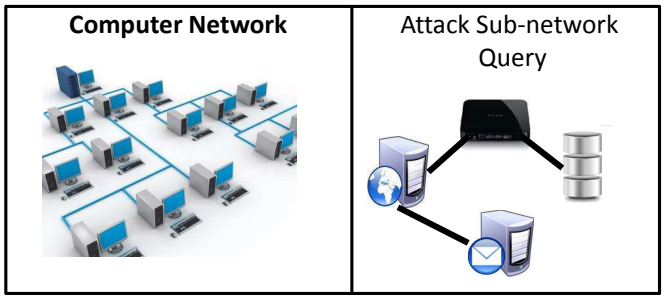

Fig. 2. Attack Localization Problem (Interestingness = Highest Data Transfer Rate)

P2: Attack Localization: Consider a computer network as shown in the left part of Figure 2. It can consist of a large number of components like database servers, hubs, switches, desktops, routers, VOIP phones, etc. Consider a simple attack on multiple web servers in such a network where the attack

script runs on a compromised web server. The script reads a lot of data from the database server (through the network hub) and sends out spam emails through the email server. Such an attack leads to an increase in data transfer rate along the connections in multiple "attack sub-networks" of the form as shown in the right part of the figure. Many such attacks follow the same pattern of increase in data transfer rates. How can a network administrator localize such attacks quickly and find the worst affected sub-networks given the observed data transfer rates across all the links in the network?

Both of these problems share a common underlying problem: Given a heterogeneous network G, a heterogeneous subgraph query Q, and an edge interestingness measure I which defines the edge weight, find the top-K matching subgraphs S with the highest interestingness. The two problems can be expressed in terms of the underlying problem as follows. **P1:** G = organization network, Q = mission query, I = historical compatibility, S = team. **P2:** G = computer network, Q = an "attack sub-network" query, I = data transfer rate, S = critical sub-networks. Besides the two tasks, this proposed problem finds numerous other applications. For example, the *interesting subgraph* matches can be useful in network bottleneck discovery based on link bandwidth on computer networks, suspicious relationship discovery in social networks, de-noising the data by identifying noisy associations in data integration systems, etc.

Comparison with Previous Work

The proposed problem falls into the category of the subgraph matching problems. Subgraph matching has been studied in the graph query processing literature with respect to approximate matches [19], [25], [26], [30] and exact matches [18], [27], [31]. Subgraph match queries have also been proposed for RDF graphs [15], probabilistic graphs [24] and temporal graphs [1]. The proposed problem can be solved by first finding all matches for the query using the existing graph matching methods and then ranking the matches. The cost of exhaustive enumeration for all the matches can be prohibitive for large graphs. Hence, this paper proposes a more efficient solution to the top-K subgraph matching problem which exploits novel graph indexes. Many different forms of top-K queries on graphs have been studied in the literature [5], [21], [23], [28], [30]. Gou et al. [5] solve the problem only for twig queries while we solve the problem for general subgraphs. Yan et al. [21] deal with the problem of finding top-K highest aggregate values over their h-hop neighbors, in which no subgraph queries are involved. Zhu et al. [28] aim at finding top-K largest frequent patterns from a graph, which does not involve a subgraph query either. Different from existing top-K work, the proposed work deals with a novel definition of top-K general subgraph match queries, which has a large number of practical applications as discussed above.

Brief Overview of Top-K Interesting Subgraph Discovery

Given a heterogeneous network containing entities of various types, and a subgraph query, the aim is to find the top-K matching subgraphs from the network. We study the following two aspects of this problem in a tightly integrated way: (1) computing all possible matching subgraphs from the network, and (2) computing interestingness score for each match by aggregating the weights of each of its edges. To solve these problems, we present an efficient solution which exploits two low-cost index structures (a graph topology index and a maximum metapath weight (MMW) index) to perform top-K ranking while matching (RWM). Multiple applications of the top-K heuristic, a smart ordering of edges for query processing, quick pruning of the edge lists using the topology index and the computation of tight upper bound scores using the MMW index contribute to the efficiency of the proposed solution in answering the top-K *interesting subgraph* queries.

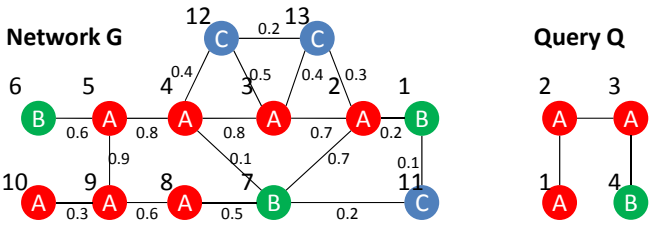

Fig. 3. Example of a Network G and a Query Q

Summary

We make the following contributions in this paper.

- We propose the problem of top-K interesting subgraph discovery in information networks given a heterogeneous edge-weighted network and a heterogeneous unweighted query.
- To solve this problem, we propose two low-cost indexes (a graph topology index and a maximum metapath weight (MMW) index) which summarize the network topology and provide an upper bound on maximum metapath weights separately.
- Using these indexes, we provide a ranking while matching (RWM) algorithm with multiple applications of the top-K heuristic to answer *interesting subgraph* queries on large graphs efficiently.
- Using extensive experiments on several synthetic datasets, we compare the efficiency of the proposed RWM methodology with the simple ranking after matching (RAM) baseline. We also show effectiveness of RWM on two real datasets with detailed analysis.

Our paper is organized as follows. In Section II, we define the *top-K interesting subgraph discovery* problem. The proposed approach consists of two phases: an offline index construction phase and an online query processing phase which are detailed in Sections III and IV respectively. In Section V, we discuss various general scenarios in which the proposed approach can be applied. We present results with detailed insights on several synthetic and real datasets in Section VI. We discuss related work and summarize the paper in Sections VII and VIII respectively.

II. Problem Definition

In this section, we formalize the problem definition and present an overview of the proposed system. We start with an introduction to some preliminary concepts.

Definition 1 (A Heterogeneous Network)**.** *A heterogeneous network is an undirected graph* $G = \langle V_G, E_G, type_G, weight_G \rangle$ *where* V_G *is a finite set of vertices (representing entities) and* E_G *is a finite set of edges each being an unordered pair of distinct vertices.* $type_G$ *is a function defined on the vertex set as* $type_G : V_G \rightarrow \mathcal{T}_G$ *where* $\mathcal{T}_G$ *is the set of entity types and* $|\mathcal{T}_G| = T$. $weight_G$ *is a function defined on the edge set as* $weight_G : E_G \rightarrow \mathbb{R} \in [0, 1]$. $weight_G(e)$ *represents the interestingness measure value associated with the edge* e.

For example, Figure 3 shows a network G with three types of nodes. $\mathcal{T}_G = \{A, B, C\}$. $|V_G|$=13, and $|E_G|$=18.

Definition 2 (Subgraph Query on a Network)**.** *A subgraph query* Q *on a network* G *is a graph consisting of node set* V_Q *and edge set* E_Q. *Each node could be of any type from* $\mathcal{T}_G$.

For example, Figure 3 shows a query Q with four nodes. $|V_Q|$=4, and $|E_Q|$=3. The network G and the query Q shown in Figure 3 will be used as a running example throughout this paper.

Definition 3 (Subgraph Isomorphism)**.** *A graph* $g = \langle V_g, E_g, type_g \rangle$ *is subgraph isomorphic to another graph* $g' = \langle V_{g'}, E_{g'}, type_{g'} \rangle$ *if there exists a subgraph isomorphism from* g *to* g'. *A subgraph isomorphism is an injective function* $M : V_g \rightarrow V_{g'}$ *such that (1)* $\forall v \in V_g$, $M(v) \in V_{g'}$ *and* $type_g(v)$=$type_{g'}(M(v))$, *(2)* $\forall e$ =$(u, v) \in E_g$, e' =$(M(u), M(v))$$\in E_{g'}$.

