Fast Best-Effort Search on Graphs with Multiple Attributes*

Senjuti Basu Roy, *University of Washington Tacoma*, Tina Eliassi-Rad, *Rutgers University*, Spiros Papadimitriou, *Rutgers University*

Abstract—We address the problem of search on graphs with multiple nodal attributes. We call such graphs *WAGs* (short for *Weighted Attribute Graphs*). Nodes of a WAG exhibit multiple attributes with varying, non-negative weights. WAGs are ubiquitous in real-world applications. For example, in a co-authorship WAG, each author is a node; each attribute corresponds to a particular topic (e.g., databases, data mining, and machine learning); and the amount of expertise in a particular topic is represented by a non-negative weight on that attribute. A typical search in this setting specifies both connectivity between nodes and constraints on weights of nodal attributes. For example, a user's search may be: find three coauthors (i.e., a triangle) where each author's expertise is greater than 50% in at least one topic area (i.e., attribute). We propose a ranking function which unifies ranking between the graph structure and attribute weights of nodes. We prove that the problem of retrieving the optimal answer for graph search on WAGs is NP-complete. Moreover, we propose a fast and effective top-k graph search algorithm for WAGs. In an extensive experimental study with multiple real-world graphs, our proposed algorithm exhibits significant speed-up over competing approaches. On average, our proposed method is more than $7\times$ faster in query processing than the best competitor.

Index Terms—Weighted Attribute Graph, Graph Search, Top-k Algorithms

1 Introduction

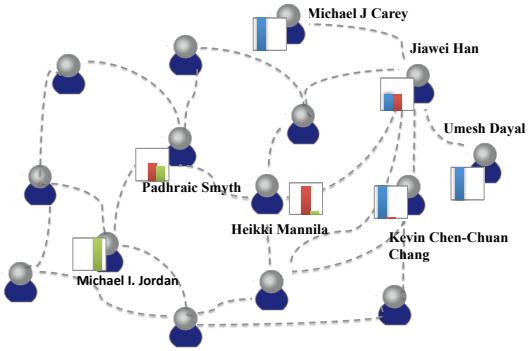
Graphs provide a natural way for representing entities that are "connected" (e.g., authors are connected by their co-authorships). A graph is commonly denoted by $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of nodes or vertices and $\mathcal{E}$ is the set of links or edges. We are interested in graphs where (1) nodes have multiple attributes (e.g., an author has multiple areas of expertise) and (2) attributes on nodes have non-negative weights associated with them (e.g., an author has varying degrees of expertise across different topics). We call such data graphs *Weighted Attribute Graphs* (or *WAGs* for short). WAGs are ubiquitous in real-world applications. Examples include (1) co-authorship networks with multiple expertise as attributes, (2) friendship networks with various activities (such as liking, commenting, or clicking on different types of posts) as attributes, (3) communication networks with various connection types as attributes, *etc*. Moreover, the outputs of mixed-membership community-discovery and role-discovery algorithms [2], [11], [13], [17] are WAGs. Such algorithms take as input a graph and assign multiple communities (or roles) to nodes, with varying memberships—i.e., they output WAGs. Note that graph search (the problem addressed in this work) is different than graph clustering even though both combine structure and node attributes. See [37] for an example of the latter.

An important task on WAGs is to quickly and effectively answer a user's search, where the search can include (1) constraints on the connectivity of nodes and (2) constraints on the weighted attributes. Consider the case where a WAG is produced by a mixed-membership community-discovery algorithm. Being able to search that output enables us to gain more insight into the data and its patterns (see Section 7.3 for case studies). In this work, we present a fast, best-effort solution for search on WAGs. We normalize weights on attributes per-node—i.e., the weights for all attributes within each node sum to one. This normalization makes the query semantics easier for the end user because it does not require the user to know about the weight distribution of attributes for the entire WAG. In addition, many applications naturally lend themselves to the per-node normalization case—e.g., finding connected ports in a communication network that are permanent ports (almost always used; 90% or more) or ephemeral ports (seldom used; 10% or less). Figure 1 depicts an example WAG and search query. If the application warrants it, our approach can easily be adapted to other types of normalization—e.g., per-attribute normalization where the sum of the weights for each attribute is one across all nodes.

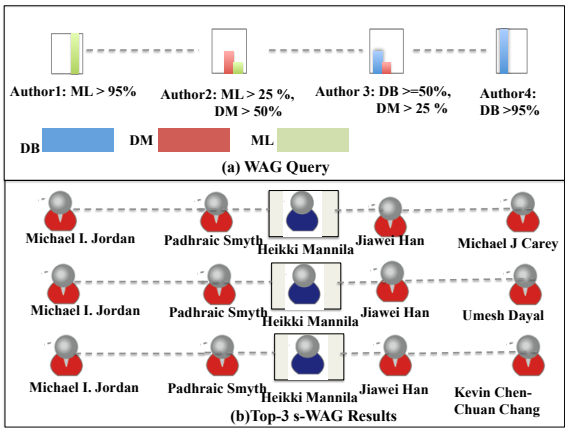
When addressing the problem of search on a WAG, one has to consider issues such as structure vs. weighted-attribute matching, point vs. range queries on weighted-attributes, exact vs. inexact algorithms, and optimal vs. approximate solutions. Our approach addresses *all* these issues, and has three components: `I-WAG`, `S-WAG`, and `R-WAG`. Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a matrix $\mathcal{W}$ whose w_{ij} entry corresponds to the weight of attribute j on node i, `I-WAG` builds a *hybrid* index on the graph that incorporates both the weighted attributes and the structure of the network. Upon receiving a search query, `S-WAG` utilizes the output of `I-WAG` to quickly retrieve the best k matches based on both weighted attributes and structure. `R-WAG` ranks the results by utilizing a novel ranking function that sums the divergence scores of the nodes in the solution.

- *senjutib@u.washington.edu,*
- *eliassi@cs.rutgers.edu,*
- *spapadim@business.rutgers.edu*

* This work was funded in part by LLNL under Contract DE-AC52- 07NA27344, by NSF CNS-1314603, by DTRA HDTRA1-10-1-0120, and by DAPRA under SMISC Program Agreement No. W911NF-12-C-0028.



(a) A partially constructed DBLP co-authorship network



(b) A query over the DBLP co-authorship network and top-3 results returned by S−WAG

Fig. 1. Example: The DBLP co-authorship network is a WAG. Each node is an author with three attributes: expertise in databases (DB), data mining (DM), and machine learning (ML). A graph query on the DBLP co-authorship network may wish to find a path of 4 authors (structural constraint) such that it connects ML researchers to DB researchers using DM researchers as intermediary. The latter property is enforced by using weights on nodal attributes (weight constraints; depicted by "bar heights"). Such a search with path structure and range constraints on attribute-weights is depicted on the top-half of Figure 1(b). The bottom half of Figure 1(b) describes the 3-best answers returned by our S−WAG. Note that author Heikki Mannila acts as "bridge" (primary expertise DM) to connect authors who are somewhat uniformly spread between ML, DM (i.e., Padhraic Smyth) and DB, DM (i.e., Jiawei Han). The expertise of the returned authors is shown in Table 1. A similar case study is presented in Section 7.3, but where the WAG is the output of a mixed-membership role-discovery algorithm.

S−WAG supports *range* and *point* queries on the weighted attributes. Range queries are flexible, where a weighted-attribute can be in a range of values—e.g., expertise in data mining must be greater than 50% (see top of Figure 1b). In point queries, a weighted-attribute must have a specific value—e.g., expertise in data mining must be 75%.

Author's Name	DB Weight	DM Weight	ML Weight
Michael I. Jordan	0	0	1
Padhriac Smyth	0	0.52	0.48
Heikki Mannila	0	0.90	0.10
Jiawei Han	0.50	0.49	0.01
Michael J. Carey	1	0	0
Umesh Agarwal	1	0	0
Kevin Chen-Chuan Chang	1	0	0

TABLE 1
Expertise in DB, DM, and ML of the authors returned in the results of Figure 1(b)'s search query. The weights are the ratio of the author's publications in a DB, DM, or ML conference listed in the DBLP data from 2005 to 2009.

Our contributions are as follows:

- *Problem:* We initiate the study of graph search on WAGs. Table 5 summarizes the existing graph search approaches and their differences with our S−WAG.
- *Flexibility:* We introduce a flexible querying mechanism on WAGs, and a novel ranking technique, R−WAG, which unifies the ranking of both structure and attribute weights into a single measure.
- *Efficiency:* We first prove that finding the optimal match for a given graph search on a WAG is NP-complete. Next, we introduce S−WAG for search on WAGs. S−WAG utilizes I−WAG's novel hybrid indexing scheme. I−WAG unifies a WAG's attribute weights, structural features (e.g., vertex degree), and graph structure into one structure that is easy to search. S−WAG uses a novel algorithm that is efficient and returns the "optimal" answer to a query in terms of its quality given the ranking function in R−WAG (see Lemma 1).
- *Effectiveness:* We demonstrate S−WAG's effectiveness and broad applicability through extensive case studies on real-world data and comparisons with the nearest competitor.

The rest of our paper is organized as follows. Section 2 outlines the notation used in the paper. Section 3 describes queries on WAGs. Section 4 introduces our ranking approach, R−WAG. Sections 5 and 6 present I−WAG and S−WAG. Section 7 presents our experiments. Section 8 discusses related works. Section 9 concludes the paper.

