2012 IEEE 28th International Conference on Data Engineering

Ego-centric Graph Pattern Census

Walaa Eldin Moustafa, Amol Deshpande, Lise Getoor

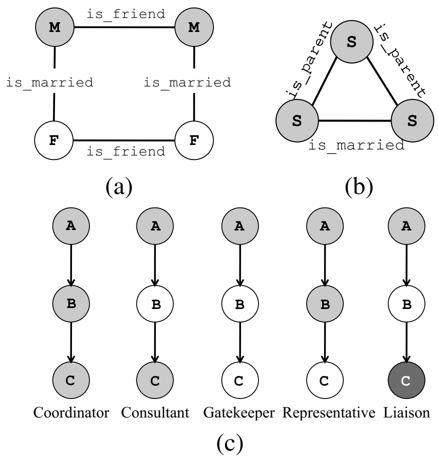
Department of Computer Science, University of Maryland
College Park, USA
{walaa,amol,getoor}@cs.umd.edu

***Abstract*—There is increasing interest in analyzing networks of all types including social, biological, sensor, computer, and transportation networks. Broadly speaking, we may be interested in global network-wide analysis (e.g., centrality analysis, community detection) where the properties of the entire network are of interest, or local *ego-centric* analysis where the focus is on studying the properties of nodes (egos) by analyzing their neighborhood subgraphs. In this paper we propose and study *ego-centric pattern census queries*, a new type of graph analysis query, where a given structural pattern is searched for in every node's neighborhood and the counts are reported or used in further analysis. This kind of analysis is useful in many domains in social network analysis including opinion leader identification, node classification, link prediction, and role identification. We propose an SQL-based declarative language to support this class of queries, and develop a series of efficient query evaluation algorithms for it. We evaluate our algorithms on a variety of synthetically generated graphs. We also show an application of our language in a real-world scenario for predicting future collaborations from DBLP data.**

I. Introduction

Network analysis is an area of growing importance in a variety of domains including the social sciences, biology, communications, ecology, finance, the internet, law enforcement, national security, social media and others. Most of the work in network analysis focuses on either (a) global macro-level analysis characterizing properties of the entire network and communities (subgraphs) within the network, or (b) local micro-level analysis characterizing properties of the actors (nodes) in the network. Macro-level analysis is important for characterizing networks and understanding the evolution of networks; examples of such analysis include measuring structural properties like degree distribution, diameter, and graph cohesion [30], and discovery of patterns or *motifs* in the network [28], [27], [6], [7], [21], [38]. Micro-level analysis focuses instead on measuring properties of the actors in network, e.g., degree centrality, second-order degree, local clustering coefficient etc. This is often called *ego network* analysis, because it looks at the individual actors (egos) and their neighbors (called *alters*) in the network. Although global network analysis is well-understood and efficient computational tools are well-developed, similar tools are not yet available for the harder problem of ego-centric network analysis that requires analyzing a very large number of small, largely overlapping networks.

In this paper, we address one such problem in ego-centric analysis, namely counting the number of structural patterns or motifs that occur either in the k-hop neighborhoods of the nodes (for a given k), or in subgraphs defined by intersections or unions of k-hop neighborhoods of pairs of nodes; k-hop neighborhood of a node n is defined to be the incident subgraph on the nodes reachable from n in k hops or less. We refer to this type of query as an *ego-centric pattern census*. We illustrate the need for such analysis using a few motivating applications:



(a)

(b)

(c)

Fig. 1. (a) Pattern that captures two couples that are friends with each other – such a pattern may be useful in a targeted marketing application; (b) Example pattern used in the node classification application; (c) Different brokerage patterns – the colors denote organizations, and the function of the broker (the middle node) depends on the organizations that the three nodes belong to (e.g., B is a coordinator if all three are in the same organization).

Targeted Marketing: Viral marketing is proving to be an effective tool for product advertisement. Selected consumers are given a product with the hope that they will like the product and recommend it to their friends. These consumers must be chosen wisely to minimize the cost and maximize the benefits of advertising. Simple criteria such as picking consumers with the most friends, or consumers that are connected to many other consumers through short paths, are typically used. However the ability to identify richer structures is desirable in many cases. For example, a travel agency may wish to identify couples that have either the largest number of couples in their combined network, or the largest number of couple pairs, i.e., couples who are friends with couples. The latter structure is depicted in Figure 1(a).

Node Classification: In collective classification [32], a node's neighborhood is used to predict the node's own class label. For example, in a collaboration network, a scientist who

1084-4627/12 $26.00 © 2012 IEEE
DOI 10.1109/ICDE.2012.113

collaborates mostly with scientists from a specific field (e.g., databases or software engineering) is likely to be from the same field. In a family relationship network (with "is married to" and "is parent of" relationships), for each child we may wish to count the number of relatives up to 3 hops away who are smokers (or obese), with their parents also being smokers (or obese). This could be a measure of the risk of being a smoker (or obese) for the child, and can be used for predicting that risk. The pattern of this query is depicted in Figure 1(b).

Structural Balance: In social balance theory, signed networks are networks that have positive and negative signs on their links, denoting whether a link expresses a positive tie (e.g., friendship) or a negative tie (e.g., foe) [10]. In signed networks, triangles with an odd number of negative links (one or three) are considered unstable. In such networks, we can measure the amount of instability in each node's ego network by counting the number of unstable triangles in its k-hop neighborhood.

Brokerage Analysis: In organizational theory, management scientists are interested in the roles different individuals play, both within an organization, and across organizations. For example, in a transaction network, the middle node in a directed triad (e.g., node B in the triad $A \rightarrow B \rightarrow C$) is called a *coordinator* if all three are in the same organization, or a *gatekeeper* if A is in a different organization than B and C. Figure 1(c) shows the different brokerage types. A brokerage score of a given type can then be computed for a node by counting the number of patterns of that type in which the node occurs as the middle node.

Graph Indexing: Finally, counts of specific structural patterns in every node's k-hop neighborhood in a large graph are regarded as *node signatures* and are often used for subgraph pattern matching to prune the search space [43]. Our algorithms can be used to efficiently build sophisticated signatures, that can be used when searching for large, complex subgraphs.

The problem of ego-centric pattern census combines elements from micro-level ego-centric network analysis, subgraph pattern matching, and motif counting. The goal of subgraph pattern matching is to find all matches of a given query graph in the database graph. While the general problem of pattern matching is NP-complete, there is much work on designing efficient algorithms and index data structures to answer those queries [20], [43], [44], [37]. Motif counting, on the other hand, is the problem of counting specified structural patterns of small sizes in the network [28], [27], [4], [6], [7], [21], [38]. Motif discovery typically does not take node or edge attributes into account, but rather depends solely on the pattern structure. It is commonly performed on naturally occurring graphs like biological networks [28], or computational graphs such as software graphs [38] or computer network graphs [3]. In these works, motif profiles (i.e., counts of different motif structures in the graph) have proven to be a strong indicator of a network's function. Local motif counting (i.e., counting motifs that a node is part of) has also been recently used as a tool to classify a node's role. For example, Kerrebroeck et al. [39] count the number of loops a node is part of, and use it as a measure to quantify the node's importance in the network. Prüzlj [31] proposes the notion of *local graphlet degree distribution* as a means for network comparison.

Several ego-centric measures can be expressed as ego-centric pattern census queries with very simple input patterns, and can be seen as special cases of ego-centric pattern census. For example, an ego-centric measure like node degree corresponds to searching for a pattern of a node in a 1-hop neighborhood, and clustering coefficient (and its k-clustering coefficient version [22]) can be expressed in terms of counting edge patterns in 1-neighborhood (and k-neighborhood, respectively). Similarly, Jaccard coefficient of a pair of nodes can be computed by counting a node pattern in the nodes' 1-neighborhood intersection and union.

Our approach and contributions: In this work we investigate ego-centric pattern census queries, and develop efficient techniques for executing them. Our key contributions include:

- We introduce a flexible declarative SQL-based query language for specifying ego-centric pattern census queries. Our query language allows users to specify the **neighborhood** size (k), the set of **focal nodes** (or pairs of nodes), and the pattern to be counted.
- We propose an efficient graph pattern matching algorithm, and show that it outperforms GraphQL [20], a recent graph pattern matching system.
- We introduce two query evaluation algorithms for the ego-centric census queries, one based on searching from nodes to patterns (*node-driven*) and another based on searching from patterns to nodes (*pattern-driven*).
- We empirically evaluate our algorithms on a variety of real-world and synthetic data.