Definition 4 (Match)**.** *The query graph* Q *can be subgraph isomorphic to multiple subgraphs of* G. *Each such subgraph of* G *is called a match or a matching subgraph of* G.

The query Q can be answered by returning all exact matching subgraphs from G. For example, the subgraph of G induced by vertices (8, 9, 5, 6) is a match for the query Q on network G shown in Figure 3. For sake of brevity, we will use the vertex set (tuple notation) to refer to the subgraph induced by the vertex set.

Definition 5 (Interestingness Score)**.** *The interestingness score for a match* M *for a query* Q *in a graph* G *is defined as the sum of its edge weights.*

For example, the interestingness score for the occurrence {8, 9, 5, 6} is 2.1. Though we use sum as an aggregation function here, any other monotonic aggregation function could also be used.

Definition 6 (Top-K Interesting Subgraph Discovery Problem)**.**
Given*: A heterogeneous information network* G, *a heterogeneous unweighted query* Q, *and an edge interestingness measure.*
Find*: Top-*K *matching subgraphs with highest interestingness scores.*

For example, (3, 4, 5, 6) and (4, 3, 2, 7) are the top two matching subgraphs both with the score 2.2 for the query Q on network G in Figure 3.

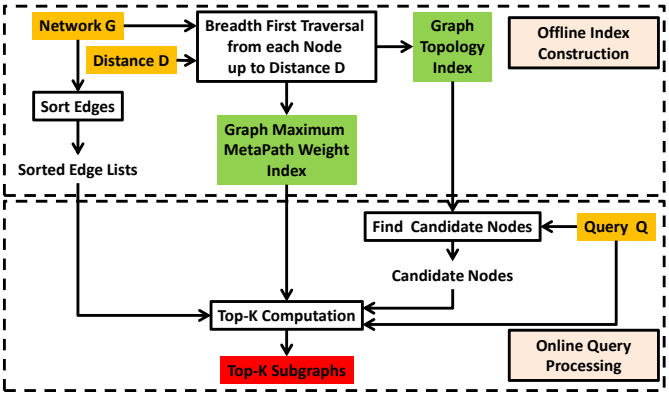

Fig. 4. Top-K Interesting Subgraph Discovery System Diagram

Baseline Method: Ranking After Matching (RAM)

A naïve way to solve the top-K interesting subgraph discovery problem is to first perform matching to discover all the subgraph matches from G, and then rank these matches. For example, for the query Q in Figure 3, 9 matches can be computed as follows (10, 9, 8, 7), (2, 3, 4, 7), (4, 3, 2, 1), (10, 9, 5, 6), (9, 5, 4, 7), (5, 9, 8, 7), (8, 9, 5, 6), (4, 3, 2, 7), and (3, 4, 5, 6). Next, the interestingness score can be computed for each of the matches by adding up the edge weights as 1.4, 1.6, 1.7, 1.8, 1.8, 2.0, 2.1, 2.2, and 2.2 respectively. If K=2, the matches (4, 3, 2, 7), and (3, 4, 5, 6) can be returned as the most interesting subgraphs for query Q.

However, since only the top 2 interesting subgraphs are desired, this approach is not efficient especially for large graphs for which the number of matches could be enormous and hence computing all the matches could be very time consuming. Hence, we propose a top-K approach which performs the interestingness score computation (and hence ranking) while matching (RWM). For this approach we need to make use of a few index structures which are constructed offline. These index structures are then exploited to efficiently answer the queries online.

System Overview

Figure 4 shows a broad overview of the proposed system. Given a user query Q, the top half denotes the offline preprocessing phase in which the two indexes are computed, while the lower half denotes the online processing phase in which the user query is processed using the two indexes. The index parameter, distance D, controls the size and the index construction time. We discuss the details of the offline index construction and the online query processing in Sections III and IV respectively.

III. Offline Index Construction

In this section, we will first discuss the details of the index structures in Section III-A and then study their construction time and space in Section III-B.

A. Index Structures

To support a ranking while matching (RWM) framework, we first propose two novel index structures in this subsection. We will use these index structures to develop a top-K methodology in Section IV.

Graph Topology Index

The graph topology index for a graph G captures the structure of the graph. It stores for every node n, the number of d-hop neighbors of each type for all $d \in \{1, \ldots, D\}$ along a particular metapath corresponding to a path of length d originating from node n, where a metapath is defined as follows.

Definition 7 (Metapath)**.** *A path $u \rightsquigarrow u'$ from a node u to a node u' in a graph $G = \langle V_G, E_G, type_G, weight_G \rangle$ is a sequence $(v_0, v_1, \ldots, v_k)$ of vertices such that $u = v_0$ and $u' = v_k$, and $(v_{i-1}, v_i) \in E_G$. The length of a path is equal to the number of edges in the path (k). If the node type is used instead of its id, for each node in the path, the path is called a metapath. Thus, the corresponding metapath is $(type_G(v_0), type_G(v_1), \ldots, type_G(v_k))$.*

For example, the metapath corresponding to the path (5, 4, 7) is (A, A, B). There are T^D distinct metapaths of length D where T is the number of types.

Each column of a topology index corresponds to a metapath. Figure 5 shows the graph topology index for the first 4 nodes of the graph shown in Figure 3. For example, for node 2, there are two 2-hop neighbors of type A (4 and 8) reachable via the metapath (B, A). Hence the entry $topology(2, (B, A))$=2. A blank entry in the index indicates that there is no node of type t at a distance d from node n along the corresponding metapath. As we shall see in Section IV-A, the topology index plays a crucial role in reducing the search space by pruning away candidate graph nodes that cannot be instantiated for a given query node.

d→	1			2								
Node Id↓	A	B	C	AA	BA	CA	AB	BB	CB	AC	BC	CC
1	1		1	1			1		1	1		
2	1	2	1	1	2	1				2	1	1
3	2		2	1		2	2			2		2
4	2	1	1	2	2	1	1			2	1	1

Fig. 5. Graph Topology Index for the first 4 Nodes

d→	1			2								
Node Id↓	A	B	C	AA	BA	CA	AB	BB	CB	AC	BC	CC
1	0.2		0.1	0.9			0.9		0.3	0.5		
2	0.7	0.7	0.3	1.5	1.2	0.7				1.2	0.9	0.5
3	0.8		0.5	1.6		0.9	1.4			1.2		0.7
4	0.8	0.1	0.4	1.7	0.8	0.9	1.4			1.3	0.3	0.6

Fig. 6. MMW Index for the first 4 Nodes

Maximum Metapath Weight Index

The maximum metapath weight (MMW) index has the same size as the graph topology index. It stores the maximum sum of weights to any node of type t along a particular metapath of length d originating from the node n. Figure 6 shows the maximum metapath weight index for the first 4 nodes of the graph shown in Figure 3. For example, for node 2, there are two 2-hop neighbors of type A (4 and 8) reachable via the metapath (B, A) with weights 0.8 and 1.2 respectively. Hence, the corresponding index entry $MMW(2, (B, A))$ is 1.2. In Section IV-C, we will show how the MMW index can be used for computation of tighter upper bound scores which leads to highly effective pruning of the partially grown candidate solutions.

Q_1	Q_2	Q_3	Q_4
2	2	2	1
3	3	3	6
4	4	4	7
5	5	5	
8	8	8	
9	9	9	
10	10	10	

Fig. 7. Potential Candidates for each Query Node (Filled Cells represent Filtered Candidates)

AA	BB	CC	AB	AC	BC
(5,9):0.9		(12,13):0.2	(2,7): 0.7	(3,12): 0.5	(7,11): 0.2
(3,4):0.8			(5,6): 0.6	(4,12): 0.4	(1,11): 0.1
(4,5):0.8			(8,7): 0.5	(3,13): 0.4	
(2,3):0.7			(2,1): 0.2	(2,13): 0.3	
(8,9):0.6			(4,7): 0.1		
(9,10):0.3					

Fig. 8. Sorted Edge Lists for Graph in Figure 3

Sorted Edge Lists

Besides the above two indexes we also maintain sorted edge lists which capture the interestingness of the edges in the graph. For each edge-type, all the edges of that type are stored in the non-ascending order of their interestingness values. Thus, the most interesting edges occur at the top of the lists. Figure 8 shows the sorted edge lists for the graph shown in Figure 3.