2 Preliminaries

Table 2 lists the notations used in this paper. Our data graph is a WAG, defined by $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and $\mathcal{W}$. $\mathcal{G}$ defines the structure of the WAG. $\mathcal{W}$, a node $\times$ attribute matrix, contains attribute weights for each node. As discussed in Section 1, the weights across the attributes are row-normalized. So, each entry $w_{i,j}$ is between 0 and 1, inclusive, and each row in $\mathcal{W}$ sum up to 1.

In this work, we consider undirected WAGs with no edge-weights. Extensions of I−WAG and S−WAG so they can incorporate directed WAGs with edge-weights are straightforward. However, modifying R−WAG to incorporate edge direction and weights in its ranking involves making non-trivial decisions about the importance of directionality and edge-weights. This is part of our future work.

Notation	Interpretation
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	the data graph with vertex set $\mathcal{V}$ & edge set $\mathcal{E}$
$\mathcal{R} = \{r_1, r_2, \ldots, r_l\}$	the set of l attributes defined on $\mathcal{V}$
$\mathcal{W}$	the node $\times$ attribute matrix containing attribute-weights
$w_{i,j}$	the weight of node i on attribute j
$\mathcal{T}$	matrix $\mathcal{W}$ concatenated with the node $\times$ degree vector, so $[\mathcal{W}, degrees]$
$H_q = (V', E')$	a graph query
W_q	the query-node $\times$ attribute matrix containing query attribute-weights
v_i	a node i in the data graph
v'_i	a node i in the graph query
G_s	a candidate subgraph in response to a graph query
$F(G_s, H_q)$	ranking function; takes as input two subgraphs
$D(v_i, v'_i)$	divergence score; takes as input two nodes

TABLE 2
Notations used in the paper

3 GRAPH QUERIES ON WAGS

S−WAG supports graph queries on any given WAG. A graph query on a WAG is a subgraph, which includes constraints on both connectivity and nodal-attribute weights. One extreme scenario of a graph query is one that only contains connectivity specifications between the participating nodes and no nodal-attribute annotations. The other extreme assumes that, in addition to connectivity specifications, a graph query also contains the attribute-weight specifications for all the query nodes. S−WAG offers a wide range of queries by supporting these extreme cases; and also those queries where only a subset of query nodes contain weights on a subset of attributes. In addition, the attribute weights can be specified as discrete values or as ranges.

Formally, a graph query is defined by $H_q = (V', E')$ and W_q (a node $\times$ attribute matrix, containing attribute-weights for each node). Relative to the data graph, H_q is a small graph. Based on two different types of queries on the nodes of H_q, we further define *point graph query* and *range graph query*.

Point graph queries (or simply point queries) contain $H_q = (V', E')$ and W_q, where each $w_{i,j} \in W_q$ contains a discrete value between 0 and 1, inclusive; and the rows of W_q must sum up to 1. Range graph queries (or simply range queries) contain $H_q = (V', E')$ and W_q. It offers flexible querying, as the user does not need to specify weight on every attribute, or even on every node. For the $w_{i,j} \in W_q$ that are specified, the attribute weight is specified as a range. Figure 1(b) describes the W_q of a range query.

4 RANKING WITH R−WAG

Given a graph query (point or range), our task is to return its top-k best matches. To do so, we need to rank the results considering *divergence* on the graph structure *and* on the weighted attributes. The choice of divergence measures is orthogonal to our work. Cha [3] provides a comprehensive survey on various similarity and distance measures. While divergence on a graph's weighted attributes is fundamentally different from the divergence on its structure, our proposed ranking approach, R−WAG, unifies these into one single function. Alternative approaches are also investigated—e.g., designing a ranking function that linearly combines weighted attribute divergence and structural divergence. However, such approaches lack generality and demand the system to *a priori* understand the relative importance between these two types of divergences.

4.1 R−WAG's Ranking Function

Designing a ranking function that is not ad-hoc but is general enough to accommodate both types of divergences is challenging. We take up a principled route, where the attribute and structural divergences are both expressed using Jensen Difference [24]. Suppose G_s is a candidate subgraph in response to the graph query H_q, then R−WAG's ranking for G_s with respect to H_q is defined as follows:

$$F(G_s, H_q) = \sum_{i=1}^{|V_s|} D(v_i, v'_i)$$

where $v_i \in G_s$ and $v'_i \in H_q$. Then, the divergence function $D(v_i, v'_i)$ is defined as follows:

$$D(v_i, v'_i) = \begin{cases} 1 & \text{if } v_i \text{ is an unmatched node} \\ JensenDiff(v_i, v'_i) & \text{otherwise} \end{cases}$$

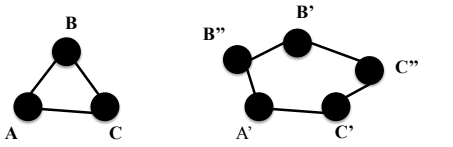
Jensen difference [24] is normalized so that a 0 value means a perfect match and a 1 value means a perfect non-match. The structural divergence is captured per *unmatched node*, which is an extra node that needs to be added to the solution to satisfy the graph query's connectivity requirements. Such a node will have maximal divergence—i.e., 1. We use the Jensen difference for our divergence function since it has a nice information-theoretic formulation. As mentioned before, R−WAG can use any divergence function. The definition of the Jensen difference is as follows:

$$F_w(v_i, v'_i) = \Sigma_{m=1}^{l}[\frac{w_{im} \times ln(w_{im}) + w_{i'm} \times ln(w_{i'm})}{2} - \frac{w_{im} + w_{i'm}}{2} \times ln(\frac{w_{im} + w_{i'm}}{2})]$$

Recall that w_{im} denotes the weight of i-th node for the m-th attribute; and l is the number of weighted attributes.

Consider the pattern query in Figure 2(a) and a candidate answer to this query in Figure 2(b). For the purpose of illustration, let us assume that the query nodes A, B, and C, respectively, match WAG nodes A′, B′, and C′. Then, the ranking score of the subgraph in Figure 2(b), is : $JensenDiff(A, A') + JensenDiff(B, B') + JensenDiff(C, C') + 1 + 1$. Note that, a Jensen Difference of 1 is added for each of the unmatched nodes: B″ and C″. Also note that the proposed ranking semantics forces a candidate answer to have at least as many nodes and edges as the query contains; and that the additional unmatched nodes are allowed to warrant approximate matching over the structure, leading to a Jensen Difference of 1 for each unmatched nodes. Our ranking function allows us to design efficient WAG search algorithms since it is monotonic.

Now that we have discussed our ranking function, we can formally define the graph search problem on a WAG.



a) A pattern query b) A candidate answer

Fig. 2. An example query in (a) and a candidate answer in (b). The ranking score of a candidate sub-graph is the summation of the node-wise Jensen Differences for matched nodes (A′, B′, and C′) and the maximum Jensen Difference of 1 for each unmatched node (B′′ and C′′).

4.2 Problem Definition

Top-k Graph Search on a WAG: Given a WAG, a graph query (point or range), and an integer k, identify a set of k subgraphs from the WAG such that (*i*) the k subgraphs are ranked in ascending order of their overall divergence scores (F) to the graph query; and (*ii*) any subgraph that is not present in the set has a larger divergence score with respect to the graph query than the overall divergence score of the k-th subgraph.

4.3 Problem Hardness

Theorem 1. *The top-1 graph search problem is NP-Complete under F.*

Proof. (Sketch) We prove the NP-completeness considering WAGs with zero nodal attributes (i.e., $l = 0$).

Given a WAG $\mathcal{G}$, a query H_q, $k = 1$, the decision problem is whether there exists a sub-graph G_s in $\mathcal{G}$, such that $F(G_s, H_q) = 0$ (i.e., zero divergence between G_s and H_q). It is easy to show that this problem is in NP. NP-hardness is proved by reducing an instance of the sub-graph isomorphism [9] to an instance of our problem. Given two graphs G and G', the problem of sub-graph isomorphism is to check whether G' contains a sub-graph that is isomorphic to G. To reduce the subgraph isomorphism to an instance of our problem, $G' = G_s$, and $G = H_q$.

Given the above instance of the graph search problem, the objective is to find G_s, such that $F(G_s, H_q) = 0$ and there exists a solution of the subgraph isomorphism problem, if and only if, a solution to our instance of the graph search problem exists.

□

The top-k graph search problem is also NP-complete, since the simpler top-1 search is NP-complete.

5 Hybrid Indexing with I−WAG

Figure 3 depicts the key components and processes for graph search on a WAG. Specifically, for a given WAG, I−WAG builds and maintains a novel hybrid index structure. This structure is subsequently used to speed-up graph search during query time and facilitates the matching over both the weighted attributes and the structure of the WAG. The I−WAG structure is hybrid because it is over weighted attributes, structural features, and graph structure. We present I−WAG in detail next and defer the discussion on S−WAG (i.e., the search algorithm) to Section 6.

First, a balanced tree is constructed for the matrix $\mathcal{T}$. Recall that $\mathcal{T}$ consists of the weight matrix $\mathcal{W}$ concatenated with the vertex degree vector (i.e. $[\mathcal{W}, degrees]$). The I−WAG structure has the following properties:

- The root has between 1 and M entries (unless it is a leaf). M is the degree of the tree. All intermediate nodes have between $M/2$ and M entries each. Each leaf has between M and $2M$ entries.
- A leaf is a WAG vertex with attribute weights $\mathcal{R}$.
- Each intermediate node is a minimum bounding rectangle (MBR) of dimensionality $l + 1$ (recall l is the number of weighted attributes). An intermediate node with j entries has j children, and the bounding rectangle indexes the space of its children.