Outline: The rest of paper is organized as follows. In Section II we present the data model and our language specification. In Section III, we propose an efficient pattern matching algorithm that is used as part of the algorithms in Section IV that we propose for evaluating pattern census queries. In Section V we discuss experimental performance evaluation through synthetic datasets and workloads. In addition, we use our language to design a link prediction experiment over DBLP and report its results. We discuss related work in detail in Section VI.

II. Data Model and Language Specification

We begin with a brief discussion of the graph data model, and some basic definitions. We denote the database graph by $G = (V_G, E_G)$, where V_G and E_G denote the sets of nodes and edges respectively. The database graph may be directed or undirected, and both the nodes and the edges can have arbitrary sets of attributes (stored as *attribute-value* pairs). Similarly we denote the pattern graph by P. A pattern graph may be associated with a set of predicates on the underlying attributes.

A pattern graph $P = (V_P, E_P)$ is said to be isomorphic to a graph $M = (V_M, E_M)$ if there exists a bijective mapping

TABLE I
EXAMPLES OF PATTERNS AND PATTERN CENSUS QUERIES

Row #	Pattern	Query
1	**PATTERN** single_node {?A;}	**SELECT** ID, **COUNTP**(single_node, **SUBGRAPH**(ID, 2)) **FROM** nodes

2	**PATTERN** single_edge {?A-?B;}	**SELECT** n1.ID, n2.ID,
		COUNTP(single_edge, **SUBGRAPH-INTERSECTION**(n1.ID, n2.ID, 1))
		FROM nodes **AS** n1, nodes **AS** n2

3	**PATTERN** square {
	?A-?B; ?B-?C;
	?C-?D; ?D-?A;
	}
		SELECT ID, **COUNTP**(square, **SUBGRAPH**(ID, 2)) **FROM** nodes

4	**PATTERN** triad {
	?A->?B; ?B->?C; ?A!->?C;
	[?A.LABEL=?B.LABEL];
	[?B.LABEL=?C.LABEL];
	SUBPATTERN coordinator {?B;}
	}
		SELECT ID, **COUNTSP**(coordinator, triad, **SUBGRAPH**(ID, 0)) **FROM** nodes

$\mu : V_P \rightarrow V_M$ such that $(v, v') \in E_P$ if and only if $(\mu(v), \mu(v')) \in E_M$, and the predicates in P are satisfied under μ. We say that a subgraph M of a graph G is a *match* for a pattern graph P if M is isomorphic to P.

Next, we introduce our language for specifying pattern census queries. Our pattern specification language is designed to be general and flexible. The language is based on SQL, but our algorithms actually operate on an disk-resident adjacency-list graph representation, and our system can be easily extended to support a visual pattern specification language as well.

The pattern census SQL queries are written against a logical representation of the graph as two relations: nodes(ID, NATTR1, ...) and edges(ID1, ID2, EATTR1, ...), where ID is the node identifier. Attribute references in queries are interpreted dynamically, and hence the list of attributes does not have to be pre-specified.

For a pattern census query, we need to be able to specify three things:

Search Neighborhoods: We need to specify the neighborhoods in which to do pattern census. We currently support specifying three types of search neighborhoods:

- **SUBGRAPH**(N, k): This specifies a k-hop neighborhood around the node, i.e., the incident subgraph on the nodes that are reachable from N in k hops or less.
- **SUBGRAPH-INTERSECTION**(N1, N2, k): Given two nodes $N1$ and $N2$ and a radius k, this specifies the intersection of the k-hop neighborhoods of $N1$ and $N2$.
- **SUBGRAPH-UNION**(N1, N2, k): Similar to above except we take union instead of intersection.

Focal Nodes: We need to be able to specify for which nodes or for which pairs of nodes to conduct the pattern census. We use standard SQL constructs for this purpose, i.e., the user can specify predicates that should be satisfied by the nodes or pairs of nodes. Predicates are given in the **WHERE** clause of the SQL statement.

Pattern: Our pattern specification language (see Table I) allows the user to specify the pattern name (e.g., single_node, square, triad), the nodes in the pattern (e.g., ?A, ?B, ?C), the connections (edges) between the nodes (e.g., ?A-?B), and predicates on either the node attributes or the edge attributes (e.g., ?A.LABEL=?B.LABEL). The structural pattern (nodes and edges) is specified using variables (i.e., A, B, C) that can be bound to any node in the graph. The user can also specify the direction of each edge if desired (e.g., ?A->?B), and can specify that a particular edge should not exist[1]. Table I (1-3) shows three simple patterns and SQL queries that count the number of patterns in different types of neighborhoods. Table I (4) shows a somewhat more complex directed pattern, that also specifies that a particular edge (from ?A to ?C) should not exist and requires all three nodes to have the same label (this pattern corresponds to a *coordinator* in brokerage analysis).

We also allow the user to specify one or more **subpatterns** in the pattern, where each subpattern is specified as a subset of the nodes in the pattern. This allows the user to precisely dictate the types of matches that should be counted. Consider the example shown in Table I (4). Here we specify a single subpattern containing the middle node in the triad, and the census is done in the 0-hop neighborhood around each node (which contains just that node). In other words, this query counts the number of triads in which ?B is the coordinator. It is not possible to do this type of census without the subpattern construct (if we simply count the number of triads in the 1-hop neighborhood around each node, we would also count the triads for which B is not a coordinator).

Finally, the above constructs are put together using two user-defined SQL aggregate functions to fully specify the query to be executed: (1) **COUNTP**(p, S), where p is the pattern name, and S is the neighborhood specified using one of the subgraph functions, and (2) **COUNTSP**(sp, p, S), where *sp* specifies a subpattern to be counted instead.

III. SUBGRAPH PATTERN MATCHING

A key component of both of our proposed query evaluation algorithms is a pattern matching algorithm that is used to find all matches for the given pattern in the graph. We

[1] In our prototype implementation, we currently optimize queries with selection predicates on the form ?A.LABEL=constant. Negation and join predicates can be incorporated as a final filtering step that filters out the tuples violating those conditions.

TABLE II
NOTATION USED IN THE PAPER

Notation	Explanation
$G = (V_G, E_G)$	Database graph
n, n'	Database graph nodes
$P = (V_P, E_P)$	Pattern graph
v, v'	Pattern graph nodes
$\mathcal{M}$	Set of matches of P in G
$V_\sigma(G)$	Focal nodes, i.e., result of the SQL node selection predicates
M	A pattern match (i.e., a subgraph of G isomorphic to P)
m, m'	Nodes in a pattern match M
$N(x)$	Immediate neighbors of node x
$N^l(x)$	Neighbors of node x with label l
$N_k(x)$	Neighbors of node x in radius k
$S(n, k)$	k-hop neighborhood subgraph of node n
$\mu(v, M)$	Image of v in a match M. M is not stated when it is clear from the context.
$\mu^{-1}(m, M)$	The node in P which m matches. M is not stated when it is clear from the context.

adapt the algorithm proposed recently by He and Singh [20] (denoted GQL henceforth), by incorporating additional novel pruning steps that lead to orders-of-magnitude performance improvements over that prior work. Our algorithm consists of four steps: (1) enumerate candidate matches for each pattern node, (2) initialize candidate neighbor sets for each candidate node, (3) simultaneously prune candidate nodes and their candidate neighbors, and (4) extract pattern matches directly from the pruned set of candidates and the candidate neighbors. Although similar in spirit to GQL, our algorithm differs from it in subtle but significant ways. Our algorithm is centered around the idea of explicitly maintaining *candidate neighbors* with each candidate node. This not only results in more efficient pruning of the search space, but also results in orders of magnitude improvements in the final stage of extracting patterns. Our algorithm is also much simpler. In the description that follows, we assume both the pattern and database graphs have an explicit attribute called *label* drawn from a finite label space; the unlabeled case is equivalent to both the database and pattern graphs having the same label for all nodes. Our algorithms are applicable to both directed and undirected graphs; however we focus on undirected graphs here for simplicity. Table II lists the notation used in the following discussion.

A. Enumerating candidates of each pattern node

The first step of this algorithm is to enumerate the candidate database graph nodes for each pattern node. We utilize *node profiles* [20], [44] for this purpose. A node profile is a compact representation of a node's neighborhood that contains the number of neighbors for each label. Let $\mathcal{L} = \{l_1, l_2, \ldots, l_L\}$ denote the L vertex labels. Then, the profile $P(n)$ of a node n is the vector: $\langle |N^{l_1}(n)|, |N^{l_2}(n)|, \ldots, |N^{l_L}(n)| \rangle$, where $N^{l_i}(n)$ denotes the set of neighbors of n having label l_i. A database graph node n is a candidate for a pattern graph node v if and only if $P(v)$ is contained in $P(n)$, i.e., for each label $l_i \in \mathcal{L}$ in $N(v)$, $|N^{l_i}(n)| \geq |N^{l_i}(v)|$. To make this filtering process fast, each database node profile is calculated once and stored along with the graph as an index. The result of this step is a set of database node candidates $C(v)$ for each pattern node $v \in V_P$.