These edge lists are further indexed by nodes. For every edge list, a hash is maintained which maps each graph node to the set of rows in the edge list which contain the graph node. For example, for the edge list $A - A$, the node pointer from node 4 points to edges (3, 4) and (4, 5) respectively. Such pointers provide a fast access to the matching edges of a particular type for a particular node.

B. Index Construction

We can compute both the index structures and the sorted edge lists offline as follows. For constructing the graph topology index and the maximum metapath weight index, a breadth first traversal needs to be performed originating from each node of the graph. For each node, each of its d hop neighbors are visited up to a maximum distance D. For the graph topology index, the breadth first traversal maintains the current frontier of the visited nodes in a queue. After each hop of the traversal, the actual paths from the origin node are expressed in terms of their corresponding metapaths and the topology index is updated based on the number of unique endpoints along a particular metapath. To update the MMW index, the sum of edge weights is computed for each path and an entry in the MMW index is updated with the maximum weight of any path for the corresponding metapath. Updating the MMW index needs exhaustive enumeration of all paths. However, for small values of D, this is not very expensive. Also, note that the index construction needs to be done just once and is an offline task. If B is the average number of neighbors for a node, the total number of d-hop neighbors up to D is $O(B^D)$. Thus, the computation of the MMW index and the topology index takes $O(|V_G|B^D)$ time. The space required to store each of the two indexes is $O(|V_G|T^D)$ where T is the number of types.

As the number of types increases, the size of the two indexes can bloat very quickly. However, most of the practical heterogeneous information networks have few node types, and

also follow a schema. The schema can itself restrict the number of edge types to a very few. Besides this, in Section V, we will discuss various ways in which we can reduce the size of these indexes without much losses in efficiency.

The sorted edge lists are created by grouping edges by type and sorting the edges within each type in a non-ascending order. If there are T types of nodes in the graph, the number of types of edges is $\frac{T(T+1)}{2}$. Time to sort an edge list of length L is $O(L \times logL)$. It is easy to see that the time to sort $\frac{T(T+1)}{2}$ lists such that their total size is E_G is maximum when each list is of the size $\frac{2|E_G|}{T(T+1)}$. Hence, the overall time complexity of computing the sorted edge lists is $O\left(|E_G| log\left(\frac{2|E_G|}{T(T+1)}\right)\right)$. The space required to store the index is $O(|E_G|)$. Also building the companion graph node pointers to rows in edge lists takes $O(|E_G|)$ time and space.

IV. Top-K Interesting Subgraph Query Processing

Given a query Q with node set V_Q and edge set E_Q, top-K matching subgraphs are discovered by traversing the sorted edge lists in the top to bottom order with the following speedup heuristics. First for each node in V_Q, a set of nodes from the graph that could be potential candidates for the query node, is identified using the topology index (Algorithm 1). The edges in the sorted edge lists that contain nodes other than the potential candidate nodes are marked as invalid. This prunes away many edges and speeds up the edge list traversal. The query Q is then processed using these edge lists in a way similar to the top-K join query processing (Section IV-B) adapted significantly to handle network queries. The approach discussed in Section IV-B is further made faster by the tighter upper bound scores computed using the MMW index (Algorithm 2). We will discuss these in detail in this section.

A. Candidate Node Filtering using Topology Index

Here, we will discuss a methodology to reduce the candidate search space by pruning away candidate graph nodes that cannot be instantiated for a given query node, using the topology index. The top-K query processing involves traversing the edge lists from top to bottom. During this traversal, the top-K pruning can improve if some edge entries in the edge lists can be marked as invalid thereby reducing the effective size of the edge lists.

Pruning Example

In Figure 3, the query consists of four nodes: Q_1, Q_2, Q_3, Q_4. The matching candidate graph nodes with respect to the node type are as follows. (2, 3, 4, 5, 8, 9, 10) for Q_1, (2, 3, 4, 5, 8, 9, 10) for Q_2, (2, 3, 4, 5, 8, 9, 10) for Q_3, and (1, 6, 7) for Q_4. In the query, we see that the node Q_2 is connected to 2 nodes of type A at distance 1. However, the row corresponding to the node 2 in the topology index indicates that there is just one neighbor of type A at distance 1. Thus, node 2 cannot be an instantiation of the query node Q_2 in any of the matches. Similarly, we observe that the nodes 8 and 10 also cannot be potential candidates for the query node Q_2. Thus, the set of potential candidates reduces to those shown in Figure 7. The nodes in red-filled cells can be filtered out.

The potential candidates can be identified for a query node q by first computing the topology structure for q (similar to a row in the graph topology index) and then verifying if the query topology for query node q fits as a subgraph of the graph topology with respect to a potential candidate node p in the graph. This topology fit can be checked by considering all the paths in the query with length from 1 to D (the index parameter) and verifying their presence in the graph. Let us denote the topology index structure for the query by $queryTopology$.

Algorithm 1 Candidate Node Filtering Algorithm

Input: (1) Query Node q, (2) Graph Node p, (3) $topology[p]$, (4) $queryTopology[q]$, (5) Index Parameter D
Output: Is p a potential candidate node for query node q?
1: **for** $d = 1 \ldots D$ **do**
2: **for** $mp = 1 \ldots T^d$ **do**
3: **if** $queryTopology[q][d][mp] > topology[p][d][mp]$ **then**
4: Return False
5: Return True

Candidate Node Filtering Algorithm

The proposed candidate node filtering approach is summarized in Algorithm 1. For each distance value d, all possible metapaths of length d are checked. By comparing the $topology$ for all metapaths with the corresponding $queryTopology$ values (Step 3), it can be inferred whether the candidate p is valid to be an instantiation of query node q for some match. The time complexity is $O(DT^{D+1})$.

Candidate pruning leads to shortening of the edge lists associated with any of the query edges. For example, nodes 2, 8 and 10 get pruned for the query node Q_2. Thus, the edge list corresponding to the query edge (Q_2, Q_3) will have the following AA edges marked as invalid: (2,3), (8,9) and (10,9).

B. Top-K Match Computation

In this sub-section, we describe the top-K algorithm to perform interestingness scoring and matching simultaneously. The algorithm is based on the following key idea. A top-K heap is maintained which stores the best K answers seen so far. The sorted edge lists are traversed from top to bottom. Each time an edge with maximum edge weight from any of the lists is picked and all possible size-1 matches in which that edge can occur are computed. Candidate size-1 matches are grown one edge at a time till they grow to the size of the query. Partially grown candidate matches can be discarded if the upper bound score of these matches falls below the minimum element in the top-K heap. The algorithm terminates when no subgraph using the remaining edges can result into a candidate match with upper bound score greater than the minimum element in the top-K heap. We discuss the details below.

Definition 8 (Valid Edge)**.** *A valid edge e with respect to a query edge qE is a graph edge such that both of its endpoints are contained in the potential candidate set for the correspond-*

ing query nodes in qE. Recall that the potential candidate set for each query node is computed using Algorithm 1.

The sorted edge lists are quite similar to the lists in Fagin's TA [4]. To traverse the edge lists in the top to bottom order, a $pointer$ is maintained with every edge list. The pointers are initialized to point to the topmost graph edge in the sorted edge list, which is valid for at least one query edge. As the pointers move down the lists, they move to the next valid edge rather than moving to the next edge in the list (as in Fagin's TA).