Second, to enable fast structural search, at every leaf (i.e., a vertex in $\mathcal{G}$), I−WAG maintains an inverted-list index of the immediate neighbors of each graph vertex.

The I−WAG structure is built by transforming each row in $\mathcal{T}$ into a multi-dimensional point, where the number of dimensions corresponds to the number of weighted attributes plus structural features (all computed at the vertex level). For this work, we use one structural feature, vertex degree. However, other vertex-level features (such as clustering co-efficient, *PageRank*, etc) can also be used by our method. Subsequently, this space is indexed. I−WAG "fuses" an R-tree family structure [12] with an inverted-list index to enable fast search over weighted attributes *and* structural features of WAG vertices, *plus* the WAG's graph structure (i.e., connectivity). Moreover, I−WAG has a `getNext()` interface over $\mathcal{T}$, which returns the next best candidate vertex of $\mathcal{G}$ that has the least divergence with respect to a given WAG query vertex.

Each leaf of the I−WAG structure corresponds to a vertex in the input WAG; and contains an *inverted-list*, which represents the set of vertices that are its immediate neighbors in the input WAG. While there exist more complex structural indexing techniques in the literature, these additional indexing capabilities require expensive pre-processing. As we show in Section 7, the I−WAG structure we propose here is sufficient for fast best-effort search.

getNext(): Given any graph query $H_q = (V', E')$, for each query vertex $v'_i \in V'$, a `getNext()` call is issued to the I−WAG structure to return the vertex in $\mathcal{G}$ that has the smallest divergence with v'_i (as described in Section 4). Specifically, `getNext()` searches the I−WAG structure based on the type of query on v'_i (range or point); then it prunes the possible matches for v'_i; and finally it selects the node with the least divergence in terms of both weighted attributes and structural features.

Range Query: Given a range query vertex v'_i, its weight distributions over the specified attributes and degree are used to transform v'_i to a query rectangle in the l+1 dimensional attribute-space. As an example, a query vertex with range specification such as <0.2 on DB and >0.5 on DM, and degree of at least 3 can be translated as ranges $[0.0, 0.2)$, $(0.5, 1.0]$,

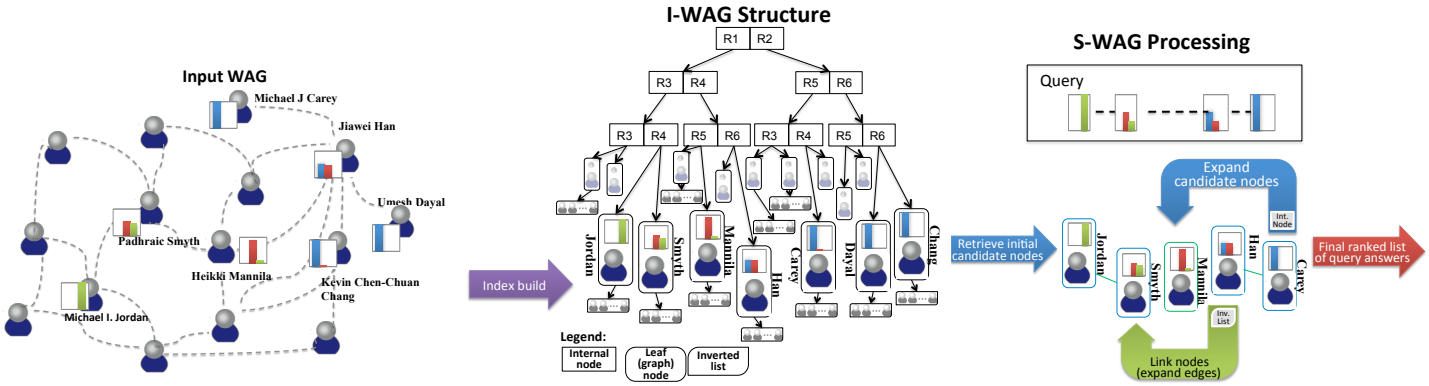


Fig. 3. Overview of graph search on a WAG: The input WAG is used to build a hybrid I−WAG structure; see Section 5. When a query arrives, the I−WAG structure is used to efficiently retrieve the top-k best matches incrementally, considering attribute weights, structural features, and graph structure; see Section 6.

and $[3, MaxDegree]$,[1] respectively; and a query rectangle can be formed. For the attributes whose weights are not specified in the query explicitly, their respective ranges are considered to be $[0, 1]$. To enable search, S−WAG starts from the root of the I−WAG structure and traverses down the tree. If a current node is not a leaf and if it overlaps with the query rectangle, it continues to search further down in that subtree. If the current node is a leaf, and the leaf is contained in the query rectangle, that leaf is returned as a candidate answer.

Point Query: Given a point query vertex v', it is first transformed to a point in l+1 dimensional space. Similar to range queries, the search for the best matching vertex to v' begins at the root of the I−WAG structure and traverses down. It tries to prune some of the intermediate branches, and keeps a list of active branches to be expanded further. The algorithm terminates once the active branch list is empty.

Pruning: We use the bounding rectangles of the I−WAG structure to decide whether or not to search inside the subtree that it indexes. These rectangles can be searched efficiently using MINDIST and MINMAXDIST [21]. MINDIST is the optimistic distance between the point under consideration and any object indexed by the minimum bounding rectangle (MBR). Specifically, if v'_i is inside the MBR, $MINDIST(MBR, v'_i) = 0$. Otherwise, $MINDIST(MBR, v'_i)$ is the minimal possible divergence from the query point v'_i to any node inside or on the perimeter of the rectangle. On the other hand, MINMAXDIST is the pessimistic distance between the point and any object that the MBR indexes. So, $MINMAXDIST(MBR, v'_i)$ is the smallest possible upper bound of distances from the point v'_i to the MBR. The following property is used for effective pruning:

$$\begin{aligned} MINDIST(MBR, v'_i) &\leq NN(v, v'_i) \\ &\leq MINMAXDIST(v'_i, MBR) \end{aligned}$$

where NN denotes the nearest-neighbor distance.

Given a query vertex, the I−WAG structure is progressively searched from the root, using the technique described above. The aforementioned pruning criteria are applied, to decide whether to prune or to add an intermediate node to the existing active branches. However, at a leaf, the actual divergence between the query vertex v' and the leaf v (which corresponds to a vertex in the WAG) is computed, their divergences with that of the smallest divergence seen so far are compared, and it is updated if necessary. When the active branch list is empty, the vertex with the smallest divergence to v' is returned.

To obtain the candidate subgraph with the smallest divergence to the graph query, the inverted-lists at the selected leaves are searched for the edges that produce the smallest divergence. This ensures that the candidate subgraph returned as an answer to the graph query has the smallest divergence in terms of weighted attributes, structural features, and graph structure.

6 Searching with S−WAG

We present an efficient and "optimal" algorithm (referred to as S−WAG in the experiments) for the top-k graph search problem on WAGs. "Optimality" here is with respect to answer quality, and not necessarily with respect to query processing time (see Lemma 1). Our algorithm uses the I−WAG structure described in the previous section. A naive algorithm will enumerate all possible candidate results before it determines the final top-k results. This naive approach is prohibitive for even moderately large graphs.

Given any graph query, S−WAG executes the following tasks. (1) It uses the `getNext()` interface of the I−WAG structure to retrieve the best candidate vertex for every query vertex. (2) It uses the inverted-list structure of the I−WAG structure, to retrieve the best candidate edges for every query edges. (3) When k answers are not fully computed, the algorithm determines whether to expand structurally (i.e., introduce additional vertices to connect the candidate vertices that represent the endpoints of a query edge), or to issue `getNext()` calls to retrieve the next best candidate vertex from the I−WAG structure. (4) It returns the top-k matches ranked by increasing order of their overall divergence scores (as described in Section 4). In order to perform this last step efficiently, the algorithm maintains a *threshold* value that captures the minimum divergence that an unseen or partially computed candidate subgraph may have. The algorithm achieves early-termination when the *threshold* is not smaller than the score

1. $MaxDegree$ is the highest vertex degree in $\mathcal{G}$.

of the k-th best result thus far. In order to accomplish tasks (3) and (4) described above, s−WAG exploits the overall divergence score as a *threshold*, which is monotonic.

Range Query: Algorithm 1 describes the pseudocode for s−WAG. Given the query vertices, the task is to gradually retrieve the candidate vertices in a round-robin manner by issuing getNext() calls to the I−WAG structure. Note that the notion of vertex match for range query is binary—i.e., as long as a vertex in the WAG satisfies the range conditions specified on each attribute of a query vertex, the WAG vertex is returned as a match for the query vertex. Next, the search algorithm "stitches" the vertices together to recover the least divergent graph structure for the graph query. We define *candidate edge* to that end. A candidate edge is a path (can be a single edge) that is representative of a query edge in the WAG. At least one endpoint of a candidate edge corresponds to a candidate vertex. The candidate edge is *complete* if both endpoints correspond to candidate vertices, or else the candidate edge is *partially complete*. A complete candidate edge does not need to be expanded further, whereas, the partially complete candidate edges may needed to be expanded. For every candidate edge (complete or partially complete), the algorithm keeps track of its current divergence. This way, a candidate edge-set $\mathcal{C}$ is maintained during the execution of the algorithm.