B. Initializing the candidate neighbor sets

Let v be a pattern node and v' be one of its neighbors in the pattern graph. For each node $n \in C(v)$ that is a candidate for v, we maintain a set of candidate neighbors with respect to v', denoted by $CN(n, v, v')$, i.e., neighbors of n that are a possible match to v'. We initialize each such set by finding the neighbors of n that have the same label as v', i.e., $CN(n, v, v') = C(v') \cap N(n)$.

C. Simultaneously pruning the candidates and their neighbors

Consider a pattern node v and a candidate node $n \in C(v)$. For every neighbor v' of v in the pattern graph, we must have that $CN(n, v, v')$ is non-empty. We use this observation to prune the candidate sets. We make passes over the candidate sets; in each pass, we remove those nodes from the candidate sets that do not satisfy this condition, and we then prune the candidate neighbor sets by identifying nodes n' such that $n' \in CN(n, v, v')$ but $n' \notin C(v')$. It is not hard to prove that the number of iterations is bounded by the number of nodes in the pattern graph (we omit the proof because of space constraints). Our approach is much simpler to implement than the approach based on semi-perfect matchings proposed by He et al. [20], but does not prune as aggressively for some types of query patterns; however, as we show in the experimental study, overall the performance of our approach is superior to theirs.

D. Extracting the set of matches from candidate sets

The output of the previous step is the set of candidates for each pattern node, along with their candidate neighbors. To find the final set of matches, we process these sets of candidates in a forward manner. For this purpose, we first choose an order of the pattern nodes such that each prefix of the order forms a connected component. Let $v_1, \cdots, v_{|V_P|}$ be that order. At step i, we produce the set of matches for the pattern subgraph consisting of $v_1, \cdots, v_i$ (and all edges between them). In step $i + 1$, we grow the matches by adding possible matching nodes to v_{i+1}. Let $v_{j_1}, \cdots, v_{j_l}$, where $j_1 < j_2 < \cdots < j_l < i + 1$, be the pattern nodes that are connected to v_{i+1} that appear before v_{i+1} in the chosen order. Then we find the possible matches for m_{i+1} efficiently by taking an intersection of candidate neighbor sets: $CN(n_{j_1}, v_{j_1}, v_{i+1})$, $CN(n_{j_2}, v_{j_2}, v_{i+1})$, …, $CN(n_{j_l}, v_{j_l}, v_{i+1})$, and removing nodes that already appear in $n_1, \cdots, n_i$, if any. Since the candidate neighbor sets are typically small, this step can be done very efficiently (as opposed to prior work [20] where this check requires scanning over comparatively large candidate sets). If the intersection of the candidate neighbor sets is empty, then the corresponding partial match is discarded. In the experimental section, we show that utilizing candidate neighbors leads to orders of magnitude savings in finding pattern matches.

IV. Query Evaluation Algorithms

Next, we develop a suite of algorithms to solve the egocentric pattern census problem. In this section, we present algorithms for evaluating queries of type:

```
SELECT ID, COUNTP(pattern, SUBGRAPH(ID, k))
FROM NODES WHERE (PREDICATE)
```

We defer the discussion of how to handle queries involving subpatterns and pairwise intersection/union search neighborhoods to the appendix.

We investigate two broad methods for answering such queries: *node-driven*, and *pattern-driven*, that can be seen as duals of each other. In node-driven algorithms, we start from the nodes and search for pattern matches in their neighborhoods, whereas in pattern-driven algorithms, we start from the pattern matches and look for the nodes whose neighborhoods contain those pattern matches. We assume the existence of a function `pattern-match(G,P)` which returns the set of all matches of the pattern P in the graph G. Furthermore, we refer to the set of database graph nodes selected as a result of applying the node restriction predicates as $V_\sigma(G)$.

A. Node-driven algorithms

The simplest node-driven algorithm, which we use as a baseline, works by extracting the k-hop subgraph around each node $n \in V_\sigma(G)$, denoted $S(n,k)$, and then performing pattern matching on that subgraph. This baseline algorithm (called **ND-BAS**), however, suffers from repeated and overlapping computations, especially for $k \geq 2$, and is computationally infeasible in practice. Next we propose two node-driven methods: pivot indexing and differential counting.

*1) Pivot Indexing (***ND-PVOT***):* The pivot indexing algorithm starts with finding all pattern matches in the database graph, denoted by $\mathcal{M}$, and then counts the number of matches in each node's neighborhood. We use the `pattern-match` algorithm to find $\mathcal{M}$. Then, for each node $n \in V_\sigma(G)$, and for each pattern match $M \in \mathcal{M}$, we check if the nodes in M are entirely contained in $S(n,k)$. However, the naive way to do this requires $O(|V_\sigma(G)| * |\mathcal{M}| * |V_P|)$ checks, which makes this base algorithm impractical. We next introduce two optimizations to reduce the running time significantly.

Pattern Indexing: To avoid checking if every match $M \in \mathcal{M}$ is contained in $S(n,k)$ for every n, we index $\mathcal{M}$ so that the relevant subset of $\mathcal{M}$ can be retrieved when needed. For this purpose, we first designate a node v in the pattern graph as the *pivot* node, and build a pattern match index (denoted PMI_v) on $\mathcal{M}$ using the nodes corresponding to the pivot node in the matches. Let $PMI_v(n')$ denote the list of matches returned by the index for node n' (i.e., the list of pattern matches in which n' is the image of the pattern node v).

Now, to count the pattern matches in $S(n,k)$, we traverse the neighborhood of every node $n \in V_\sigma(G)$ in a breath first fashion starting with n until we reach the maximum depth k. For each node n' visited in this process, we retrieve $PMI_v(n')$ and for each match $M \in PMI_v(n')$, we check if V_M is completely contained within $N_k(n)$. Next we discuss how to efficiently reduce these containment checks further.

Avoiding Containment Checks: Let max_v denote the distance between v and the node farthest from it in the pattern graph. Let $d(n,n')$ denote the shortest distance between n and n'. Then, if $d(n,n') + max_v \leq k$, any pattern match in $M \in PMI_v(n')$ must be completely contained in $S(n,k)$.

Thus, for any node $n' \in V_G$ for which $d(n,n') \leq k - max_v$, we can avoid checking whether each $M \in PMI_v(n')$ is entirely contained in $S(n,k)$ and instead we simply add $|PMI_v(n')|$ to the overall pattern match count for n. On the other hand, if $d(n,n') + max_v > k$, we need to explicitly check whether all nodes in $PMI_v(n')$ are in $S(n,k)$. Specifically, let v' denote a node in the pattern graph such that $d(v,v') + d(n,n') > k$. Then, we must explicitly check whether the corresponding node in M is within k hops from n. However, if $d(v,v') + d(n,n') \leq k$, then this check can be avoided. Note that both $d(v,v')$ and $d(n,n')$ are easily computed (the former can be pre-computed once for the pattern graph, whereas the latter is available since we are using breadth first search).

Pivot Selection: Finally, the choice of the pivot node v becomes critical for the performance of this algorithm. However, it is easy to see that choosing the node with the minimum value of max_v is optimal with respect to the number of database nodes for which we have to do explicit checks, i.e.,

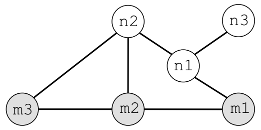
$$v = argmin_{x \in V_P}\{d(x, argmax_{y \in V_P}\{d(x,y)\})\}$$

The pseudocode is listed in Algorithm 2 in Appendix B.

*2) Differential Counting (***ND-DIFF***):* The second node-driven approach that we investigate is based on the idea of exploiting shared neighborhoods – Zhang et al. [43] use a similar idea for building a pairwise signature index in their proposed approach for subgraph pattern matching. Let $\mathcal{M}[n]$ denote the set of pattern matches contained in $S(n,k)$. Given two nodes, n and n', and $\mathcal{M}[n]$, we can construct $\mathcal{M}[n']$ by: (1) removing all matches $M \in \mathcal{M}[n]$ for which at least one node in M is present in $N_k(n) - N_k(n')$, and (2) by finding additional matches that contain nodes present in $N_k(n') - N_k(n)$ and are fully contained in $S(n',k)$.

