ABSTRACT

We address the problem of finding subgraphs that best match a user’s query on Weighted Attributed Graphs (WAGs). We define a WAG as a graph where nodes exhibit multiple attributes with varying, non-negative weights. An example of a WAG is a co-authorship network, where each author has multiple attributes, each corresponding 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 user query in this setting specifies both connectivity patterns between query nodes and constraints on attribute weights of the query nodes. For example, a user query may have a triangle connectivity pattern where each node is an author with expertise greater than 50% in at least one topic area (i.e., attribute). We propose a ranking function which unifies the matching on attribute weights over the nodes and on the graph structure. We prove that the problem of retrieving the optimal match for such queries is NP-complete. Moreover, we propose a fast and effective top-k pattern matching algorithm for weighted attributed graphs. In an extensive experimental study with multiple real-world datasets, 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 strongest competing method.

Categories and Subject Descriptors
H.2.8 [Database Applications]: Data mining; E.1 [Data Structures]: Graphs and networks

General Terms
Algorithms, Design, Performance, Experimentation

Keywords
Weighted attributed graphs, pattern matching

1. INTRODUCTION

Graphs provide a natural way for representing entities that are “connected”—e.g., people connected by their co-authorship relationships. A graph is often denoted by $\mathcal{G} = (V, \mathcal{E})$, where $V$ is the set of nodes or vertices and $\mathcal{E}$ is the set of links or edges. Given a graph $\mathcal{G}$, a long-standing problem of interest has been to find patterns that match user queries—e.g., find all triangles in a graph. An extension of the original pattern-matching problem is to look at graphs where nodes have one attribute [17, 23]. For example, given a co-authorship graph where each node contains the primary expertise of an author, find matches to a triangle query where the three nodes (in the triangle) are experts in databases, data mining, and machine learning, respectively. We are interested in the more general problem of pattern-matching on graphs where (1) nodes have multiple attributes (e.g., an author has multiple expertise) and (2) the node attributes 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 Attributed Graphs (or WAGs for short) and present a fast and effective solution to pattern-matching on WAGs. For simplicity, we assume weights on attributes are normalized per-node, i.e., the weights for all attributes within each node sum to one. iPaM can easily be extended to other cases, e.g., per-attribute normalization where the sum of the weights for each attribute is one across all nodes.

Figure 1 depicts an example WAG and pattern-query. A wide variety of real-world applications are naturally modeled as WAGs, from co-authorship networks with author-nodes having multiple expertise with varying degrees, to IP communication networks with IP-nodes having multiple functionalities (such as DNS server, Web server, and P2P client) with varying degrees.

When addressing the problem of pattern matching 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. We introduce iPaM (short for Index-based Pattern Matching), which addresses all these issues. More formally, iPaM takes as input a graph $\mathcal{G} = (V, \mathcal{E})$ and a matrix $W$ whose $w_{ij}$ entry corresponds to the weight of attribute $j$ on node $i$. iPaM then builds an hybrid index on the graph that incorporates both the weighted attributes and the structure of the network. Upon receiving a weighted pattern query, iPaM utilizes its index to quickly retrieve the best $k$ matched results based on both weighted-attributes and structure. iPaM’s ranking function is the sum of divergence scores of each matched and unmatched node in the solution. iPaM supports both range and point queries. Range queries are flexible, where a weighted-attribute can be in a range of values (see top of Figure 1b).

To summarize, our contributions are as follows:

- Problem: We introduce the notion of WAGs, and investigate
the results of Figure 1(b)’s pattern query.

Figure 2: Author Christos Faloutsos acts as “bridge” connecting authors whose primary expertise is in DB to authors whose expertise is more uniformly spread depicted on the top-half of Figure 1(b). The bottom half of Figure 1(b) describes the 4 databases (DB), data mining (DM), and machine learning (ML); and the remaining authors are primarily DB experts. Such a pattern query 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 iPaM. Note that for all three answers, author Christos Faloutsos acts as “bridge” connecting authors whose primary expertise is in DB to authors whose expertise is more uniformly spread between DB and DM. The expertise of the returned authors is shown in Figure 2.

Table 1 lists the notations used in this paper. Our data graph is a WAG, defined by $G = (V, E)$ and $W$. $G$ defines the structure of the WAG. For brevity, in this paper we only consider undirected graphs with no edge weights. $W$, a node × attribute matrix, contains attribute-weights for each node. For simplicity, we assume the weights across the attributes are normalized. So, each entry $w_{i,j}$ is between 0 and 1, inclusive, and the rows of $W$ sum up to 1.