Note that for range queries, all candidate vertices that satisfy the specified range constraints of the corresponding query vertex are equally desirable. Therefore, the overall divergence score primarily counts the additional vertices (each corresponding to Jensen Difference of 1) that are to be used to connect the candidate vertices in order to match the query structure for the purpose of ranking.

During query processing, when the k results are not fully computed, s−WAG (see Algorithm 1) issues additional calls (in a round-robin manner) to the I−WAG structure to retrieve the next best candidate vertex that may be used to match the query. If the I−WAG structure no longer returns a new candidate vertex with the getNext() call, the algorithm attempts to expand the retrieved candidate vertices structurally, one by one, until k results are computed. However, a more interesting scenario occurs when the algorithm has already found k results, but it needs to guarantee that the current k results are indeed the global k-best answers, based on the ranking function. This step is validated by considering the value of the *threshold* (described next).

The value of the *threshold* is the minimum divergence score that any unseen candidate answer may have. In order to compute the *threshold* efficiently, for every query edge e', the algorithm keeps track of two additional quantities when considering its candidate edges: (1) the smallest divergence score considering its candidate edges, and (2) the latest divergence score of its last partially complete candidate edge. Given the query H_q with $|E'|$ edges, the *threshold* at that step is computed by aggregating the respective divergences of the query edges that lead to the smallest sum. The algorithm terminates when the divergence score of the k-th best result does not exceed the threshold. Note that, as soon as the algorithm retrieves a new vertex from the I−WAG structure or expands the partially computed answers by an edge, it updates the *threshold* value.

Algorithm 1 Optimal Algorithm for Top-k Range Query

Require: s−WAG-Index, Query $H_q = (V', E')$, k
1: Issue getNext() in a round-robin fashion to get the next best candidate vertex for each query vertex v'_i
2: Form candidate edges and add to candidate edge set $\mathcal{C}$
3: stitch candidate edges and form k answers
4: Update $ResultSet$ with top-k answers based on divergence
5: *threshold* = min sum of divergences between query & candidate edges
6: **while** the $\mathcal{C}$ is not empty **do**
7: **if** (*threshold* $<$ *DivergenceScore*($ResultSet[k]$)) **then**
8: Issue getNext() in round robin fashion
9: Update $ResultSet$ with top-k candidate outputs
10: **else**
11: Output $ResultSet$ as the best k-results
12: break
13: **end if**
14: **end while**
15: **return** $ResultSet$

Point Query: The optimal algorithm for searching point queries is similar (in principle) to that for range queries. The main differences are that (1) ranking of a point query requires stricter matches on the weighted attributes; and (2) during query processing, if the top-k answers are not fully computed, then the algorithm has to decide whether to expand node-wise (i.e., issue another getNext() call), or to expand structurally. For that, the algorithm computes two *threshold* values – one associated with issuing another getNext() call and another associated with structural expansion. It chooses the expansion that produces the smaller *threshold*, updates the *threshold*, and rechecks the pruning conditions. The algorithm overall only differs in matching the WAG vertices with query vertices considering Jensen Difference. Unlike range query, a match between a query vertex and a WAG vertex is not binary (i.e., match or no-match), rather, there is an associated divergence score between $[0, 1]$. The final ranking score, therefore, has to account for divergence arises due to both vertex and pattern match. Beyond that, the rest of the s−WAG is akin to that of range query.

Lemma 1. *The top-k results returned by* s−WAG *are optimal in the context of answer quality, considering the ranking function in* r−WAG *(described in Section 4.1).*

Proof. (Sketch) We prove this lemma by contradiction. Without loss of generality, let us assume that the above claim is incorrect—i.e., suppose given a query, s−WAG does not return the top-k results in the context of answer quality. This means that after s−WAG terminates, there is at least one result not returned by s−WAG whose ranking score is smaller than the ranking score of the k-th result, returned by s−WAG. Recall that low ranking score denotes a smaller divergence score and a better match. As described before, r−WAG is monotonic—i.e., the ranking score only increases as s−WAG expands to include more vertices. Also by definition, the *threshold* contains the best (i.e., minimum) score of any unseen candidate result. So, the value of *threshold* being tracked by s−WAG is smaller than the score of the k-th result when the

algorithm terminates. Note this is a contradiction, because s−WAG will terminate only when *threshold* is at least as large as the score of the k-th result. □

7 Experimental Evaluation

i−WAG and s−WAG were implemented in JDK 1.6. All experiments were conducted on a 2.66GHz Intel Core i7 processor with 8GB memory, 500GB HDD, running OS X. The Java virtual memory size is set to 1GB. The closest competitor of s−WAG is G−ray [28]. We had to modify G−ray to accept WAGs as input (see details below). We implemented the new version of G−ray, which we call WAG−ray, using Matlab 7.12.0 and the same system configuration used in [28].

7.1 Data

Table 3 lists the real-world data sets used in our experiments. We have four smaller datasets with thousands of vertices and edges; and three large datasets with millions of vertices and edges.

Our smaller datasets include two DBLP co-authorship graphs, one from databases (DB) and another from data mining (DM).[2] Each author constitutes a vertex in the WAG. The attributes for DBLP-DB are SIGMOD, VLDB, and ICDE, and that of DBLP-DM are KDD, ICDM, and SDM. Given a vertex (i.e., an author) and an attribute (i.e., a conference), the weight of that attribute is computed by calculating the percentage of the author's publications in the corresponding conference. Data in these graphs are from 2005 to 2009.

Our third smaller dataset is an IP communication network from Lawrence Berkley National Laboratory (LBL) that contains the communication information between different IP addresses in a 24-hour time window.[3] Each unique IP address corresponds to a vertex in the LBL graph. An edge between a pair of vertices exists if they have had any communication between them. We have three different attributes for each IP address: well-known ports in range $[0, 1023]$, registered ports in range $[1024, 49151]$, and ephemeral ports in range $[49152, 65535]$. The weight on each attribute corresponds to the percentage of communications on that port for the given IP address.

Our fourth and last smaller dataset is an online social network extracted from YouTube.[4] Each vertex here is a YouTube user; and two users, x and y are connected if either x's profile page links to y's, or y's profile page links to x's, or both. The attributes are discovered through a mixed-membership role-discovery algorithms [13]. Each vertex contains 6 different roles (i.e., attributes). They are:

1) *Global Hub*: Propensity to be a global hub vertex. Vertices with high weights on this attribute have high out-degree, high in-degree (though lower than out-degree), high PageRank, and high biconnected component score.
2) *Periphery*: Propensity to be a periphery vertex. Vertices with high weights on this attribute have slightly higher than default eccentricity.
3) *Authority*: Propensity to be an authority vertex. Vertices with high weights on this attribute have high PageRank, high in-degree, high total degree, and high biconnected component score.
4) *Cliquey*: Propensity to be a cliquey vertex. Vertices with high weights on this attribute have a high clustering coefficient.
5) *Local Hub*: Propensity to be a local hub vertex. Vertices with high weights on this attribute have high out-degree and high biconnected component score.
6) *Nondescript*: Propensity to be a nondescript vertex. Vertices with high weight on this attribute do not stand out in terms of their values for degree, clustering coefficient, PageRank, eccentricity, or biconnected component score.

Our larger graphs consist of a Yahoo! IP communication network, an Amazon product co-purchasing network, and a social-reader network from a large US-based media company. Our first larger dataset is from Yahoo!.[5] Each record in the Yahoo! dataset contains start and end times of communication, source and destination IP addresses, their respective port addresses, and other related fields. The Yahoo! graph consists of IP addresses as vertices and communications between IPs as edges. The attributes (as well their respective weights on each vertex) are extracted following the same procedure as described above for the LBL dataset. The attribute weights are extracted by considering different port numbers through which a unique IP communicates. These weighted attributes are similar to the LBL weighted attributes: well-known ports, registered ports, and ephemeral ports.

Our second of the larger datasets is a co-purchasing graph from the Amazon website.[6] It contains product metadata and review information for $548,552$ different products (Books, music CDs, DVDs and VHS video tapes). Each product contains: title, sales rank, list of co-purchased products, and product reviews (time, customer-id, rating, number of votes, number of people that found the review helpful). Each product is a vertex in the graph, and an edge exists if two products were co-purchased. The attributes are extracted from product ratings (in $[1, 5]$), and categorized further: excellent (rating 5), good (ratings 3 or 4), and bad (ratings 1 or 2). The attribute weights are the normalized ratio of the total number of votes on a given attribute over the total number of votes.

Our third of the larger datasets is a social-reader network from a large US-based media company with approximately $1.5M$ vertices and $6.8M$ edges. Each vertex in this dataset is a user of an online social network, who has subscribed to the social reader application. We extract a friendship social network based on users who are friends and who have read at least one common article during the week of April 2, 2012. We have topic information on each article. So, the attributes for a vertex are the propensity to read articles on news, arts and entertainment, and technology. The weights on the attributes correspond to the normalized ratio of the total number of articles read in a given attribute over the total number of articles read by the user.

2. http://www.dblp.org/search/index.php
3. http://www.icir.org/enterprise-tracing/download.html
4. http://socialnetworks.mpi-sws.org/data-imc2007.html
5. http://webscope.sandbox.yahoo.com/
6. http://snap.stanford.edu/data/amazon-meta.html

Dataset	# Nodes	# Edges	# Attributes	Sparsity
Amazon	548,000	1,788,725	3	0.01
DBLP-DB	6,413	40,493	3	0.32
DBLP-DM	2,104	8,816	3	0.33
YouTube	15,480	86,788	6	0.42
Yahoo!	1,941,878	3,929,607	3	0.52
LBL	3,055	15,577	3	0.53
SocialReader	1,475,348	6,831,944	3	0.6

TABLE 3
Datasets used in our experiments. The table is sorted in ascending order of *sparsity* values. Sparsity is measured by: $\frac{\#0's \text{ in } \mathcal{W}}{\#\text{entries in } \mathcal{W}}$.