Definition 9 (Size-c candidate match)**.** *A size-c candidate match is a partially grown match such that c of its edges have been instantiated using the matching graph edges.*

Generating Size-1 Candidate Matches

The processing starts by picking the edge type pointing to the edge with the maximum score in the edge lists as ET. The graph edge e (with the max score) is then instantiated for all query edges of type ET to form multiple size-1 candidate matches. Size-1 candidate matches are gradually grown to a larger size match containing more edge instantiations, one edge at a time. Note that if the edge type ET has both end points of the same node type, then the graph edge e could match the query edge qE in 2 ways. This case is hence handled by creating another new candidate with the reversed graph edge $reverse(e)$.

For example, instantiation of edge (5,9) in Figure 3 results into four size-1 candidate matches: $(Q_1 - 9 - 5 - Q_4)$, $(5 - 9 - Q_3 - Q_4)$, $(Q_1 - 5 - 9 - Q_4)$, and $(9 - 5 - Q_3 - Q_4)$.

Actual Score and Upper Bound Score (UBScore) of a Candidate Match

Given a size-c candidate match, it can either be pruned off because its upper bound score is less than the least element in the top-K heap or it can be grown further or put into the heap. To make this decision, the upper bound score needs to be computed. At any point during the processing, $CurrCandidates$ contains the candidate matches each of which contains instantiated edges listed in a set called $ConsideredEdges$. The actual score of each such candidate match is simply the sum of the weights of all the instantiated edges. The upper bound score of the candidate is its actual score plus the upper bound score of each of its non-instantiated edges. Further, the upper bound score of a non-instantiated query edge qE of type (t_1, t_2) is computed as the maximum score of any graph edge of type (t_1, t_2) compatible with the current edges in the candidate match and lying below the current pointer position in the edge list (t_1, t_2).

Growing the Candidate Matches

If the heap contains at least K elements, and if the upper bound score of a candidate match is less than the minimum element in the heap, the candidate match is pruned off. The next query edge to be instantiated, qE', for all the candidates in any iteration should be selected such that it does not belong to the set of $ConsideredEdges$ but is linked to some edge in $ConsideredEdges$.

Maintaining the Top-K Heap

For each fully computed match in $CurrCandidates$, if the score for the candidate is greater than the minimum element in the heap, the new candidate is added to the heap and the minimum element is removed from the heap. Also, after the processing for all query edges in $QueryEdges$ has been finished, the pointer is moved to the next valid edge in the edge list of type ET. Before proceeding to the next valid edge to be processed, an upper bound of any possible candidate match is computed by simply summing up the upper bound score for all edges in the query. If the upper bound score for any potential candidate is less than the minimum element in the heap (which we refer to as the global top-K quit check), the algorithm terminates. At this point, the heap contains the desired top-K matches for the query Q.

For example, after processing the edge $(4, 5)$, the heap contains two elements both with score 2.2. At this stage, the edge pointers point to the edges $(2, 3)$ and $(2, 7)$. The maximum upper bound score now is 0.7 (due to edge $(2, 3)$)+0.6 (due to edge $(8, 9)$)+0.7 (due to edge $(2, 7)$)=2.0 which is $<$ 2.2 and hence processing can be stopped.

C. Faster Query Processing using Graph Maximum Metapath Weight Index

In Section IV-B, the upper bound score for partially instantiated candidates is computed by summing up the actual score for the considered edges and the upper bounds for the non-considered edges. Is it possible to have a tighter bound for the non-considered edges? A tighter bound will help in more aggressive pruning of the candidate matches and thereby make the top-K query processing faster.

Computing Upper Bounds using Paths rather than Edges

Consider the query Q' as shown in Figure 9. The top right part shows the size-1 candidate match where a graph edge was used to instantiate the query edge (Q_1, Q_2). So, the actual score of the candidate is simply the edge weight corresponding to the instantiation. In Section IV-B, the upper bound score was computed as the sum of the actual score for the edge corresponding to (Q_1, Q_2) and the upper bound scores each for the edges (Q_1, Q_3), (Q_3, Q_4), (Q_4, Q_5) and (Q_2, Q_3). However, a stricter upper bound can be obtained if we could compute the upper bound score as the sum of the actual score for the edge corresponding to (Q_1, Q_2) and the upper bound scores for the paths (Q_1, Q_3, Q_4, Q_5) and (Q_2, Q_3).

This results into the following two problems. (1) How do we split the set of the non-considered query edges into paths? (2) How do we compute the upper bound scores for these paths?

To answer the second question, consider a path $p = (v_1, v_2, \ldots, v_n)$ of length n in the query. Let the corresponding metapath be $t = (t_1, t_2, \ldots, t_n)$. Suppose that the query edge (v_1, v_2) has been instantiated with the graph edge (u_1, u_2). We can estimate the upper bound score for the path in the partial candidate match as the actual edge weight of (u_1, u_2)+$MMW[u_2][t_3, t_4, \ldots, t_n]$. Thus, the upper bound

score of a path can be computed using the maximum metapath weight index.

To answer the first question, the query needs to be split into paths which satisfy the following criteria.

- The path must originate from an instantiated node. This is necessary because we can use the MMW index only if the origin of the path has been instantiated.
- The paths should not overlap with each other or with the already instantiated edges so as to obtain a stricter upper bound.
- The length of each path should be less than the index parameter D.

Greedy Path Set Selection

A partial instantiated match consists of instantiated edges IE, instantiated nodes IN which is the set of nodes covered by the instantiated edges, and a set of non-instantiated edges. The method starts by first enumerating all possible paths that can cover the non-instantiated edges originating from nodes$\in IN$ and satisfying the above criteria. The union of all such paths across all instantiated query nodes is called $AllPaths$. Though we can design more principled benefit-cost based method to select the set of paths from $AllPaths$, we resort to a greedy method for the sake of efficiency. Thus, the paths are selected one by one from this set of paths in a greedy manner such the longest path is selected at each step. After selecting a path, the set of available paths $AllPaths$ is updated by removing all the paths that overlap with the already selected paths. The algorithm stops when $AllPaths$ becomes empty.

However such a greedy selection of paths does not guarantee that all non-instantiated edges will get covered. For example, in Figure 10 (with D=4), the edge (Q_1, Q_2) is already instantiated. Next the processing selects the longest path (with length $\leq MMW$ index parameter) originating from the instantiated nodes such that it does not overlap with any of the instantiated edges or other paths. Hence, the path $(Q_1, Q_3, Q_4, Q_5, Q_7)$ is chosen and then the path (Q_2, Q_3). Now, the edges (Q_4, Q_6) and (Q_6, Q_7) need to be considered separately since they cannot be covered by any of the paths originating from the instantiated nodes. Thus, the query is actually split into three disjoint sets: (1) already instantiated edges, (2) edges on the selected paths, and (3) extra edges not covered by (1) or (2).

Path Based Upper Bound Score Computation

The approach is summarized in Algorithm 2. The upper bound score of the candidate is initialized to the sum of the edge weights for all instantiated edges. For each instantiated node, the set of all paths originating from this node is computed (Step 4). This set excludes any path that contains any of the already instantiated edges. Until all the query edges are not covered, the longest path $maxPath$ is chosen from $AllPaths$ (Step 6). Edges from the $maxPath$ are added to the set of instantiated edges (Step 7) and the upper bound score is updated with the upper bound score of $maxPath$ by looking up the appropriate entry from the MMW index (Step 8). $AllPaths$ is updated by removing all the paths that overlap with $maxPath$ (Step 9). Finally when there are no more paths left in $AllPaths$, the upper bound score of the candidate match is updated by adding the upper bound score of the query edges not yet covered (Step 10).