The rest of our paper is organized as follows. Section 2 describes queries on WAGs. Section 3 introduces iPaM’s ranking function. Sections 4 and 5 present iPaM’s indexing and matching algorithms. Section 6 presents our experiments. Section 7 discusses related works. Section 8 concludes the paper.

2. PATTERN QUERIES ON WAGS

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

Formally, a pattern-query is defined by $H_q = (V', E')$ and $W_q$ (a node × 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 patterns on the nodes of $H_q$, we further define point pattern query and range pattern query.

Point pattern 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, 1), inclusive, and the rows of $W_q$ must sum up to 1. Range pattern queries (or simply range queries) contain $H_q =$.
Table 1: Notations used in the paper

<table>
<thead>
<tr>
<th>Notation</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( G = (V, E) )</td>
<td>the data graph</td>
</tr>
<tr>
<td>( R = {r_1, r_2, \ldots, r_l} )</td>
<td>the set of ( l ) attributes defined on ( V )</td>
</tr>
<tr>
<td>( W )</td>
<td>the node \times attribute matrix containing attribute-weights</td>
</tr>
<tr>
<td>( w_{i,j} )</td>
<td>the weight of node ( i ) on attribute ( j )</td>
</tr>
<tr>
<td>( I )</td>
<td>matrix ( W ) concatenated with the node ( x )-degree vector, so ([W, degrees])</td>
</tr>
<tr>
<td>( H_q = (V^q, E^q) )</td>
<td>a pattern query</td>
</tr>
<tr>
<td>( W_q )</td>
<td>the query-node \times attribute matrix containing query attribute-weights</td>
</tr>
<tr>
<td>( v_i )</td>
<td>a node ( i ) in the data graph</td>
</tr>
<tr>
<td>( v_i' )</td>
<td>a node ( i ) in the pattern query</td>
</tr>
<tr>
<td>( G_s )</td>
<td>a candidate subgraph in response to a pattern query</td>
</tr>
<tr>
<td>( F(G_s, H_q) )</td>
<td>ranking function; takes as input two subgraphs</td>
</tr>
<tr>
<td>( D(v_i, v_i') )</td>
<td>divergence score; takes as input two nodes</td>
</tr>
</tbody>
</table>

\((V^q, E^q)\) and \( W_q\). It offers flexible querying, as the user does not need to specify attribute-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 pattern query.

3. iPaM-RANKING

Given a pattern query (point or range), our task is to return its top-\( k \) best matches. To do so, iPaM needs 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 graph weighted attributes is fundamentally different from the divergence on the graph structure, our proposed ranking function unifies these into one single function.

iPaM Ranking Function.

Suppose \( G_s \) is a candidate subgraph in response to the pattern query \( H_q\), then iPaM’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:

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

Jensen difference is normalized so that a 0 value means a perfect match and 1 value means a perfect non-match. We use the Jensen difference for our divergence function since it has a nice information-theoretic formulation. As mentioned before, iPaM can use any divergence function. The definition of the Jensen difference is as follows:

\[
F_w(v_i, v_i') = \sum_{m=1}^{l} \left[ \frac{w_{im} \times \ln(w_{im}) + w_{im}' \times \ln(w_{im}')} {2} - \frac{w_{im} + w_{im}'} {2} \times \ln\left( \frac{w_{im} + w_{im}'} {2} \right) \right]
\]

Recall that \( w_{im} \) weights the node \( i \)-th node for the \( m \)-th attribute; and \( l \) is the number of weighted attributes.

What constitutes an unmatched node? We define an unmatched node to be an extra node that needs to be added to the solution to satisfy the pattern query’s connectivity requirements. Such a node will have maximal divergence (i.e., 1).

Now that we have discussed the ranking function, we can formally define the problem.

Problem Definition.

Top-\( k \) Pattern Matching in a WAG: Given a WAG, a pattern 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 score \( (F) \) to the pattern query; (ii) any subgraph not present in the set has a larger divergence score with the pattern query than the \( k \)-th subgraph’s overall divergence score.

Problem Hardness.

THEOREM 1. The decision version of the top-1 pattern matching problem is NP-Complete under \( F \).

PROOF. (Sketch) NP-hardness is proved by reducing an instance of the \( \gamma \)-quasi clique [9] problem to the instance of our problem. For brevity, we omit the details of the proof from this version of the paper.