As above, we start with finding all pattern matches $\mathcal{M}$ using the `pattern-match` algorithm. We then build a modified pattern match index that indexes $\mathcal{M}$ using all the nodes in the match (instead of just the pivot node). In other words, $PMI[n]$ contains all pattern matches that contain n. We start with an arbitrary database graph node n and compute $\mathcal{M}[n]$ using $PMI[n]$ (using a technique very similar to the above algorithm). We then pick an arbitrary neighbor n' of n and compute $\mathcal{M}[n']$ using $\mathcal{M}[n]$. The detailed algorithm is listed in Algorithm 3 in Appendix B.

Differential counting is appropriate for finding node-centric counts of compact structures such as nodes or edges, but more complex patterns will likely have parts that fall in unshared areas, making differential counting less effective in such cases. Furthermore, picking a random neighbor does not always



(a)

#	Queue	Head
1	$m_1(0,1,2), m_2(1,0,1), m_3(2,1,0)$	m_1
2	$m_2(1,0,1), m_3(2,1,0), n_1(1,2,3)$	m_2
3	$m_3(2,1,0), n_1(1,2,3), n_2(2,1,1)$	m_3
4	$n_1(1,2,3), n_2(2,1,1)$	n_1
5	$n_2(2,1,1), n_3(2,3,4)$	n_2
6	$n_3(2,3,4), n_1(1,2,2)$	n_3
7	$n_1(1,2,2)$	n_1
8	$n_3(2,3,3)$	n_3
9	ϕ	–

(b)

#	Queue	Head
1	$m_2(1,0,1), m_1(0,1,2), m_3(2,1,0)$	m_2
2	$m_1(0,1,2), m_3(2,1,0), n_2(2,1,2)$	m_1
3	$m_3(2,1,0), n_2(2,1,2), n_1(1,2,3)$	m_3
4	$n_2(1,2,2), n_1(1,2,3)$	n_2
5	$n_1(1,2,2)$	n_1
6	$n_3(2,3,3)$	n_3
7	ϕ	–

(c)

Fig. 2. (a) Example used to illustrate the advantage of best-first traversal order. (b) and (c) Simultaneous node expansions around the pattern match $\{m_1, m_2, m_3\}$ using breadth-first and best-first approaches, respectively.

guarantee that the shared neighborhood is large enough (we experimented with a heuristic based on shingle ordering [12], but the results were essentially the same and hence we do not report those here). In addition, if there is a selection predicate that specifies a subset of nodes to do pattern census for, then sharing opportunities may be rarer (especially with selective predicates). In our experimental evaluation, pivot indexing technique always outperformed differential counting.

B. Pattern-driven Algorithms

The second class of algorithms that we propose start with the pattern matches and search for nodes that contain the pattern match within their neighborhoods. These can be seen as dual to the node-driven algorithms in that, here we process each pattern match once, but may process the nodes multiple times, whereas in node-driven algorithms, we process each node once, but may process each pattern match multiple times.

The baseline pattern-driven algorithm (called **PT-BAS**) processes the pattern matches in the database graph independently one at a time. As before, let $S(n, k)$ denote n's k-hop neighborhood subgraph. For each pattern match $M = (V_M, E_M)$, for each node $m_i \in V_M$, we traverse $S(m_i, k)$ in a breadth-first fashion, and for each node in $S(m_i, k)$, we compute its distance from m_i. We then find the node $m_{min} \in V_M$ with the least number of k-hop neighbors, and for each of its k-hop neighbors, we check whether that neighbor is reachable within k hops from every other node in V_M.

Next we discuss a series of optimizations that improve upon this baseline algorithm.

1) Simultaneous Traversal: In the baseline algorithm, an edge may be traversed multiple times if it is shared among the neighborhoods of the nodes in V_M (this will often be the case). We reduce the number of such edge traversals by traversing the neighborhoods of all nodes in V_M *simultaneously*, using a breadth-first algorithm whose queue is initialized with V_M. In each step, we remove and process one node from the queue. With each visited node n, for each pattern match node $m \in V_M$, we maintain $PMD_m[n]$, the current upper bound on the distance between n and m. When a node n is visited, we update the distance vector for its neighbor n' according to the relation: $PMD_m[n'] = min(PMD_m[n]+1, PMD_m[n'])$ for each $m \in M$. If at least one of the distances is updated, then n' is pushed on the queue. The algorithm terminates when the queue is empty. Initially $PMD_m[m] = 0$ for each $m \in V_M$ and is equal to ∞ (or $k + 1$) otherwise.

2) Distance Shortcuts: We can save some initial PMD computation steps by utilizing the fact that the pattern P is isomorphic to any pattern match $M \in \mathcal{M}$. We find the distances between every pair of nodes v, v' in the pattern, and reuse these to initialize PMD for the nodes in V_M for each match M. Specifically, for $m, m' \in V_M$, we set $PMD_m[m'] = d(\mu^{-1}(m), \mu^{-1}(m'))$ if $d(\mu^{-1}(m), \mu^{-1}(m')) \leq k$, and initialize it to $k + 1$ otherwise (recall that $\mu^{-1}(m)$ denotes the pattern node $\in P$ that matches the node $m \in M$).

3) Best-first Ordering: Depending on the order in which the nodes are visited, unnecessary traversals can still occur despite the above two optimizations. Here we present a heuristic approach to further minimize the unnecessary computation by choosing which node to visit next. Specifically, we choose the node with the minimum $score(n) = \Sigma_{m \in V_M} PMD_m[n]$ in the queue to visit next. The intuition behind this heuristic is that the node with lowest $score()$ value is the node that is closest (of the remaining nodes) to all the pattern match nodes combined, and likely more influential in determining the distances from the pattern match nodes.

As an example, consider the graph in Figure 2(a). In this graph, the pattern match nodes are m_1, m_2 and m_3, and $k = 3$. Figure 2(b) shows the operation of the simultaneous breadth-first traversal approach. Initially, the traversal queue is initialized with the three M nodes, m_1, m_2, m_3, along with their PMD values for (m_1, m_2, m_2). At each step, the node n_h at the head of the queue is removed and its neighbors are inserted into the queue along with their PMD values if they do not exist, or their PMD values are updated if they already exist. In Figure 2(b), we observe that in step 4, n_1 is examined before n_2, which is examined in step 5. As a result, when n_2 is examined, it updates the PMD of n_1, causing it to be reinserted, and subsequently causing n_3 to be reinserted too. Figure 2(c) shows the operation of the algorithm by employing the best-first order. It can be seen that the reinsertions of nodes n_1 and n_3 have been eliminated, and each node is visited exactly once. The details of the algorithm are provided in Algorithm 4 in Appendix B.

Although the best-first approach reduces the number of traversals, it comes with an additional cost of having to maintain a priority queue, which requires $O(\log |Q|)$ time for insertion and deletion, where $|Q|$ is the queue size. However, in our implementation, we eliminate the cost of maintaining a heap-based priority queue by observing that the range of possible scores is pre-defined and small. Specifically, $score(n) \leq (k+1)|V_P|$ (since $PMD_m[n] \leq k+1$). Hence, we use an array-based priority queue where we store the nodes with score equal to i at position i, leading to a complexity of $O(1)$ for both insertions and deletions.

4) Center-based Expansion: Best-first ordering is aimed at reducing the number of node reinsertions into the queue; a node reinsertion may cause its neighbors to be reinserted and hence is an expensive operation. However, best-first ordering does not entirely eliminate node reinsertions. Our next optimization is based on the idea of identifying a set of *important* nodes and making sure they are not reinserted into the queue. Let $\mathcal{C} \in V_G$ denote the set of nodes (called *centers*) that are picked apriori for this purpose. We pre-compute the distances $d(c,n)\ \forall\ c \in \mathcal{C}, n \in V_G$. At query time, we insert these nodes along with their scores (computed at query time) to the traversal queue as part of the queue initialization, i.e., $PMD_m[c] = d(c,m)$ for all $c \in \mathcal{C}$ and $m \in V_M$. Now once these nodes are visited (and their neighbors processed), they will never be reinserted into the queue. Further, we can use the *triangle inequality* to get tighter upper bounds on the distances for other nodes. For any $m, n', c \in V_G$, we have that $d(m,n') \leq d(m,c) + d(c,n')$. So when we visit a node n whose neighbor n' is not yet initialized, we can set the PMD values of n' as:

$$PMD_m[n'] = min(PMD_m[n]+1, min_{c\in\mathcal{C}}(d(m,c)+d(c,n')))$$

Our final task is to choose a set of centers apriori. Many network centrality measures have been proposed in the social network analysis literature to reflect various notions of importance in social networks [40], including *page rank*, *betweenness centrality*, *closeness centrality*, to name a few. In our implementation, we pick $\mathcal{C}$ to be the set of nodes with the highest degree centrality, i.e., the nodes with the highest degrees, primarily due to its low computation cost compared to other centrality measures.