7.2 Experimental Setup and Methodology

To the best of our knowledge, previous graph search methods cannot deal with graph queries on graphs with multiple weighted attributes—i.e., WAGs. However, rather than just comparing against naive baselines, we appropriately modified an existing work, called G-ray [28], that efficiently supports search on graphs with unweighted, single attributes (i.e., each vertex has a single attribute and no weights). We primarily report comparative studies between S-WAG and this modified competitor, which we call WAG-ray. The performance of other baseline methods is typically even worse. We describe WAG-ray next.

WAG-ray: We appropriately modify G-ray [28], which was originally designed for search over single attribute graphs, to handle WAGs. G-ray utilizes a goodness score based on random walk with restarts to measure how well a subgraph matches a graph query. G-ray uses a scalable algorithm to find and rank candidate subgraphs by first finding seed vertices (or seeds for short), then expanding neighborhoods around seeds, and finally finding bridges to connect the neighborhoods. We refer to the modified G-ray as WAG-ray. Our modifications are as follows. We create an augmented graph by adding l 'dummy' vertices to the WAG (these dummy vertices correspond to the l attributes of each vertex); then we connect each original vertex to all l dummy vertices with edge weights that correspond to the weights of the attributes on that vertex. This process introduces $|V| \times l$ additional edges to the original graph. Next, given a query vertex with multi-attribute constraints, the seed finding process now selects a seed based on the query condition. We initiate the random walk from the seed. After these changes, our query becomes the standard WAG-ray query for range constraints. WAG-ray is implemented only for range queries. We note that for point queries, the attribute matching function in WAG-ray needs to be modified to consider approximate matches over weighted attributes. Unless otherwise stated, we set the number of iteration in WAG-ray to 10, and the escape probability to 0.9. These numbers were suggested in [28].

Additional Baselines: In addition to the experiments reported in this paper, we also performed extensive experiments on three other baseline methods, which we summarize here. A naive algorithm was implemented that adopts a brute-force approach to find the top-k answers of a graph query. Given a WAG query, it first finds a set of mapping candidate vertices for every vertex in the query, with a linear scan over $\mathcal{W}$. It then establishes edges between every pair of candidate vertices that represent the endpoints of an edge in the query. Finally, it ranks all enumerated valid answers based on the overall divergence score and returns the best k results. This algorithm takes several minutes to process queries on smaller graphs (with thousands of nodes and edges), and does not terminate within 30 minutes on larger graphs (with millions of nodes and edges).

With a similar ranking function to R-WAG but one that ranks a query edge relative to an edge in the WAG (instead of ranking a query vertex), we extended our proposed indexing and the query processing approaches to accommodate edge-based ranking. In this case, instead of a vertex, the I-WAG structure indexes each edge based on the attribute weights of its endpoints. Similarly, the algorithm also retrieves one candidate edge at a time (as opposed to a candidate vertex), and performs the matching accordingly. In our experiments, this method improves the query processing time by less than 30% on average, while it takes significantly longer time to build the I-WAG structure. For brevity, these results are omitted.

Finally, we also implemented a greedy algorithm, which uses the I-WAG structure to find the best matching candidate vertex of the highest-degree query vertex as the seed. After that, it greedily expands and matches the neighbors of the seed vertex with the query vertices using our proposed ranking function. Due to its greedy nature, this algorithm is about 20% faster in query processing than S-WAG on an average. However, this comes at a substantial penalty in quality: about 55% of the time it does not return the best solutions. These results are also omitted for brevity.

Queries: We primarily considered four different types of structures for graph query—namely, star, path, loop, and clique. For our performance experiments, a point query vertex is generated from the underlying $\mathcal{W}$ matrix uniformly at random. A range query vertex is generated from a point query, where we arbitrarily introduce ranges on the specific attribute weights. In either case, the graph search is one of the four aforementioned graph queries.

Summary of experiments: Our experiments are divided into three groups: (1) case studies, (2) build-time experiments and (3) run-time experiments. We present a case study using the YouTube data. All five algorithms described above are evaluated in the build time and runtime experiments using the datasets described earlier. For brevity, we report results on a subset of our datasets. Unless otherwise noted, the omitted results are similar to the ones depicted. Moreover, since WAG-ray is the most competitive baseline to S-WAG, we only report results on S-WAG and WAG-ray.

Summary of Results: Our build time experiments demonstrate that I-WAG scales well with increasing graph size, number of attributes, and sparsity. The run-time experiments are conducted by varying the number of nodes, number of returned results (i.e., k), sparsity, selectivity, and weights of the node attributes. We observe that the run-time of S-WAG outperforms WAG-ray in all the cases. Moreover, we observe that the difference in run-time between S-WAG and WAG-ray is smallest under very high sparsity. This observation can

Dataset	# Vertices	# Edges	# Attributes	**Sparsity**	Build Time s-WAG (sec)	Build Time WAG-ray (sec)	# Queries after which s-WAG recovers the difference in build time with WAG-ray
SocialReader	1,475,348	6,831,944	3	0.60	911	65.7	127
LBL	3,055	15,577	3	0.53	11.4	0.03	2
Yahoo!	1,941,878	3,929,607	3	0.52	486	18.9	66
YouTube	15,480	86,788	6	0.42	83.7	4.2	7
DBLP-DM	2,104	8,816	3	0.33	6	0.02	1
DBLP-DB	6,413	40,493	3	0.32	18.6	0.06	5
Amazon	548,000	1,788,725	3	0.01	327.6	5.2	32

TABLE 4

Build times (in seconds of wall clock) for the I-WAG structure and WAG-ray's index. The table is sorted in descending order on sparsity of the vertex $\times$ attribute matrix, $\mathcal{W}$. Naturally, WAG-ray is faster than I-WAG in build time, since the number of links it creates in the augmented graph is linear to the number of vertices (when number of attributes is constant). The last column lists the number of queries for WAG-ray after which s-WAG recovers the difference in build time. The small values in this column suggest that the combination of I-WAG and s-WAG are a better graph search approach than WAG-ray.

be explained with the reasoning that under high sparsity, a weighted attribute graph effectively becomes a graph with one nodal attribute. Therefore, the effectiveness of WAG-ray improves, as WAG-ray was originally designed for graphs with one nodal attribute. Build time of WAG-ray is superior than that of I-WAG as WAG-ray uses a shallow indexing structure (recall that it creates a small number of dummy nodes, same as the number of nodal attributes and establishes connections between the actual nodes and the dummy ones), leading to comparatively smaller build time. However, since the run-time of WAG-ray is much higher than that of s-WAG, s-WAG levels the extra build time with a reasonably small work-load. Additionally, we demonstrate the pruning effectiveness of I-WAG for a given query workload. This observation corroborates the effectiveness of s-WAG for weighted attribute graphs.

7.3 Case Studies

In Section 1, we presented a case study on the DBLP co-authorship network. Here we present another case study on the YouTube data ($\approx 15.5K$ vertices and $\approx 87K$ edges), where the attributes were produced by a mixed-membership role-discovery algorithm (see details in Section 7.1). We asked the following graph searches:

- How many paths exist, where a primarily global hub (i.e. *Global Hub* ≥ 0.5) is connected to a primarily local hub (i.e. *Local Hub* ≥ 0.5), which in turn is connected to a primarily peripheral vertex (i.e. *Periphery* ≥ 0.5)? This is an interesting query because a periphery vertex uses a local hub as a bridge to connect to a global hub. s-WAG finds 412 such patterns in the YouTube graph. These patterns represent 1.8% of the 3-vertex paths in this graph.
- How many triangles exist that involve a global hub (i.e. *Global Hub* ≥ 0.5), a local hub (i.e. *Local Hub* ≥ 0.5), and a cliquey vertex (i.e. *Cliquey* ≥ 0.5)? This query is interesting because a cliquey vertex has a connected global- and local-hub as neighbors. s-WAG finds 119 such patterns in the YouTube graph. These patterns represent 0.7% of the triangles in this graph.
- How many star patterns with 5 vertices exist, where the center of the star is an authority (i.e. *Authority* ≥ 0.5), and all other remaining vertices are regular users (i.e. *Nondescript* ≥ 0.5)? This is an interesting query because an authority has a handful of nondescript users as its neighbors. s-WAG finds 137 such patterns in the YouTube graph. These patterns represent 1.1% of the 5-vertex stars in this graph.
- How many loop patterns with 4 vertices exist, where all the vertices are local hubs (i.e. *Local Hub* ≥ 0.5)? This query is interesting because local hubs are forming a loop with each other. s-WAG finds 523 such patterns in the YouTube graph. These patterns represent 3.5% of the 4-vertex cycles in this graph.

Baseline Algorithms: We also run the case studies on the baseline algorihtms. The results indicate that WAG-ray finds the same set of answers as s-WAG does; however, the processing time is about 3.75 times longer compared to that of s-WAG. Our brute force baseline algorithm is significantly slower and does not terminate in 1 hour in most of the cases. Our third baseline solution (i.e., edge-based solution) also discovers the same patterns, but takes 70% longer time to build its index structure. Finally, the greedy algorithm (i.e., the final baseline algorithm) finds those appropriate patterns about 47% of the time on an average.