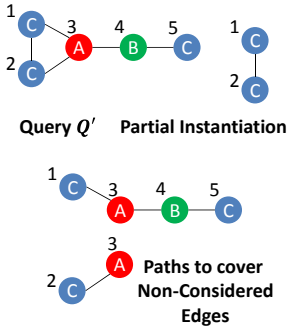

Fig. 9. Estimating Upper Bound using Paths (Non-Considered Query Edges Covered by Paths)

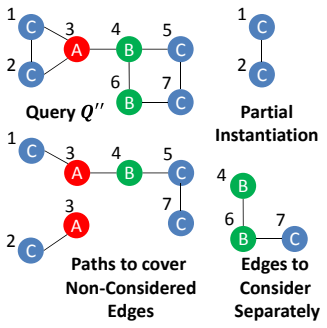

Fig. 10. Estimating Upper Bound using Paths (Non-Considered Query Edges Covered by Paths and Extra Edges)

Algorithm 2 Path Based Upper Bound Score Computation

Input: (1) Query Q, (2) Instantiated Edges IE, (3) Instantiated Graph Nodes IN, (4) MMW Index, (5) $SortedEdgeLists$
Output: Upper Bound Score $UBScore$

1: $UBScore \leftarrow \sum_{ie \in IE} weight_G(ie)$.
2: $AllPaths \leftarrow \phi$.
3: **for** each instantiated node $n \in IN$ **do**
4: $AllPaths \leftarrow AllPaths \cup$ Paths in query graph originating from n, length$\leq D$ and do not contain edges from IE.
5: **while** $\exists$ a query edge not yet covered or $AllPaths \neq \phi$ **do**
6: $maxPath \leftarrow$Compute path with maximum length from $AllPaths$.
7: Add edges from $maxPath$ to IE.
8: $UBScore \leftarrow UBScore+$ Score of $maxPath$ using MMW.
9: Remove paths from $AllPaths$ containing edges in $maxPath$.
10: $UBScore \leftarrow UBScore+$ UBScore for edges $\in Q$ but $\notin IE$.

Claim 1. *Pruning using the maximum metapath weight (MMW) index is more effective compared to the pruning using just the sorted edge lists.*

When computing the upper bound using the sorted edge lists, the upper bound for a non-instantiated edge can be arbitrarily high if there is any unprocessed valid edge anywhere in the entire network with a high weight. However, because of the MMW index, the upper bound score for the non-instantiated edges gets restricted to the local neighborhood of the already instantiated edges thereby restricting the overall upper bound score, which in turn results in more effective pruning.

The MMW index can indeed help prune candidate matches which cannot be pruned using the sorted edge lists. For example, consider the candidate $(9 - 5 - Q_3 - Q_4)$ and recall that the heap contains the two elements with scores 2.1 and 2.0 at this stage. Now, the upper bound score for this candidate match using the edge lists is 0.9+0.8+0.7=2.4 and since $2.4 > 2.0$ it cannot be pruned off. On the other hand, the upper bound score using the MMW index is 0.9+MMW(5, (A, B))=0.9+0.9=1.8 and since $1.8 < 2.0$ the candidate match can be pruned off.

Time Considerations

Though the computation of the upper bound score using the MMW index results into a stricter upper bound, the

computation may itself consume a lot of time. Hence, this optimization needs to be used sparingly. The maximum benefit of such an optimization occurs when the candidate size is small (e.g., only one edge). Hence, the upper bound for only the size-1 candidates is computed using this path-based method while the edge-based method (Section IV-B) is still used for other cases. After computing the path based UBScore, it is compared with the edge based UBScore and the one which represents a tighter upper bound is used.

V. Discussions

In this section, we discuss various general scenarios in which the proposed approach can be applied.

Queries with Multiple Edge Semantics

In this paper, we considered queries such that the interestingness of every edge in the query carries the same semantics. However, we might need to address the cases where the semantics are very different across the different edges in the query and the graph. For example, consider a query: Find an interesting combination of a movie and 2 persons where the first person is the director of the movie and the second person is an actor in the movie. Now, the query consists of 2 edges: both are movie-person edges, however the relationship for the first edge is "director" while for the second edge, the relationship is "actor". Such queries can still be answered using the proposed system by defining metapaths in terms of edge labels rather than node types.

Directed and Homogeneous Graphs

We presented the ideas based on undirected graphs and undirected queries. However, the approach is general enough to work with directed graphs and directed queries. Trivial updates are needed for the index construction and the candidate match growth to make them direction sensitive. Also, when T=1, the system conforms to the setting of homogeneous networks. In the case of homogeneous networks, there will be a single edge list, if all the relationships have the same semantics.

Weighted Query Edges

Weighted query edges can have two semantics. Weights can be assigned to a query to signify the expected amount of interestingness on each edge. The interestingness score of an instantiated edge e with weight w for a query edge qE with weight qW can then be computed as some function of w and qW, e.g., the squared error, $(w - qW)^2$. Such edge-weighted queries can still be handled by the proposed system as long as the function is a monotonic function and has a well defined upper bound. Another semantics of edge weights is to specify how much importance an edge carries in the query. Thus, a user can specify that the interestingness with respect to say the edge (Q_1, Q_2) is more than the interestingness with respect to the edge (Q_2, Q_3) in Figure 3. In that case, the interestingness of a subgraph can be computed as a *weighted* sum of the interestingness of its edges rather than simply the sum of interestingness of its edges. Again, this can be easily implemented in the proposed framework by multiplying the edge interestingness scores with the appropriate user-specified weights when computing the interestingness scores. Note that for both the cases discussed above, the MMW index cannot be used as the path scores cannot be estimated any more because the edge interestingness is now defined based on the query.

Faster Computations versus Index Size

If the number of types is high and the graph is very dense, the size of the topology and the MMW index could bloat quickly with increasing D. Even with small D storing the metapaths improves the pruning capability but consumes memory. Various schemes could be used to serve as a trade off between the index size and the pruning capability (and hence computational efficiency) achievable due to index usage. One approach is to index only the destination node of the metapaths rather than the actual metapath itself. This reduces the index size from $O(|V_G|T^D)$ to $O(|V_G|TD)$. However, the pruning capability reduces too due to looser upper bounds on interestingness scores. A mix of the two schemes could also be used where the path level information is stored for $d = 1$ to $d = D_0$ and then only the destination node level information is stored for $d = D_0 + 1$ to $d = D$ where D_0 is decided based on a memory-vs-efficiency trade off. We plan to explore this trade off further as part of future work.

Another way of reducing the size of the topology and the MMW indexes is by storing only a selective few metapaths rather than storing columns for every metapath. One scheme could be to store only the columns corresponding to the most frequent metapaths in the entire network. Most frequent metapaths can be identified when performing breadth first traversal (for MMW index construction) itself. The intuition is that if a metapath is rare and if the query contains that metapath, then the number of matches would be low too, and then the top-K algorithm may not be efficient anyway. On the other hand, further flexibility can be obtained by storing information for different metapaths for different nodes, i.e., store information for node-wise most frequent metapaths. Intuitively, a nodewise scheme could provide a better size versus efficiency tradeoff compared to the reduced columns scheme.

VI. Experiments

We perform experiments on multiple synthetic datasets each of which simulates power law graphs. We evaluate the results on the real datasets using case studies. We perform a comprehensive analysis of the objects in the top subgraphs returned by the proposed algorithm to justify their interestingness. Data and code is available at http://dais.cs.uiuc.edu/manish/RWM/.