5) Pattern Match Clustering: The algorithm presented so far processes each pattern match independently. Since many pattern matches may be close together, and in fact may overlap, processing groups of them together could potentially lead to more savings. However, the trade-off here is a larger number of distance computations – for a pattern match M that is processed in isolation, we compute distances of all nodes in M to all nodes that are within k hops of at least one node in M. If we were to process multiple pattern matches together, a larger set of distances has to be computed (for every node in a pattern match, we have to compute distances to all database nodes that are within k hops of a node in any pattern match).

We use the center distance index along with the K-means clustering algorithm to group pattern matches together. For each match M, we construct a feature vector:

$$F(M) = \langle d(c_1,m_1), d(c_1,m_2), \ldots, d(c_{|\mathcal{C}|}, m_{|V_P|})\rangle$$

After computing these feature vectors for all the matches, we use the K-means clustering algorithm [26] to cluster the matches into K clusters. (We discuss the issues in choosing K in the next section.) We then process each cluster independently by simultaneously expanding around all pattern matches in the cluster.

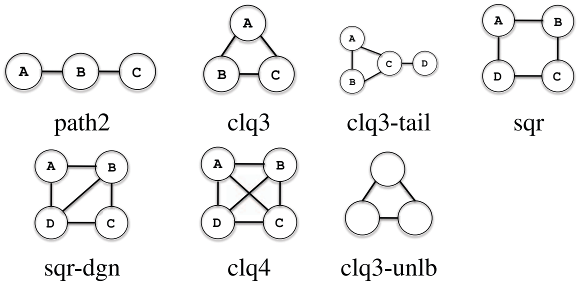


Fig. 3. Query patterns used in the synthetic dataset experiments – the letters inside the circles indicate the label of the node.

Incorporating this final optimization gives us our proposed pattern-driven algorithm (called **PT-OPT**). The details of the algorithm are listed in Algorithm 4 in Appendix B. (We omit the pattern clustering optimization for simplicity.)

V. Experimental Evaluation

In this section, we present the results of a comprehensive experimental evaluation using our prototype implementation, which is written in Java on top of the disk-based graph database engine Neo4j [2]. We begin with comparing our graph pattern matching algorithm with the prior approach by He et al. [20], and demonstrate that our approach of using candidate neighbor sets results in orders of magnitude savings. We then compare the performance of our node-driven and pattern-driven algorithms, and we study the effect of the various optimizations proposed for pattern-driven algorithms in detail. Furthermore, we discuss a real-world experiment, where we solve a link prediction problem over DBLP through our framework and report its results.

For the first set of experiments, we use synthetic database graphs generated according to the preferential attachment model [8][2]. For labeled graphs, the labels are generated randomly. The graph sizes vary from 20K nodes to 1M nodes, with the number of edges 5× the number of nodes in all graphs. The patterns used in the experiments are shown in Figure 3. All experiments were performed on identical Linux machines with 2.2 GHz quad-core processor, 8 GB of RAM, and a 750 GB 7200 RPM disk drive.

A. Experiments using synthetic datasets

We begin with comparing the performance of our pattern matching algorithm (CN) with GraphQL system (GQL) [20], also written in Java. For this purpose, we use the executable binaries that we obtained from the authors of the system.

1) Comparison with GQL for different graph sizes: Figure 4(a) shows the results (in log scale) of comparing CN with GQL for varying graph sizes (from 200K nodes/1M edges to 1M nodes/5M edges), with labels drawn randomly from a set

[2] The datasets are available at: http://www.cs.umd.edu/~walaa/datasets.html.

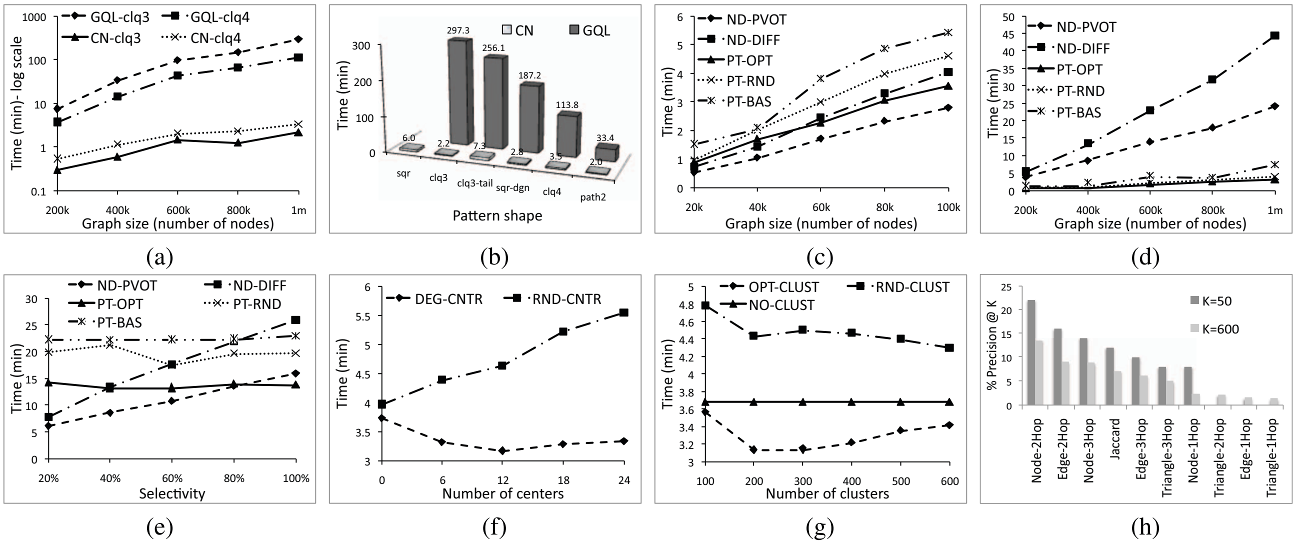


Fig. 4. (a) Comparison with GQL for different graph sizes and (b) for different patterns; (c) Pattern census: varying graph size (unlabeled graphs); (d) Pattern census: varying graph size (labeled graphs); (e) Pattern census: varying node selectivity; (f) Effect of centers on the pattern-driven algorithm; (g) Effect of clustering on the pattern-driven algorithm; (h) Precision @50 and @600 of DBLP link prediction using different structures and hop lengths.

of 4 labels, for two query patterns: clq3 and clq4 (Figure 3). As we can see, our algorithm is orders of magnitude better in almost all cases (with speedups ranging from 10 to 140). Our detailed study (omitted because of space constraints) indicates that the speedups are attributable, in large part, to the use of candidate neighbor sets.

2) Comparison with GQL for different patterns: Here we compare CN and GQL using a 1M node graph (with 5M edges), for the the labeled query patterns shown in Figure 3. The results are shown in Figure 4(b). GQL takes 37 hours for calculating matches for sqr (480 times the runtime of CN); therefore, we do not show that point on the graph. The results confirm that our algorithm outperforms GQL by orders of magnitude.

For the next three experiments (3, 4 and 5), we compare the following pattern census query evaluation algorithms:

Node-driven baseline (ND-BAS): In this algorithm, we extract $S(n, k)$ for each node and use the pattern matching algorithm to count the number of matches.

Node-driven differential counting method (ND-DIFF): This method, based on the GADDI index method in [43], traverses the nodes in the graph in some order, and computes the pattern matches for one node by utilizing the pattern matches for the prior node in the sequence.

Node-driven pivot method (ND-PVOT): Our proposed pivot indexing node-driven algorithm.

Pattern-driven baseline (PT-BAS): The baseline algorithm presented in Section IV-B.

Optimized pattern-driven algorithm (PT-OPT): The proposed pattern-driven algorithm with all the proposed optimizations. Unless otherwise is stated, the number of clusters is set to be the number of matches divided by 4, and we use 12 centers. The number of K-means iterations is 10.

Random-first pattern-driven algorithm (PT-RND): The proposed pattern-driven algorithm with all the proposed optimizations except best-first traversal. Instead, we choose the next node to process from the queue randomly.