As these results show, conducting graph search on WAGs is a powerful data exploration tool and s-WAG is an effective solution towards that end.

7.4 Build Time Experiments

What is the build time of the I-WAG structure on different datasets? Table 4 presents the build time results. For WAG-ray, the build time is the time it takes to create the augmented graph. This could be considered as a "shallow index." As is evident, the I-WAG structure is efficient and scales well with increasing network size. Given similar sparsity value, build times are comparable (between DBLP-DB and DBLP-DM) in I-WAG, even when the number of vertices increases by 3 times. With a $302\times$ increase in the number of vertices between DBLP-DB and Yahoo!, build-time increases only by $26\times$ in I-WAG. Amazon, despite being a large graph

with very low sparsity (i.e., very few zeros in its $\mathcal{W}$ matrix), has a small build time (< 6 minutes). I-WAG takes less time to build sparse $\mathcal{W}$ matrix, as it needs to create a smaller number of intermediate nodes in its structure to index the vertices of the WAG.

WAG-ray's simpler index takes less time to build than I-WAG. However, as shown in the very last column of Table 4, S-WAG is able to recover this time after processing a small number of queries—e.g., 1 query for DBLP-DM, and 66 queries for Yahoo!.

7.5 Runtime Experiments

We compare the efficiency of S-WAG vs. WAG-ray. We report the average query processing time (Avg QPT) by varying result size (i.e., k), varying the number of query vertices (since the queries have specific graph patterns, the number of edges are also known accordingly), varying sparsity, and then varying query selectivity and attribute weights. For these runtime experiments, we consider four different types of graph queries: star, path, loop, and clique. Our query workload consists of 80 queries (20 queries of a particular graph search), and we report the average query processing time. Finally, we present results describing the pruning effectiveness of I-WAG.

Query-processing time as a function of result size: We compare query-processing time of S-WAG and WAG-ray by varying k (i.e., result size). Figure 4 shows these results: S-WAG outperforms WAG-ray on both the smaller and larger graphs. Among the larger graphs, the difference in query processing time is more significant in Amazon than in Yahoo!. The very low sparsity of Amazon's weighted attribute matrix (i.e., $\mathcal{W}$) makes the computation favorable to S-WAG, while WAG-ray was primarily designed for graphs that contain one attribute per vertex (i.e., very high sparsity of the weighted attribute matrix).

Query-processing time as a function of number of vertices: We compare query-processing time of S-WAG and WAG-ray by varying the number of vertices in the query. Figure 5 lists the results. S-WAG outperforms WAG-ray both on the smaller and larger graphs. Both S-WAG and WAG-ray scale well with the increasing number of query vertices. Among the larger graphs in our set, Amazon exhibits the best performance with increasing number of vertices. This observation is again due to the very low sparsity value of the weighted attribute matrix (i.e., $\mathcal{W}$) of Amazon, which helps it scale well with the increasing number of vertices in the query.

Query-processing time as a function of sparsity in the weighted attribute matrix: In these experiments, we vary the sparsity of the matrix $\mathcal{W}$, and report the average query processing time of S-WAG and WAG-ray. For each vertex, we compute the entropy of its weight distribution, and sort vertices based on their entropy values. That is, the vertices at the top are those whose distributions are closest to uniform. To satisfy certain sparsity percentage larger than its original sparsity, we scan this sorted list from the bottom, and transform the membership of a vertex to a single attribute (i.e., attribute that has the largest weight gets 1, the rest get 0 values) until the desired sparsity is satisfied. In these experiments, k is set to 5, and the number of vertices in the query is set of 5. Figure 6 depicts our results for this experiment. We observe that the query processing time increases with increasing sparsity for S-WAG, but decreases for WAG-ray. The impact is more in the larger graphs compared to the smaller graphs. Note that with increasing sparsity, WAG-ray for a WAG becomes the standard G-ray [28], which positively impacts its query processing time. Conversely, S-WAG takes more time with increasing sparsity (more zeros in $\mathcal{W}$), since it requires the generation of more candidate subgraphs.

Query-processing time as a function of query selectivity and attribute weights: We first vary query selectivity and measure the average query processing time. We profile the intermediate nodes of I-WAG to ensure certain selectivity x and design the query predicates that satisfy the bounding values of those selected intermediate nodes. Then, we demonstrate the average query processing time by varying weights. For a query with n nodes and l attributes, we choose one of the attributes r_i at random for each node, and vary its weight in five different windows: $[0, 10\%]$, $[0, 20\%]$, $[0, 30\%]$, $[0, 40\%]$, $[0, 50\%]$. The other attributes of that node are re-adjusted based on the change on r_i. For breivty, these results are presented for the DBLP-DB and Yahoo!. Figure 7 contains the results. S-WAG outperforms WAG-ray in all the cases.

Pruning effectiveness of I-WAG: Here, we consider a query workload of 80 queries (each with 5 nodes) and measure the average percentage of nodes that is pruned by I-WAG across our datasets. Figure 8 depicts the results. I-WAG is able to prune 60% of the nodes on average. The pruning is highest on the Amazon dataset due to its very low sparsity.

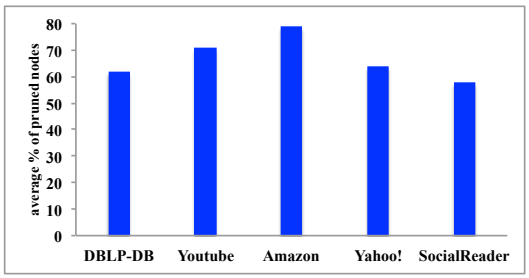
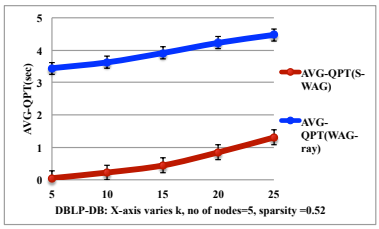


Fig. 8. I-WAG is able to prune more than 60% of the nodes across our datasets on average. This experiment considered star, path, loop, and clique pattern queries with 5 nodes each.

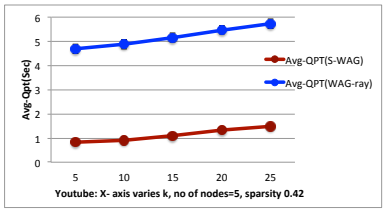
7.6 Discussion

Both WAG-ray and S-WAG scale linearly with increasing graph size in build time. At the same time, WAG-ray outperforms S-WAG in build time almost consistently. This is unsurprising, since the indexes in WAG-ray are rather "shallow," as it was not originally designed to perform search over WAGs. This slightly higher build time in S-WAG is compensated by its very efficient query processing time (results in Table 4).

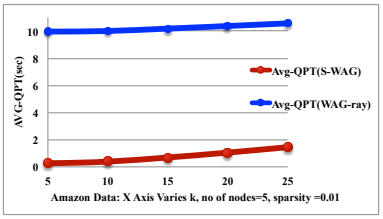
For the query processing time, both WAG-ray and S-WAG scale well, under varying graph size, query size, number of returned results, or query selectivity and weights. However, S-WAG outperforms WAG-ray consistently in query processing time, irrespective of the sparsity of the weighted attribute matrix. The difference in query processing time is the highest



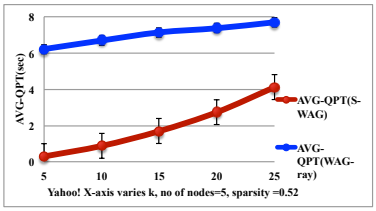
(a) DBLP-DB



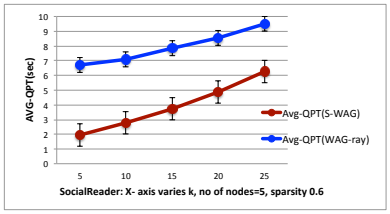
(b) YouTube



(c) Amazon

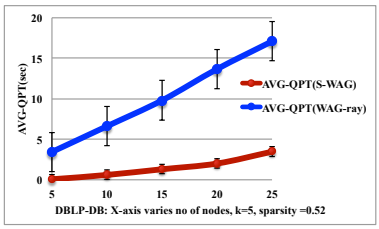


(d) Yahoo!

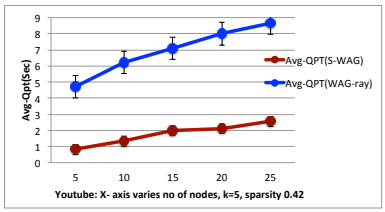


(e) SocialReader

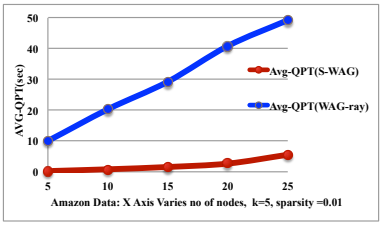
Fig. 4. Comparison of query processing time between `s-WAG` and `WAG-ray` when varying result size k. Number of vertices is set to 5, and the original sparsity of the respective $\mathcal{W}$ matrices are used. Both algorithms scale linearly with increasing k. `s-WAG` outperforms `WAG-ray` in all the cases.