A. Synthetic Datasets

We construct 4 synthetic graphs using the R-MAT graph generator in GT-Graph software [2]: $G1$, $G2$, $G3$ and $G4$ with 10^3, 10^4, 10^5, and 10^6 nodes respectively. Each graph has a number of edges equal to 10 times the number of nodes. Thus, we consider graphs with exponential increase in graph size. Each node is assigned a random type from 1 to 5. Also, each edge is assigned a weight chosen uniformly randomly between 0 and 1. All the experiments were performed on an Intel Xeon

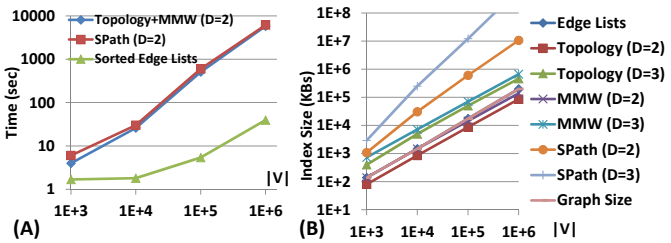

Fig. 11. (A) Index Construction Times (B) Size Comparison of Various Indexes

CPU X5650 4-Core 2.67GHz machine with 24GB memory running Linux 3.2.0. The code is written in Java. The distance parameter D for the indexes is set to 2 for both the proposed approach *RWM* (Ranking While Matching) and the baseline *RAM* (Ranking After Matching), unless specified explicitly. Also unless specified explicitly, we are interested in computing top 10 interesting subgraphs (K=10) and the execution times mentioned in the tables and the plots are obtained by repeating the experiments 10 times.

Baseline: Ranking After Matching (RAM)

The problem of finding the matches of a query Q in a heterogeneous network G has been studied earlier [20], [27]. In [27], the authors present an index structure called SPath. SPath stores for every node, a list of its typed-neighbors at a distance d for $1 \leq d \leq D$. SPath index is then used to efficiently find matches for a query in a path-at-a-time way: the query is first decomposed into a set of shortest paths and then the matches are generated one path at a time. This method is used as a baseline.

Index Construction Time

Figure 11 (A) shows the index construction times for the various indexes. Generating the sorted edge lists is very fast. Even for the largest graph with a million nodes, the sorted edge lists creation takes around 40 seconds. The Topology+MMW (D=2) and SPath (D=2) curves show the time required for construction of these indexes, for various graph sizes. The X axis denotes the number of nodes in the synthetic graphs and the Y axis shows the index construction time in seconds. Note the Y axis is plotted using a log scale.

The index construction time rises linearly as the graph size grows. Also, as expected the index construction time rises as D increases.

Index Size

Figure 11 (B) shows the size of each index for different values of D. The X axis plots the number of nodes in the synthetic graphs and the Y axis plots the size of the index (in KBs) using a logarithmic scale. Different curves plot the sizes of various indexes, and the graph. Note that the size of the topology index and the MMW index for D=2 is actually smaller than the size of the graph. Even when the index parameter is increased to D=3, the topology and the MMW indexes remain much smaller than the SPath index for D=2. For D=3, the SPath index grows very fast as the size of the graph increases. As expected as the graph size increases, the size of each index increases. While the increase is manageable for the Edge lists, the MMW index and the topology index, the increase in SPath index size is humongous.

	$\|V_Q\|$=2	$\|V_Q\|$=3	$\|V_Q\|$=4	$\|V_Q\|$=5
RAM	245	2004	14628	169328
RWM0	15	32	43	122
RWM1	19	36	98	178
RWM2	20	40	442	6887
RWM3	218	1733	2337	3933
RWM4	18	34	42	118

TABLE I
QUERY EXECUTION TIME (MSEC) FOR PATH QUERIES (GRAPH $G2$ AND INDEXES WITH D=2)

	$\|V_Q\|$=2	$\|V_Q\|$=3	$\|V_Q\|$=4	$\|V_Q\|$=5
RAM	144	8698	34639	174992
RWM0	10	375	14689	229136
RWM1	13	446	16754	200065
RWM2	12	562	19088	201708
RWM3	156	2277	17182	161533
RWM4	11	346	13547	199616

TABLE II
QUERY EXECUTION TIME (MSEC) FOR CLIQUE QUERIES (GRAPH $G2$ AND INDEXES WITH D=2)

Query Execution Time

We experiment with three types of queries: path, clique and general subgraphs, of sizes from 2 to 5. We present a comparison of different techniques for the graph $G2$ using the indexes with D=2. The tables I- III show the average execution times for an average of 10 queries per experimental setting each repeated 10 times. The six different techniques are as follows: RAM (the ranking after matching baseline), RWM0 (without using the candidate node filtering), RWM1 (without using the MMW index), RWM2 (same as RWM1 without the pruning any partially grown candidates), RWM3 (same as RWM1 without the global top-K quit check), RWM4 (same as RWM1 with the MMW index). Clearly, RAM takes much longer execution times for all types of queries. We observed that the larger the number of candidate matches, the more the execution time gap between the RAM method and the RWM methods. An interesting case is $|V_Q|$=5 for the clique queries. Actually there are very few (less than 10) cliques of size 5 of a particular type in the graph. Hence, we can see that almost all the approaches take almost the same time. In this case, the top-K computation overheads associated with the RWM approaches and lack of pruning result in relatively lower execution time for RAM.

Next, note that RWM4 usually performs faster than RWM1. The time savings are higher for the path queries compared to the subgraph or clique queries. This is expected because the upper bound scores computed in RWM4 are tighter only if most of the query structure can be covered by the non-overlapping paths. Also, RWM0 performs slightly better than RWM4 for smaller query sizes, but candidate node filtering helps significantly as query size increases.

Table IV shows the time split between the candidate filtering step and the actual top-K execution. Note that the candidate filtering takes a very small fraction of the total query execution time.

	$\|V_Q\|$=2	$\|V_Q\|$=3	$\|V_Q\|$=4	$\|V_Q\|$=5
RAM	158	3186	39294	469962
RWM0	10	165	824	4660
RWM1	12	195	1022	5891
RWM2	12	212	3135	27363
RWM3	111	1486	3978	9972
RWM4	12	165	791	4518

TABLE III
QUERY EXECUTION TIME (MSEC) FOR SUBGRAPH QUERIES (GRAPH $G2$ AND INDEXES WITH D=2)

QuerySize → / QueryType ↓	$\|V_Q\|$=2		$\|V_Q\|$=3		$\|V_Q\|$=4		$\|V_Q\|$=5	
	CFT	TET	CFT	TET	CFT	TET	CFT	TET
Path	8	10	10	24	10	32	12	106
Clique	5	6	8	338	9	13538	9	199608
Subgraph	6	6	9	156	10	781	12	4506

TABLE IV
RUNNING TIME (MSEC) SPLIT BETWEEN CANDIDATE FILTERING (CFT) AND TOP-K EXECUTION (TET) FOR GRAPH $G2$ (D=2)

	$\|V_Q\|$=2	$\|V_Q\|$=3	$\|V_Q\|$=4	$\|V_Q\|$=5	$\|V_Q\|$=6	$\|V_Q\|$=7
$\|V\| = 10^3$	5	18	77	382	1870	7656
$\|V\| = 10^4$	10	90	407	2267	12366	87657
$\|V\| = 10^5$	52	396	2794	18412	131256	1006773
$\|V\| = 10^6$	362	4907	28600	184523	1216893	9786327

TABLE V
RUNNING TIME (MSEC) FOR DIFFERENT QUERY SIZES AND GRAPH SIZES (D=2)

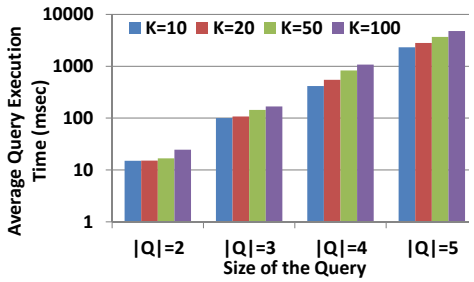

Fig. 12. Query Execution Time for Different Values of K

Scalability Results

We run the 20 path and general subgraph queries (each 10 times) over all the 4 synthetic graphs using RWM4 and present the results in Table V. The table shows that the execution time increases linearly with the graph size, and exponentially with the query size. Even though the execution time is exponential in query size, (1) that is the case with most subgraph matching algorithms, and (2) intuitive user queries are limited in size by limits of human interpretability for most applications.

Effect of Varying the K

Figure 12 shows the effect of varying K on 20 path and general subgraph queries on graph $G2$ using RWM4. As expected, the query execution time increases as K increases. However, the increase in execution time is reasonably small enough making the system usable even for larger values of K.