3) Varying graph size – unlabeled graphs: Here we compare the performance of the 6 algorithms in evaluating the query (with $k = 2$):

```
SELECT ID, COUNTP(clq3-unlb, SUBGRAPH(ID, 2))
FROM nodes
```

We vary the graph size from 20K nodes to 100K nodes. The results are shown in Figure 4(c). We do not plot the running time of ND-BAS – for 20K nodes, the runtime of ND-BAS is 116 minutes, which is 218 times higher than our best performing algorithm (ND-PVOT). We see that ND-PVOT outperforms not only the other node-driven algorithms, but also the pattern-driven algorithms. This is because the query pattern (unlabeled triangle) is not very selective, i.e., the number of matches is quite high, and hence the approaches based on searching from patterns do not perform as well. We observed consistent behavior for other non-selective query patterns.

4) Varying graph size – labeled graphs: Here we use graphs with node labels randomly chosen from a set of 4 labels, and vary the graph size from 200K nodes to 1M nodes. We use a similar query as above ($k = 2$) but use a labeled triangle pattern (clq3) instead. As we can see (Figure 4(d)), PT-OPT significantly outperforms the other pattern-based algorithm, including PT-RND, illustrating the importance of the best-first order in reducing the overall runtime. Pattern-driven algorithms generally outperform node-driven algorithms because the query pattern is more selective in this case.

5) Varying focal node selectivity: Next, we vary the selectivity of the focal nodes specified in the query, controlled by the **WHERE** clause. We use an unlabeled 500K database graph. The query is:

```
SELECT ID, COUNTP(clq3-unlb, SUBGRAPH(ID, 2))
FROM nodes WHERE RND() < R
```

where we vary R from 20% to 100%. As shown in Figure 4(e), performance of pattern-driven algorithms is not affected by the focal nodes' selectivity, because those algorithms start from the pattern matches and examine their neighborhood irrespective of whether the nodes in the neighborhood are selected or not. On the other hand, running time of the node-driven methods increases linearly with the selectivity, and eventually becomes worse than pattern-driven methods.

6) Effect of the number of centers on pattern-driven algorithm: Next we examine the effect of both the number of centers and how they are chosen, using a labeled graph of 1M nodes and 5M million edges, and 4 labels. The query is:

```
SELECT ID, COUNTP(clq3, SUBGRAPH(ID, 2))
FROM nodes
```

We compare the proposed way of choosing centers, i.e., using nodes with the highest degree (DEG-CNTR) versus using randomly chosen centers (RND-CNTR). For both methods, we vary the number of centers from 0, which corresponds to not using centers, to 24 centers. Note that the number of centers affects both (1) the clustering quality and (2) distance initializations in the pattern match neighborhoods (PMD). The purpose of this experiment is to study (2) in isolation of (1) since using too few centers clearly degrades the clustering quality and the overall performance. Therefore, we isolate the effect of (1) in this experiment by fixing the number of centers that are used for clustering regardless of the number of centers used for PMD. The results are shown in Figure 4(f). We can see that using the high-degree nodes as centers greatly helps the query performance, whereas with random centers the performance worsens with increasing number of centers. On the other hand, looking at the performance of DEG-CNTR as the number of centers increases, we observe that the performance initially improves, but as the number of centers becomes too large, the overheads of using centers start dominating.

7) Effect of pattern clustering: Finally we study the effect of the pattern clustering optimization on the performance of our proposed pattern-driven algorithm using a labeled graph of 1M nodes and 5M edges, and 4 labels. The query is:

```
SELECT ID, COUNTP(clq3, SUBGRAPH(ID, 2))
FROM nodes
```

We compare the performance of three alternatives: (1) no clustering (NO-CLUST), (2) random clustering (RND-CLUST), and (3) the proposed K-means approach that is based on using the centers (OPT-CLUST). We also vary the number of clusters from 100 to 600 to show the effect of changing the number of clusters on the performance. Note that this parameter has no effect on NO-CLUST.

The results are shown in Figure 4(g). We observe that OPT-CLUST significantly outperforms both RND-CLUST and NO-CLUST, illustrating both the benefits of clustering and the need to choose the cluster carefully. Furthermore, we can see that there is a trade-off in setting the number of clusters – with too large a number of clusters (600), there is no significant advantage to using clusters since the matches are largely processed independently, but the performance also degrades with too few clusters (100). This is because in the latter case, there are too many matches in each cluster and the resulting redundant distance computations outweigh the benefits of clustering.

B. Real-world Experiment

In this experiment, we utilize our language to compare the predictive power of different structures in predicting future scientific collaborations (this is an example of a *link prediction* task). We collected publication data from SIGMOD, VLDB and ICDE conferences from 2001 to 2010. Given the co-authorship information from years 2001 to 2005, we predict collaborations in the period from 2006 to 2010. For this purpose we defined 9 pairwise measures using our language. For each pair of authors, we measure the number of nodes, edges and triangles in their common 1, 2, and 3 hop neighborhoods. In other words, we use a query of the form:

```
SELECT n1.ID, n2.ID, COUNTP
(struct, SUBGRAPH-INTERSECTION(n1.ID, n2.ID, r))
FROM nodes AS n1, nodes AS n2 WHERE n1.ID > n2.ID
```

where $struct$ represents a node, edge, or triangle pattern, and k is 1, 2 or 3, resulting in 9 total configurations. In the prediction step, for each configuration, we pick the top K pairs in terms of their common structures (i.e., the pairs of authors with the highest counts for the corresponding pattern), and then measure the precision at K defined as the number of correct predictions divided by K. Figure 4(h) shows the precision of each of the nine configurations at $K = 50$ and $K = 600$. In addition to the nine measures, the figure shows the performance of Jaccard coefficient, a similarity measure that is regarded as a good predictor and commonly used in link prediction [25]. We also measured the precision of the random predictor (which selects random K pairs of nodes) and it yielded a zero precision at both $K = 50$ and $K = 600$. For our measures, common nodes within 2 hops has the strongest prediction power, almost twice that of Jaccard coefficient. Several other measures also outperform Jaccard coefficient. With respect to runtime performance, we compared **ND-BAS**, **PT-BAS**, and **PT-OPT**. While **ND-BAS**'s performance was the poorest of all (by orders of magnitude), **PT-OPT** speedups over **PT-BAS** ranged between 0.9x for searching nodes in 1 hop (i.e. slightly slower due to optimization overhead) to 3.4x faster for searching triangles in 3 hops. This simple experiment illustrates the power of our framework in enabling social network analysis.

VI. Related Work

Our work is closely related to several active research topics that are being studied in different communities. In social network analysis, distinction is often made between socio-centric analysis and ego-centric analysis. The former has seen much work over the last two decades with focus on understanding how networks evolve (see, e.g., [30], [8]), computing and reasoning about global or local properties of the networks, designing visualization tools to help with analysis (e.g., NodeXL [36]) and so on. In ego-centric analysis, instead the focus is typically on understanding how the structure of the neighborhood around a node affects the node or dictates its function. For example, *structural holes* in ego networks are considered indicative of the positional advantage or disadvantage of individuals [9], [23]. Although computational techniques for ego-centric analysis are not as well-developed yet, there is increasing interest in understanding how to do ego-centric analysis more efficiently and several software packages support reasoning over ego networks (e.g., EgoNet [1]).

Another related research area is the study of network motifs [28], [27], [6], [7]. Roughly speaking, network motifs are subgraphs that occur more frequently than expected to appear in a random network. Most real-world networks exhibit a small set of motifs that occur repeatedly in the network, and can be considered its building blocks. There is much work on efficiently counting the number of motifs that appear in a given network [4], [5], [16]. Although similar in spirit, our focus on counting motifs (generalized to allow predicates on the node or edge attributes) in all ego networks requires us to develop new computational techniques to solve the problem.

In the area of graph databases, several query languages have been proposed to query and manage graph data including GraphLog [13], GOOD [18], GraphDB [17], GOQL [35], PQL [24], and GrDB [29]. There is also much work on subgraph pattern matching with renewed interest in recent years. Several researchers have proposed exact or approximate methods for searching for patterns in graph databases consisting of several relatively small graphs as well as a single large graph (e.g., [34], [41], [19], [45], [42], [11], [33]). Examples of exact methods include GraphQL [20], GADDI [43], and SPath [44]. We have already discussed GraphQL in detail in Section III. GADDI [43] uses a distance index based on the number of discriminating substructures between pairs of nodes. Zhao et al. [44] propose an indexing technique that is based on neighborhood signatures and shortest paths. In future work, we plan to comprehensively compare our pattern matching algorithm with these alternatives – as we noted earlier, our pattern census algorithms can use any exact pattern matching algorithm as a subroutine. Other work in this area has focused on variants of the pattern matching problem. Fan et al. [15], [14] allow an edge in the pattern to represent a short path in the database graph, and the matching is based on the concept of *bounded simulation*. Similarly, Zou et al. [46] propose *distance join* where the query is a pattern along with a distance δ. A match exists iff for two vertices v_i and v_j that are connected by an edge in the pattern, the shortest path between their images v'_i and v'_j is $\leq \delta$.