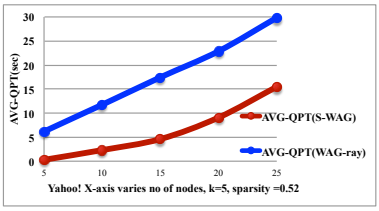
(a) DBLP-DB



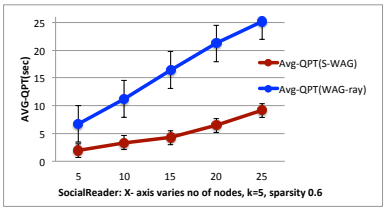
(b) YouTube



(c) Amazon



(d) Yahoo!

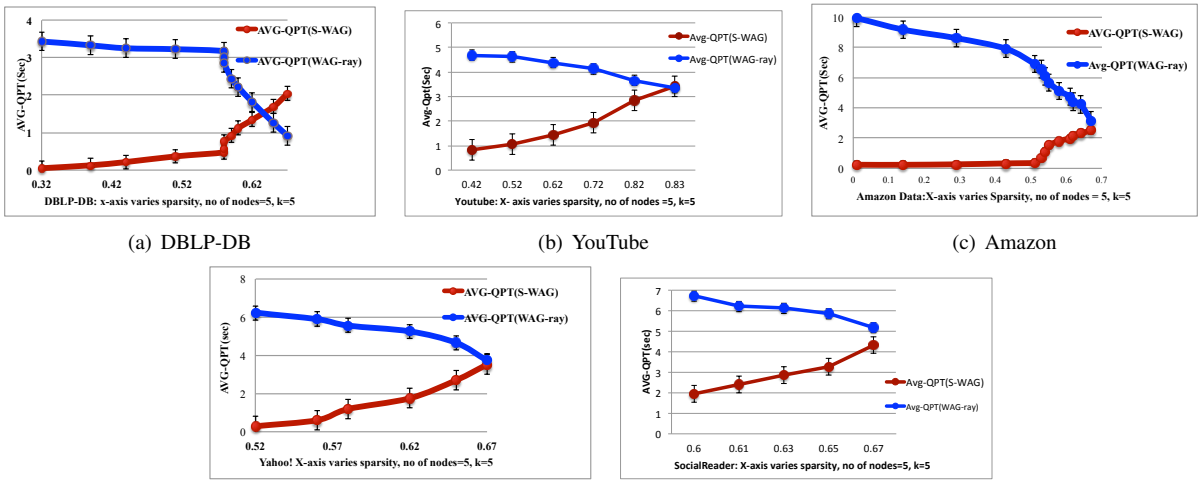


(e) SocialReader

Fig. 5. Comparison of query processing time between `s-WAG` and `WAG-ray` when varying number of query vertices. Result size k is set to 5, and the original sparsity of the respective $\mathcal{W}$ matrices are used. Unlike `WAG-ray`, `s-WAG` scales almost linearly with increasing number of vertices. This observation is most prominent for the Amazon dataset that has a very low sparsity value of 0.01. `s-WAG` significantly outperforms `WAG-ray` in all the cases.

for WAGs with low sparsity (like Amazon), compared to the ones with high sparsity (such as SocialReader). For further validation purposes, we conducted a large set of experiments, where we synthetically varied the sparsity of the weighted attribute matrix of the underlying WAGs (results in Figure 6). The impact of sparsity became even more apparent in these experiments. Specifically, high sparsity in weighted attribute matrix essentially turns a WAG into a graph with one attribute per node. In such cases, the difference in query processing time between `s-WAG` and `WAG-ray` reduces considerably. Overall, considering both build time and query processing time, our experimental results demonstrate that the combination of `I-WAG` and `s-WAG` are a better graph search approach than `WAG-ray` for WAGs.

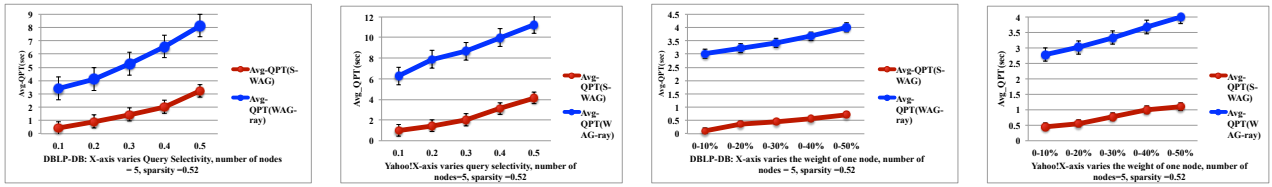
Our current implementation uses *node degree* as a dimension while building `I-WAG` (other dimensions are the nodal attributes). This feature allows us to easily perform degree-biased ranking during search. For example, two nodes that have exactly the same attribute weights could still be differentiated, if one has higher degree than the other. With trivial modifications inside `I-WAG` (e.g., the actual nodes can be sorted in descending degree inside an intermediate node in `I-WAG`, such that getNext() always returns the highest degree node first), the former (i.e., higher degree node) would be returned before the latter (i.e., the lower degree node) during the search. For the case of DBLP data, this gives us an easy solution for ranking a prolific author higher than a non-prolific author (who may be a fresh PhD graduate). Furthermore, there exists an easy extension to handle additional query constraints inside `I-WAG`. For example, if the query contains



(a) DBLP-DB (b) YouTube (c) Amazon

(d) Yahoo! (e) SocialReader

Fig. 6. Comparison of query processing time between `s-WAG` and `WAG-ray` when varying sparsity. Result size k is set to 5, and the number of vertices in the query is 5. With increasing sparsity, `WAG-ray` processes queries with less time, whereas `s-WAG` requires more time to process queries on extremely sparse WAGs. However for the larger graphs with millions of vertices and edges, `s-WAG` outperforms `WAG-ray` under any sparsity value.



(a) DBLP-DB Varying Selectivity (b) Yahoo! Varying Selectivity (c) DBLP-DB Varying Weight (d) Yahoo! Varying Weight

Fig. 7. Comparison of query processing time between `s-WAG` and `WAG-ray` when query selectivity and nodes weights are varied. Number of vertices is set to 5, and the original sparsity of the respective $\mathcal{W}$ matrices are used. `s-WAG` outperforms `WAG-ray` in all the cases. The maximum difference in query processing is observed for the Amazon dataset, which has a very small sparsity value of 0.01. This demonstrates the effectiveness of `s-WAG`.

clustering coefficient [29] or PageRank [18] constraints, each of these additional properties could be infused as an additional dimension inside `I-WAG`. After that, `s-WAG` will not require any further modifications to return results that match these additional constraints as well. At the same time, `I-WAG` could be extended to consider heterogeneous attributes as it is commonly seen in knowledge graphs with appropriate adaptations in the ranking function. The `I-WAG` and `s-WAG` will remain unaltered after that. An interesting question here is how to extend our work for time evolving WAGs, which involves designing `I-WAG` that is updated efficiently. We realize that updates in the attribute-weights of the nodes or in the graph structure (e.g., addition and.or deletion of nodes and/or edges) could be easily handled with *incremental index maintenance* by allowing insert() and delete() operations on the nodes and the edges in `I-WAG`. Of course, if the underlying WAG changes too much, it is better to build everything from scratch.

8 Related Work

To the best of our knowledge, ours is the first work towards *graph search* on weighted attribute graphs (WAGs). Next, we compare and contrast our work with existing literature on graph search, pattern matching in graphs, graph indexing, keyword search on databases, and other related tasks.

Graph Search and Graph Pattern Matching: The literature here is extensive [28], [4], [40], [27], [7], [16], [39], [34]. What sets our work apart is that our graphs have multiple weighted attributes on their nodes. Table 5 provides a comparative list between previous works and `s-WAG`. Tong *et al.* [28] propose a pattern matching problem over a large directed graph (based on *reachability* constraints) and a corresponding approximate matching algorithm to select the *best* set of patterns when an exact match is not possible, considering one attribute per node. Subsequently, Zou *et al.* [40] extend the problem by designing constraints on *distance* instead of *reachability* and design algorithms for that problem. Our problem is tangentially different due to the existence of weights over vertex attributes.

Top-k pattern matching problems are studied in [39], [34]. However, these works do not consider attributes on the nodes. Therefore, the proposed algorithms are primarily limited to pattern matching over structure, and do not lend an easy extension for our problem. A comprehensive survey describing variations among different graph matching problems, general- and specific-solution approaches, and evaluation techniques

can be found in [8].

Graphs without attributes	Tian & Patel (2008) Zou, et. al. (2007) Zeng, et. al. (2012)
Graphs with single attribute	Tong et al. (2007) Cheng, et al. (2008) Kahn et al. (2011) Zhu et al. (2011)
Graphs with multiple weighted attributes (i.e., WAGs)	S-WAG

TABLE 5
Various graph search approaches. Only S-WAG considers the problem of search on WAGs.

Keyword Search on Databases: The database literature has extensively studied the problem of keyword search over relational databases [1], [14]. Fundamentally, we consider a different problem here, since search on WAGs needs to consider the structural matching of the input graph, which the former body of works does not need to consider. Therefore, the algorithms suggested for the former problem do not lend themselves to graph search on WAGs.

Unified similarity measures to consider structural and attribute match: In [28], the similarity measures of a pattern query with a graph that has *one attribute per node* is defined in a unified manner, considering *reachability* constraints using random-walk-with-restart probabilities. A recent work [37] studies the *graph clustering problem*, where nodes can have multiple attributes with respective weights, and a unified similarity measure is defined considering both structural and attribute properties akin to that of [28]. While the clustering problem studied in [37] does not lend itself to perform graph search, we do note that in our experimental study, we have implemented WAG-ray, which is an extension of [28] using a unified distance measure considering structural match and multiple node attributes. As it appears, the unified distance measure in WAG-ray is exactly same as that of [37].