Pruning Power of Top-K

The time efficiency of the RWM algorithm is mainly attributed to the way it leverages the top-K framework. The algorithm starts with the size-1 candidates which are grown one edge at a time till they grow up to $|E_Q|$ or get pruned.

	$\|V_Q\|$=2	$\|V_Q\|$=3	$\|V_Q\|$=4	$\|V_Q\|$=5
#Size-1 Candidates	9.54	7.86	4.38	1.63
#Size-2 Candidates		28.28	18.31	7.94
#Size-3 Candidates			24.42	25.5
#Size-4 Candidates				13.61

TABLE VI
NUMBER OF CANDIDATES AS PERCENTAGE OF TOTAL MATCHES FOR DIFFERENT QUERY SIZES AND CANDIDATE SIZES

	DBLP	Wikipedia
Number of Nodes	138K	670K
Number of Edges	1.6M	4.1M
Number of Types	3	10
Sorted Edge List Index Size	50 MB	261 MB
Topology Index Size	5.8 MB	148 MB
MMW Index Size	11.4 MB	249 MB
SPath Index Size	4.3 GB	13.7 GB
Sorted Edge List Construction Time	12 sec	23 sec
Topology+MMW Construction Time	461 min	1094 min
Average Query Time	100 sec	42 sec

TABLE VII
DATASET AND INDEX DETAILS

Table VI shows the percentage of candidates of different sizes with respect to the total number of matches. The results shown in this table are obtained by running the algorithm for the 20 path and subgraph queries on graph $G2$. We removed the clique queries because the number of cliques of size 5 matching such queries is less than 10 and hence no pruning occurs. Note that on an average, the number of candidates is around 14% of the total number of matches. Clearly, for subgraph queries there are candidates of higher sizes also, but the number of such candidates is much smaller ($< 1\%$) compared to the number of matches, and so we do not show them here.

B. Real Datasets

We experiment with two real datasets: DBLP and Wikipedia, and obtain some interesting results.

DBLP Dataset

The DBLP network consists of authors (A), keywords (K) and conferences (C). We considered a temporal subset of DBLP[1] for 2001-2010. We obtained a list of conferences from the Wikipedia Computer Science Conferences page[2] which categorizes conferences into 14 research areas (or communities). By associating keywords from these conferences with the research areas, we obtained the keyword priors which were used as input for *NetClus* [17] to perform community detection on DBLP. The interestingness of an edge is then defined as the KL-divergence between the community distributions of its end points.

Results on the DBLP Dataset

Details of the dataset and the index are shown in Table VII. Note that, compared to the topology and the MMW index, the SPath index for RAM actually takes 4.3GB space. The number of edges of different types are as follows: AA – 288K, AC – 608K, AK – 392K, CK – 211K, KK – 87K. On an

[1] http://www.informatik.uni-trier.de/~ley/db/
[2] http://en.wikipedia.org/wiki/List_of_computer_science_conferences

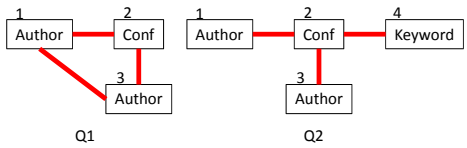

Fig. 13. Two Queries for the DBLP Dataset

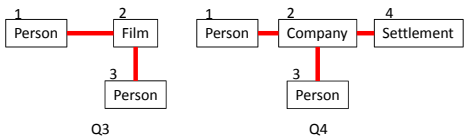

Fig. 14. Two Queries for the Wikipedia Dataset

average, query execution time is 100 seconds on the DBLP network using D=2. We present two case studies for this dataset corresponding to the two queries shown in Figure 13.

Case Study 1 For the query $Q1$ shown in Figure 13, the top subgraph turns out to be (1: Rohit Gupta, 2: BICoB, 3: Vipin Kumar). The three entities were linked because of the paper "Rohit Gupta, Smita Agrawal, Navneet Rao, Ze Tian, Rui Kuang, Vipin Kumar: Integrative Biomarker Discovery for Breast Cancer Metastasis from Gene Expression and Protein Interaction Data Using Error-tolerant Pattern Mining" at BICoB 2010. This case is interesting mainly because it represents an interesting collaboration of people from multiple areas. Rohit Gupta primarily works in computer networking. Vipin Kumar is known for his work in Data and Information Systems. BICoB (International Conference on Bioinformatics and Computational Biology) is a conference focused on bioinformatics.

Case Study 2 For the query $Q2$ shown in Figure 13, the top subgraph turns out to be (1: Jimeng Sun, 2: Operating Systems Review (SIGOPS), 3: Christos Faloutsos, 4: mining). The four entities are linked because of the paper "Evan Hoke, Jimeng Sun, John D. Strunk, Gregory R. Ganger, Christos Faloutsos: InteMon: continuous mining of sensor data in large-scale self-infrastructures." at Operating Systems Review (SIGOPS) in 2006. Again, this case represents an interesting collaboration of people from multiple areas. Jimeng Sun and Christos Faloutsos are mainly focused on Data and Information Systems. Also, "mining" is a keyword which is frequently associated with the Data and Information Systems community. On the other hand, "Operating Systems Review (SIGOPS)" can be considered to belong to the areas of Operating systems, and Computer architecture which are completely different from the research areas associated with the other entities.

Wikipedia Dataset

We generate a network using Wikipedia Infobox pages as follows. For an entity e, an edge was created from e to the entity e' in the entity relationship network, if the entity e' appears in the Wikipedia page for e and the Wikipedia pages for both the entities have Infoboxes. We restrict our study to the entities of top ten types (film, person, company, football biography, nrhp, television, album, settlement, musical artist, single). This ten-type network covers about 45% of the Wikipedia Infobox entities (1.7 million). We use the Wikipedia Infobox data as entity attributes. On an average, each entity has 28 attributes. We augment the original network of entities with the categorical and the sets-of-strings attribute values of the entities as nodes. Entity nodes are linked to each of their attribute nodes. Attribute nodes within the same set of strings are linked to each other. We use METIS [12] to compute the hard partitions (K=20) on such an augmented network. Further, we aggregate the cluster labels of all the attribute neighbors of an entity to get its soft cluster distribution. The interestingness of an edge is then defined as the KL-divergence between the community distributions of its end points.

Results on the Wikipedia Dataset

Details of the dataset and the index are shown in Table VII. Note that, compared to the topology and the MMW index, the SPath index for RAM actually takes 13.7GB space. On an average, query execution time is 42 seconds on the Wikipedia network using D=2. Again, we present two case studies for this dataset corresponding to the two queries shown in Figure 14.

Case Study 1 For the query $Q3$ shown in Figure 14, the top subgraph turns out to be (1: Stacy Keach, 2: The Biggest Battle, 3: John Huston). The Biggest Battle is an Italian Macaroni war movie (1978) in which Stacy Keach and John Huston starred. There are multiple ways in which these connections are unusual. Stacy Keach is an American actor and narrator. Usually, American actors used to act in American movies in those years. Also, Stacy has has done narration work in educational programming on PBS and the Discovery Channel, as well as some comedy and musical roles, which are quite different from this war movie. Again, John has worked mostly in US movies, rather than Italian ones. Also, John was an American film director, screenwriter and actor who worked mostly for drama, documentary, adventure and comedy movies, and not war movies.

Case Study 2 For the query $Q4$ shown in Figure 14, the top subgraph turns out to be (1: Medha Patkar, 2: BBC, 3: Felix D'Alviella, 4: Mogilino). The British Broadcasting Corporation (BBC) is a British public service broadcasting corporation. Medha Patkar is an Indian social activist and is linked to BBC because she won the Best International Political Campaigner by BBC. Felix D'Alviella is a Belgian actor best known for his character Rico Da Silva in the BBC soap opera Doctors. Mogilino is a village in Bulgaria. In 2007, the BBC showed the film "Bulgaria's Abandoned Children" which became quite popular. This combination of entities is surprising in multiple ways. It is rare for a British company to reward an Indian woman. Similarly, it is rare for a British company to be linked to a place in Bulgaria or a person from Belgium. Thus, each of these links to BBC are quite rare causing the entire combination to be reported as the top interesting subgraph.