VII. Conclusions

We introduced a new type of graph analysis query, called a pattern census query, which has broad applications in a variety of domains including targeted marketing, brokerage analysis, and social sciences. We designed a general and flexible language for specifying pattern census queries, and developed efficient algorithms for answering such queries. Our comprehensive experimental evaluation over a prototype system that we have built illustrates that our algorithms can efficiently evaluate pattern census queries over large graphs. In future work, we plan to focus on approximation techniques for even larger graphs and also top-k query evaluation techniques to more efficiently identify the nodes with the highest pattern census counts.

Acknowledgments

We thank Dr. Huahai He and Prof. Ambuj K. Singh for providing us with GraphQL binaries. This work was funded by the Air Force Research Lab (AFRL) under contract FA8750-10-C-0191, and by NSF under grant IIS-0916736.

References

[1] EgoNet. http://egonet.sf.net.

[2] Neo4j open source NoSQL graph database. http://neo4j.org/.

[3] E. G. Allan, Jr., W. H. Turkett, Jr., and E. W. Fulp. Using network motifs to identify application protocols. In *GLOBECOM*, 2009.

[4] N. Alon, R. Yuster, and U. Zwick. Color-coding. *Journal of the ACM*, 42:844–856, July 1995.

[5] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17:209–223, 1997.

[6] U. Alon. Network motifs: theory and experimental approaches. *Nature Reviews Genetics*, 8(6):450–461, 2007.

[7] M. Baiesi. Scaling and precursor motifs in earthquake networks. *Physica A*, 360(2):4, 2004.

[8] A. L. Barabasi and R. Albert. Emergence of Scaling in Random Networks. *Science*, 286(5439):509–512, 1999.

[9] R. Burt. *Structural holes: The social structure of competition*. Harvard University Press, 1992.

[10] D. Cartwright and F. Harary. Structural balance: a generalization of Heider's theory. *Psychological Review*, 63(5):277–93, 1956.

[11] J. Cheng, Y. Ke, W. Ng, and A. Lu. FG-index: towards verification-free query processing on graph databases. In *SIGMOD*, 2007.

[12] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *KDD*, 2009.

[13] M. Consens and A. O. Mendelzon. GraphLog: a visual formalism for real life recursion. In *PODS*, 1990.

[14] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. Adding regular expressions to graph reachability and pattern queries. In *ICDE*, 2011.

[15] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: from intractable to polynomial time. *VLDB*, 2010.

[16] M. Fellows, G. Fertin, D. Hermelin, and S. Vialette. Sharp tractability borderlines for finding connected motifs in vertex-colored graphs. In *ICALP*, 2007.

[17] R. H. Güting. GraphDB: Modeling and querying graphs in databases. In *VLDB*, 1994.

[18] M. Gyssens, J. Paredaens, and D. van Gucht. A graph-oriented object database model. In *PODS*, 1990.

[19] H. He and A. K. Singh. Closure-tree: An index structure for graph queries. In *ICDE*, 2006.

[20] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*, 2008.

[21] S. Itzkovitz and U. Alon. Subgraphs and network motifs in geometric networks. *Phys. Rev. E*, 71, 2005.

[22] B. Jiang and C. Claramunt. Topological analysis of urban street networks. *Environment and Planning B: Planning and Design*, 2004.

[23] J. M. Kleinberg, S. Suri, E. Tardos, and T. Wexler. Strategic network formation with structural holes. *Sigecom Exchanges*, 7:284–293, 2008.

[24] U. Leser. A query language for biological networks. *Bioinformatics*, 21:33–39, January 2005.

[25] D. Liben-Nowell and J. Kleinberg. The link prediction problem for social networks. In *CIKM*, 2003.

[26] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, 1967.

[27] S. Mangan and U. Alon. Structure and function of the feed-forward loop network motif. *Proc. of The National Academy of Sciences*, 2003.

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

[29] W. E. Moustafa, G. Namata, A. Deshpande, and L. Getoor. Declarative analysis of noisy information networks. In *ICDE GDM Workshop*, 2011.

[30] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45:167–256, 2003.

[31] N. Przülj. Biological network comparison using graphlet degree distribution. *Bioinformatics/computer Applications in the Biosciences*, 2007.

[32] P. Sen, G. M. Namata, M. Bilgic, L. Getoor, B. Gallagher, and T. Eliassi-Rad. Collective classification in network data. *AI Magazine*, 2008.

[33] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *VLDB*, 2008.

[34] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, 2002.

[35] L. Sheng and G. Özsoyoglu. A graph query language and its query processing. In *ICDE*, 1999.

[36] M. A. Smith, B. Shneiderman, N. Milic-Frayling, E. Mendes Rodrigues, V. Barash, C. Dunne, T. Capone, A. Perer, and E. Gleave. Analyzing (social media) networks with nodexl. In *Proceedings of the fourth international conference on Communities and technologies*, 2009.

[37] Y. Tian, R. C. Mceachin, C. Santos, D. J. States, and J. M. Patel. Saga: a subgraph matching tool for biological graphs. *Bioinformatics*, 23:232–239, January 2007.

[38] S. Valverde and R. V. Solé. Network motifs in computational graphs: A case study in software architecture. *Physical Review E*, 2005.

[39] V. Van Kerrebroeck and E. Marinari. Ranking by loops: a new approach to categorization. *Phys. Rev. Lett.*, 101:098701, 2008.

[40] S. Wasserman and K. Faust. *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994.

[41] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *SIGMOD*, 2004.

[42] S. Zhang, M. Hu, and J. Yang. TreePi: A novel graph indexing method. In *ICDE*, 2007.

[43] S. Zhang, S. Li, and J. Yang. GADDI: distance index based subgraph matching in biological networks. In *EDBT*, 2009.

[44] P. Zhao and J. Han. On graph query optimization in large networks. *Proc. VLDB Endow.*, 3:340–351, September 2010.

[45] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: tree + delta <= graph. In *VLDB*, 2007.

[46] L. Zou, L. Chen, and M. T. Özsu. Distance-join: pattern match query in a large graph database. *Proc. VLDB Endow.*, 2:886–897, August 2009.

Appendix

A. Subgraph Pattern Matching Algorithm

Algorithm 1 lists our proposed pattern matching algorithm.

B. Algorithms for Evaluating Pattern Census Queries

Here we provide pseudocode for our proposed algorithms for evaluating ego-centric pattern census queries. We also discuss how to extend these algorithms to handle subpatterns, and pairwise union and intersection search neighborhoods. As discussed in Section II, subpatterns are useful when the user wants to look for the entire pattern, but only requires checking whether a portion of the pattern is present in a node's neighborhood. On the other hand, pairwise union and intersection neighborhoods are often of interest when the focus is on pairs of nodes rather than single nodes (e.g., in link prediction or entity resolution).