Graph Indexing: Given a graph database, graph indexing techniques aim at indexing the graphs based on the frequent substructures present in them [38], [33], [31]. To that end, *GraphGrep* [22] is a famous representative of the path-based indexing approaches. GraphGrep enumerates all existing paths up to a certain length l_p in a graph database G, then selects these paths as indexing features. In comparison to a path-based indexing approach, Yan *et al.* [32] use graphs as basic indexing features, which is often referred to as a graph-based indexing approach. Later on, Zhao *et al.* [36] describe tree-based indexing features and demonstrate how these features can circumvent the disadvantages that are present in the previous two approaches. Our work is different in principle, since we wish to perform fast best-effort graph search when the vertices have multiple attributes with varying weights.

Inference Queries in Graphical Models: A probabilistic database [6] may exhibit tuple-existence uncertainty, or attribute-value uncertainty, or a combination of both. Kanagal and Deshpande [15] propose efficient indexing schemes on correlated probabilistic databases. The nodal attribute weights in WAGs bear resemblance with attribute-value uncertainly. However, we consider the problem in the context of graphs, and the existence of edges between the nodes makes our model significantly different from the problems related to probabilistic databases.

Efficiently evaluating inference queries [5] has been a major research area in the probabilistic reasoning community. Observe that, our problem is different in nature. Unlike the former body of work, the existence of an edge in our weighted attribute graph is not probabilistic and, furthermore, the nodes of a WAG contain multiple weighted attributes.

OLAP in Multidimensional graphs: Zhao *et al.* [35] propose OLAP functionalities in multidimensional graphs. They present *Graph Cube*: an aggregate graph within every possible multidimensional space, by taking into account both attribute aggregation and structural summarization of the graphs. While *Graph Cube* is designed only to tackle OLAP queries, our I-WAG structure is useful in answering any subgraph query on WAGs. Unlike Zhao *et al.*'s work [35], the vertices in our problem contain multiple attribute weights, which make our problem significantly different in principle.

Other work: Search in graphs, particularly social networks, is an active area of research (e.g., [26], [25]), since vertices with weighted attributes arise naturally.

Our work has the potential to help in analyses that aim to understand graphs with vertex attributes [19], [20]; or work that leverages such graphs for some particular purpose [10]. Of particular relevance here are recommendations based on social network graphs [30]. In general, heterogeneous graphs [23] (which may have attributes or labels associated with both nodes as well as edges) arise in countless applications; our work focuses in particular on efficient search in graphs with multiple weighted vertex-attributes.

9 Conclusions

We define Weighted Attribute Graphs (WAGs), which can model a wide range of data arising in diverse scenarios and applications; and study the problem of graph search on WAGs. Although, as we prove, finding the optimal answer for a given graph search on a WAG is NP-complete, we introduce a three-part approach—consisting of R-WAG, I-WAG, and S-WAG—that can perform fast best-effort search on WAGs. I-WAG produces a novel hybrid index structure that incorporates weighted attributes, structural features, and the graph structure. S-WAG uses a novel algorithm to efficiently return the best answers to a graph query. R-WAG ranks the results based on a unified measure of both weighted-attribute and graph-structure divergences. We demonstrate the effectiveness and scalability of our approach based on extensive experiments on real-world data, exhibiting up to 7× better query response times.

References

[1] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, 2002.
[2] E. M. Airoldi, D. M. Blei, S. E. Fienberg, and E. P. Xing. Mixed membership stochastic blockmodels. *JMLR*, 9:1981–2014, 2008.
[3] S.-H. Cha. Comprehensive survey on distance/similarity measures between probability density functions. *Int'l J. of Mathematical Models and Methods in Applied Sciences*, 1(4):300–307, 2007.
[4] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang. Fast graph pattern matching. In *ICDE*, 2008.
[5] R. G. Cowell, A. P. David, S. L. Lauritzen, and D. J. Spiegelhalter. *Probabilistic Networks and Expert Systems*. Springer, 1999.

[6] N. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *VLDB*, 16(4):523–544, 2007.
[7] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: From intractable to polynomial time. *VLDB*, 2010.
[8] B. Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. In *AAAI Fall Symposia*, pages 45–53, 2006.
[9] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. 1979.
[10] S. Ghosh, N. K. Sharma, F. Benevenuto, N. Ganguly, and P. K. Gummadi. Cognos: Crowdsourcing search for topic experts in microblogs. In *SIGIR*, pages 575–590, 2012.
[11] S. Gilpin, T. Eliassi-Rad, and I. Davidson. Guided learning for role discovery: Framework, algorithms, and applications. In *KDD*, 2013.
[12] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.
[13] K. Henderson, B. Gallagher, T. Eliassi-Rad, H. Tong, S. Basu, L. Akoglu, D. Koutra, C. Faloutsos, and L. Li. Rolx: structural role extraction & mining in large graphs. In *KDD*, 2012.
[14] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.
[15] B. Kanagal and A. Deshpande. Indexing correlated probabilistic databases. In *SIGMOD*, pages 455–468, 2009.
[16] A. Khan, N. Li, X. Yan, Z. Guan, S. Chakraborty, and S. Tao. Neighborhood based fast graph search in large networks. In *SIGMOD*, pages 901–912, 2011.
[17] H. Li, Z. Nie, W.-C. Lee, C. L. Giles, and J.-R. Wen. Scalable community discovery on textual data with relations. In *CIKM*, 2008.
[18] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: bringing order to the web. Technical Report SIDL-WP-1999-0120, InfoLab, Stanford University, Stanford, CA, 1999.
[19] D. Quercia, L. Capra, and J. Crowcroft. The social world of Twitter: Topics, geography, and emotions. In *ICWSM*, pages 298–305, 2012.
[20] D. Quercia, R. Lambiotte, D. Stillwell, M. Kosinski, and J. Crowcroft. The personality of popular Facebook users. In *CSCW*, pages 955–964, 2012.
[21] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, pages 71–79, 1995.
[22] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, 2002.
[23] Y. Sun and J. Han. *Mining Heterogeneous Information Networks: Principles and Methodologies*. Morgan & Claypool Publishers, 2012.
[24] I. J. Taneja. *Generalized Information Measures and Their Applications*. www.mtm.ufsc.br/ taneja/book/book.html, 2001.
[25] J. Tang, S. Wu, B. Gao, and Y. Wan. Topic-level social network search. In *KDD*, pages 769–772, 2011.
[26] J. Tang, J. Zhang, R. Jin, Z. Yang, K. Cai, L. Zhang, and Z. Su. Topic level expertise search over heterogeneous networks. *MLJ*, 82(2):211–237, 2011.
[27] Y. Tian and J. M. Patel. Tale: A tool for approximate large graph matching. In *ICDE*, pages 963–972, 2008.
[28] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *KDD*, pages 737–746, 2007.
[29] D. J. Watts and S. H. Strogatz. Collective dynamics of small-worldnetworks. *Nature*, 393(6684):440–442, 1998.
[30] S. Wu, J. Sun, and J. Tang. Patent partner recommendation in enterprise social networks. In *WSDM*, pages 43–52, 2013.
[31] Y. Xie and P. S. Yu. CP-Index: On the efficient indexing of large graphs. In *CIKM*, pages 1795–1804, 2011.
[32] X. Yan, P. S. Yu, and J. Han. Graph indexing: A frequent structure-based approach. In *SIGMOD*, pages 335–346, 2004.
[33] J. Yang, S. Zhang, and W. Jin. Delta: Indexing and querying multi-labeled graphs. In *CIKM*, pages 1765–1774, 2011.
[34] 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.
[35] P. Zhao, X. Li, D. Xin, and J. Han. Graph cube: on warehousing and olap multidimensional networks. In *SIGMOD*, 2011.
[36] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: Tree + delta >= graph. In *VLDB*, pages 938–949, 2007.
[37] Y. Zhou, H. Cheng, and J. X. Yu. Graph clustering based on structural/attribute similarities. *VLDB*, 2(1):718–729, Aug. 2009.
[38] Y. Zhu, L. Qin, J. X. Yu, Y. Ke, and X. Lin. High efficiency and quality: Large graphs matching. In *CIKM*, pages 1755–1764, 2011.
[39] L. Zou, L. Chen, and Y. Lu. Top-k subgraph matching query in a large graph. In *CIKM Ph.D. Workshop*, pages 139–146, 2007.
[40] L. Zou, L. Chen, and M. T. Özsu. Distancejoin: Pattern match query in a large graph database. *VLDB*, 2(1):886–897, 2009.

Senjuti Basu Roy is an Assistant Professor at the Institute of Technology at the University of Washington Tacoma. Prior to joining UW in 2012, she was a postdoctoral fellow at DIMACS at Rutgers University. Senjuti received her Ph.D. in Computer Science from the University of Texas at Arlington in 2011.

Tina Eliassi-Rad is an Associate Professor of Computer Science at Rutgers University. Before joining academia, she was a Member of Technical Staff and Principal Investigator at Lawrence Livermore National Laboratory. She earned her Ph.D. in Computer Sciences (with a minor in Mathematical Statistics) at the University of Wisconsin-Madison.

Spiros Papadimitriou is an Assistant Professor at the Department of Management Science & Information Systems at Rutgers Business School. Previously, he was a research scientist at Google, and a research staff member at IBM Research. He obtained his MSc and PhD degrees from Carnegie Mellon University; and was a Siebel scholarship recipient in 2005.