VII. Related Work

The network (graph) query problem can be formulated as a selection operator on graph databases and has been studied first in the theory literature as the subgraph isomorphism problem [3], [14], [20]. One way of answering network queries

is to store the underlying graph structure in relational tables and then use join operations. However, joins are expensive, and so fast algorithms have been proposed for approximate graph matching as well as for exact graph matching. A problem related to the proposed problem is: given a subgraph query, find graphs from a graph database which contain the subgraph [16], [22], [29]. All top-K processing algorithms are based on the Fagin et al.'s classic TA algorithm [4]. Growing a candidate solution edge-by-edge in a network can be considered to be similar to performing a join in relational databases. The candidates are thus grown one edge at a time much like the processing of a top-K join query [11] and as detailed in Section IV-B. However, we make the top-K join processing faster by tighter upper bounds computed using the MMW index and list pruning using the topology index. The top-K joins on networks with the support of such graph indexes is our novel contribution. The proposed problem is also related to the team selection literature. However, most of such literature following the work of Lappas et al. [13] focuses on clique (or set) queries [10], unlike the general subgraph queries handled by the proposed approach. Top-K matching subgraphs can also be considered as statistical outliers. Compared to our previous work on outlier detection from network data [6], [7], [8], [9], we focus on query based outlier detection in this work. For more comparisons with previous work, please refer to Section I.

VIII. Conclusion

In this paper, we studied the problem of finding top-K interesting subgraphs corresponding to a typed unweighted query aplied on a heterogeneous edge-weighted information network. The problem has many practical applications. The baseline ranking after matching solution is very inefficient for large graphs where the number of matches is humongous. We proposed a solution consisting of an offline index construction phase and an online query processing phase. The low cost indexes built in the offline phase capture the topology and the upper bound on the interestingness of the metapaths in the network. Further, we proposed efficient top-K heuristics that exploit these indexes for answering subgraph queries very efficiently in an online manner. Besides showing the efficiency and scalability of the proposed approach on synthetic datasets, we also showed interesting subgraphs discovered from real datasets like Wikipedia and DBLP. In the future, we plan to study this problem in a temporal setting.

IX. Acknowledgements

The work was supported in part by the U.S. Army Research Laboratory under Cooperative Agreement No. W911NF-11-2-0086 (Cyber-Security) and W911NF-09-2-0053 (NS-CTA), the U.S. Army Research Office under Cooperative Agreement No. W911NF-13-1-0193, and U.S. National Science Foundation grants CNS-0931975, IIS-1017362, and IIS-1320617.

References

[1] P. Bogdanov, M. Mongiovì, and A. K. Singh. Mining Heavy Subgraphs in Time-Evolving Networks. In *ICDM*, pages 81–90, 2011.

[2] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, pages 442–446, 2004.

[3] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *TPAMI*, 26(10):1367–1372, 2004.

[4] R. Fagin, R. Kumar, and D. Sivakumar. Comparing Top-K Lists. In *SODA*, pages 28–36, 2003.

[5] G. Gou and R. Chirkova. Efficient Algorithms for Exact Ranked Twig-pattern Matching over Graphs. In *SIGMOD*, pages 581–594, 2008.

[6] M. Gupta, J. Gao, C. C. Aggarwal, and J. Han. Outlier Detection for Temporal Data. In *SDM*, 2013.

[7] M. Gupta, J. Gao, and J. Han. Community Distribution Outlier Detection in Heterogeneous Information Networks. In *ECML PKDD*, pages 557–573, 2013.

[8] M. Gupta, J. Gao, Y. Sun, and J. Han. Community Trend Outlier Detection using Soft Temporal Pattern Mining. In *ECML PKDD*, pages 692–708, 2012.

[9] M. Gupta, J. Gao, Y. Sun, and J. Han. Integrating Community Matching and Outlier Detection for Mining Evolutionary Community Outliers. In *KDD*, pages 859–867, 2012.

[10] M. Gupta, J. Gao, X. Yan, H. Cam, and J. Han. On Detecting Association-Based Clique Outliers in Heterogeneous Information Networks. In *ASONAM*, 2013.

[11] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting Top-K Join Queries in Relational Databases. *VLDB Journal*, 13(3):207–221, Sep 2004.

[12] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *J. Sci. Comp*, 20(1):359–392, Dec 1998.

[13] T. Lappas, K. Liu, and E. Terzi. Finding a Team of Experts in Social Networks. In *KDD*, pages 467–476, 2009.

[14] B. D. McKay. Practical Graph Isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[15] Y. Qi, K. S. Candan, and M. L. Sapino. Sum-Max Monotonic Ranked Joins for Evaluating Top-K Twig Queries on Weighted Data Graphs. In *VLDB*, pages 507–518, 2007.

[16] S. Ranu and A. K. Singh. GraphSig: A Scalable Approach to Mining Significant Subgraphs in Large Graph Databases. In *ICDE*, pages 844–855, 2009.

[17] Y. Sun, Y. Yu, and J. Han. Ranking-based Clustering of Heterogeneous Information Networks with Star Network Schema. In *KDD*, pages 797–806, 2009.

[18] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient Subgraph Matching on Billion Node Graphs. *PVLDB*, 5(9):788–799, May 2012.

[19] Y. Tian, R. C. Mceachin, C. Santos, D. J. States, and J. M. Patel. SAGA: A Subgraph Matching Tool for Biological Graphs. *Bioinformatics*, 23(2):232–239, Jan 2007.

[20] J. R. Ullmann. An Algorithm for Subgraph Isomorphism. *J. ACM*, 23(1):31–42, Jan 1976.

[21] X. Yan, B. He, F. Zhu, and J. Han. Top-K Aggregation Queries over Large Networks. In *ICDE*, pages 377–380, 2010.

[22] X. Yan, P. S. Yu, and J. Han. Substructure Similarity Search in Graph Databases. In *SIGMOD*, pages 766–777, 2005.

[23] J. Yang, W. Su, S. Li, and M. M. Dalkiliç. WIGM: Discovery of Subgraph Patterns in a Large Weighted Graph. In *SDM*, pages 1083–1094, 2012.

[24] Y. Yuan, G. Wang, L. Chen, and H. Wang. Efficient Subgraph Similarity Search on Large Probabilistic Graph Databases. *PVLDB*, 5(9):800–811, May 2012.

[25] X. Zeng, J. Cheng, J. X. Yu, and S. Feng. Top-K Graph Pattern Matching: A Twig Query Approach. In *WAIM*, pages 284–295, 2012.

[26] S. Zhang, J. Yang, and W. Jin. Sapper: Subgraph indexing and approximate matching in large graphs. *PVLDB*, 3(1):1185–1194, 2010.

[27] P. Zhao and J. Han. On Graph Query Optimization in Large Networks. *PVLDB*, 3(1):340–351, 2010.

[28] F. Zhu, Q. Qu, D. Lo, X. Yan, J. Han, and P. S. Yu. Mining Top-K Large Structural Patterns in a Massive Network. *PVLDB*, 4(11):807–818, 2011.

[29] Y. Zhu, L. Qin, J. X. Yu, and H. Cheng. Finding Top-K Similar Graphs in Graph Databases. In *EDBT*, pages 456–467, 2012.

[30] L. Zou, L. Chen, and Y. Lu. Top-K Subgraph Matching Query in a Large Graph. In *PIKM*, pages 139–146, 2007.

[31] L. Zou, L. Chen, and M. T. Özsu. Distance-join: Pattern Match Query in a Large Graph Database. *PVLDB*, 2(1):886–897, Aug 2009.