Node-driven Algorithms: Our proposed pivot indexing algorithm is listed in Algorithm 2. Next we discuss how to extend the algorithm for subpatterns and pairwise neighborhoods.

```
Input : Database graph $G = (V_G, E_G)$; pattern
        $P = (V_P, E_P)$; A permutation of the pattern nodes
        $v_1, v_2, ..v_{|V_P|}$ s.t. each prefix is a connected
        component of $P$
Output: Matches of $P$ in $G$
for $v \in V_P$ do
    $C(v) \leftarrow \{\}$;
    for $n \in V_G$ s.t. $l(n) = l(v)$ do
        if $profile(v) \sqsubseteq profile(n)$ then
            $C(v) \leftarrow C(v) \cup n$;
            for $v' \in N(v)$ do $CN(n, v, v') \leftarrow C(v') \cap N(n)$;
repeat
    for $v \in V_P, n \in C(v), v' \in N(v)$ do
        if $CN(n, v, v') = \{\}$ then $C(v) \leftarrow C(v) - n$;
    for $v \in V_P, n \in C(v), v' \in N(v), n' \in CN(n, v, v')$ do
        if $n' \notin C(v')$ then
            $CN(n, v, v') \leftarrow CN(n, v, v') - n'$;
until no change in $C$ and $CN$ ;
/* Let $M_i$ denote the matches of pattern
   subgraph $v_1, \dots, v_i$.                          */
for $n \in C(v_1), n' \in CN(n, v_1, v_2)$ do
    $M_2 \leftarrow M_2 \cup (n, n')$;
for $i = 2$ to $|V_P| - 1$ do
    for $(n_1, \dots, n_i) \in M_i$ do
        for $n_{i+1} \in \bigcap_{v_j \in N(v_{i+1}), j<i+1} CN(n_j, v_j, v_{i+1})$ do
            if $n_{i+1}$ not in $(n_1, \dots, n_i)$ then
                $M_{i+1} \leftarrow M_{i+1} \cup (n_1, \dots, n_{i+1})$;
return $M_{|V_P|}$;
```

Algorithm 1: Subgraph Pattern Matching Algorithm

```
Input : Database graph $G$; pattern $P$; set of nodes $V_\sigma(G)$;
        neighborhood radius $k$
Output: The number of matches of $P$ within $k$ hops of each
        node of $V_\sigma(G)$
$v \leftarrow argmin_{x \in V_P}\{d(x, argmax_{y \in V_P}\{d(x, y)\})\}$;
$max_v \leftarrow d(v, argmax_{y \in V_P}\{d(x, y)\})$;
for $u \in V_P$ do
    for $i \leftarrow 1$ to $max_v$ do
        if $d(v, u) \geq i$ then $distant[i] \leftarrow distant[i] \cup u$
$\mathcal{M} \leftarrow$pattern-match($G, P$);
$PMI_v \leftarrow$build-pmi-index($\mathcal{M}, v$);
for $n \in V_\sigma(G), n' \in N_k(n)$ do
    if $max_v + d(n, n') \leq v$ then
        $counts[n] \leftarrow counts[n] + |PMI_v[n']|$;
    else
        for $M \in PMI_v[n']$ do
            if $\mu(distant[k - d(n, n') + 1], M) \subseteq N_k(n)$ then
                $counts[n] \leftarrow counts[n] + 1$;
return $counts$;
```

Algorithm 2: Pivot Indexing Algorithm

```
Input : Database graph $G$; pattern $P$; set of nodes $V_\sigma(G)$;
        neighborhood radius $k$
Output: The number of matches of $P$ within $k$ hops of each
        node of $V_\sigma(G)$
$\mathcal{M} \leftarrow$ pattern-match($G,P$);
/* Index the matches on all the pattern
   nodes                                      */
$PMI \leftarrow$ build-pmi-index($\mathcal{M}, V_P$);
$S \leftarrow V_\sigma(G)$;
$current \leftarrow$ Next element from S;
$\mathcal{M}_{current} \leftarrow \{\}$;
while $S$ is not empty do
    $S \leftarrow S - current$;
    if $prev = NULL$ then
        $N_1 \leftarrow N_k(current)$;
        $N_2 \leftarrow \{\}$; $\mathcal{M}_{current} \leftarrow \{\}$;
    else
        $N_1 \leftarrow N_k(current) - N_k(prev)$;
        $N_2 \leftarrow N_k(prev) - N_k(current)$;
    for $n \in N_1, M \in PMI[n]$ do
        if $V_M \subseteq N_k(current)$ then
            $\mathcal{M}_{current} \leftarrow \mathcal{M}_{current} \cup M$;
    for $n \in N_2$ do $\mathcal{M}_{current} \leftarrow \mathcal{M}_{current} - PMI[n]$;
    $counts[current] \leftarrow |\mathcal{M}_{current}|$;
    if there exists n s.t. $n \in S \cap N(current)$ then
        $prev \leftarrow current$;  $current \leftarrow n$;
    else
        $prev \leftarrow$ NULL;  $current \leftarrow$ Next elem. from S;
return $counts$;
```

Algorithm 3: Differential Counting Algorithm

Handling Subpatterns: In this algorithm, handling subpatterns is straightforward. As before, pattern matching is performed using the entire pattern graph; however, the pivot is selected from the set of subpattern nodes $V_{SP} \subseteq V_P$, and the distance checks are only done for the database graph nodes that match the subpattern nodes.

Handling Pairwise INTERSECTION and UNION: In the case of intersection and union, the outer loop (line 9) iterates over pairs of nodes $(n_1, n_2) \in V_\sigma^2(G)$, where $V_\sigma^2(G)$ is the set of selected pairs, and the $N_k(n)$ is replaced with the set of nodes in $N_k(n_1) \cap N_k(n_2)$ and $N_k(n_1) \cup N_k(n_2)$ for intersection and union, respectively (lines 10 and 15). The distance $d(n,n')$ is replaced with $max(d(n_1,n'), d(n_2,n'))$ and $min(d(n_1,n'), d(n_2,n'))$, respectively.

The adapted differential counting algorithm for handling individual node cases is listed in Algorithm 3. We omit the extensions for handling subpatterns and pairwise neighborhoods due to space constraints.

Pattern-driven Algorithm: The pseudocode for our proposed pattern-driven algorithm is listed in Algorithm 4 (we omit the clustering-based optimization for brevity).

Handling Subpatterns: To handle subpatterns, we use $\mu(V_{SP}, M)$ instead of V_M in the algorithm above. In other words, for each match M, we only consider its subgraph incident on the nodes in V_M that match nodes in the subpattern.

```
Input : Database graph $G$; pattern $P$; set of nodes $V_\sigma(G)$;
        neighborhood radius $k$; set of centers $\mathcal{C}$
Output: The number of matches of $P$ within $k$ hops of each
        node of $V_\sigma(G)$
$\mathcal{M}$=pattern-match($G,P$);
for $M \in \mathcal{M}$ do
    for $m \in V_M, m' \in V_M$ do
        if $d(\mu^{-1}(m), \mu^{-1}(m')) \leq k$ then
            $PMD_m[m'] \leftarrow d(\mu^{-1}(m), \mu^{-1}(m'))$;
        else
            $PMD_m[m'] \leftarrow k+1$;
    $Q \leftarrow V_M$;
    while $Q$ is not empty do
        $n \leftarrow argmin_{q \in Q} \sum_{m \in V_M} PMD_m[q]$;
        dequeue($Q,n$);
        $near \leftarrow$ TRUE;   $far \leftarrow$ TRUE;
        for $m \in V_M$ do
            if $PMD_m[n] > k$ then $near \leftarrow$ FALSE;
            if $PMD_m[n] < k$ then $far \leftarrow$ FALSE;
        if $near$ then $\mathcal{N}[M] \leftarrow \mathcal{N}[M] \cup n$;
        if not $far$ then
            for $n' \in N(n)$ do
                $noChange \leftarrow$ TRUE;
                for $m \in N(M)$ do
                    if $PMD_m[n'] = NULL$ then
                        $noChange \leftarrow$ FALSE ;
                        $PMD_m[n'] \leftarrow min(PMD_m[n] +$
                        $1, min_{c \in \mathcal{C}}(d(m,c) + d(c,n')))$;
                    if $PMD_m[n'] > PMD_m[n] + 1$ then
                        $noChange \leftarrow$ FALSE ;
                        $PMD_m[n'] \leftarrow PMD_m[n] + 1$;
                if not $noChange$ then enqueue($Q,n'$);
    $\mathcal{N}[M] \leftarrow \mathcal{N}[M] \cap V_\sigma(G)$;
    for $n \in \mathcal{N}[M]$ do $counts[n] \leftarrow counts[n] + 1$;
return $counts$;
```

Algorithm 4: Pattern-driven Algorithm

Handling Pairwise INTERSECTION and UNION: To handle INTERSECTION, we note that all the pairs in $\mathcal{N}[M]$ already have the pattern in the intersection of their neighborhood. Therefore, instead of adding the match M to each node in $\mathcal{N}[M]$, we add it to each node pair in $\mathcal{N}[M] \times \mathcal{N}[M]$. For UNION, for each match M, we partition the set $N(M)$ into all possible size 2 partitions P_1, P_2. We denote nodes reachable from P_1 and P_2 by $\mathcal{N}[P_1]$ and $\mathcal{N}[P_2]$, respectively. The match M is added for each pair of nodes $(n_1, n_2) \in \mathcal{N}[P_1] \times \mathcal{N}[P_2]$. Because of the requirement to partition the pattern in different ways, and perform the computation in on every partitioning way, pattern-driven UNION evaluation is only useful for very simple and selective patterns.