Consequently, the top-\( k \) pattern matching problem is also an NP-complete problem, since the simpler top-1 pattern matching is NP-complete.

4. iPaM-INDEX: A HYBRID INDEX

To enable pattern matching on WAGs, we propose a novel hybrid index structure, called iPaM-Index, and demonstrate how it lends itself to efficient pattern matching. Figure 3 depicts how iPaM works overall, highlighting the key components and processes. The indexing technique is presented next in more detail, while we defer the matching algorithm (iPaM-Match) to Section 5. Given a WAG, iPaM builds and maintains an index structure offline, which is used to speed-up pattern matching during query time and facilitates the matching over both the weighted attributes and the structure of the WAG.

First, a balanced tree index is constructed for the matrix \( W \). Re-indexing is required at query time. The root of iPaM-Index has between \( M \) and \( 2M \) entries. Each leaf node has between \( M/2 \) and \( M \) entries each. The node degrees are \( M \) and \( 2M \) entries.

A leaf node in an iPaM-Index is a WAG vertex with attribute weights \( R \).

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.

Next, at every leaf node (where a leaf node is a vertex in \( G \)), an inverted-list index of immediate neighbors of each graph vertex is maintained. This index enables efficient structural match. We devise the iPaM-Index on \( T \) to be a balanced tree. This tree is built by transforming each row in \( T \) to a multi-dimensional point, where the number of dimensions corresponds to the number of attributes plus a dimension for indexing the degree. Subsequently, this space is indexed. While the core idea is borrowed from the R-tree family [10], which facilitates multi-dimensional spatial searching and the integration into object-relational database management systems, we extend it in several novel ways to enable search on WAGs. This includes: (1) indexing over \( T \) that includes structural and weighted attribute properties; (2) integrating this tree with an inverted-list index to enable fast structural match; and (3) designing a \( \text{getNext()} \) interface over \( T \), which returns the next best candidate vertex of \( G \) that has the least divergence with respect to a WAG query node.

Each leaf node of the iPaM-Index corresponds to a vertex in the input WAG. Each leaf of the iPaM-Index contains an inverted-list that represents the set of nodes that are its immediate neighbors. While we are aware of the existence of more complex structural indexing techniques in the literature, these additional indexing capabilities require expensive pre-processing. It turns out that for our purposes, this structural indexing is sufficient for effective pruning during query time (see Section 6).

**getNext()**: Given any pattern query \( H_q = (V', E') \), for each vertex \( v'_i \in V' \), a \( \text{getNext()} \) call is issued to the iPaM-Index to return the node in \( G \) that has the closest divergence with \( v'_i \) (i.e., nearest neighbor to \( v'_i \)). The inverted-list of the iPaM-Index is subsequently used for the purpose of structural match, after the candidate node for each query node is returned.

**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 node with range specification such as \(< .2 \) on DB and \( > .5 \) on DM, and degree 3 can be translated as ranges \([0 - .2], [.5 - 1.0], \) and \([3 - \text{MaxDegree}] \), 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 as \([0 - 1]\). To enable searching, iPaM-Match starts from the root of the iPaM-Index tree and traverses down the tree. If a current node is non-leaf and 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 an answer.

**Point Query**: Given a point query node \( v' \), it is first transformed to a point in \( l + 1 \)-dimensional space. Like range query, searching for the best matching node of \( v' \) begins at the root of the iPaM-Index 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 iPaM-Index to decide whether or not to search inside the subtree that it indexes. These rectangles can be searched efficiently using MINDIST and MINMAXDIST [14]. MINDIST is the optimistic distance between the point and any object indexed by MBR. Specifically, if \( v'_i \) is inside the MBR, \( \text{MINDIST}(MBR, v'_i) = 0 \). Otherwise, \( \text{MINDIST}(MBR, v'_i) \) 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, \( \text{MINMAXDIST}(MBR, v'_i) \) is the smallest possible upper bound of distances from the point \( v'_i \) to the MBR. The following property is used for effective pruning: \( \text{MINDIST}(MBR, v'_i) \leq \text{NN}(v, v'_i) \leq \text{MINMAXDIST}(v'_i, MBR) \), where \( \text{NN} \) denotes the nearest-neighbor distance.

Given a query node, the tree is progressively searched from the root, using the technique described above. It applies the aforementioned pruning criteria to decide whether to prune or to add an intermediate node to the existing active branches. However, at a leaf node, it computes the actual divergence between the query vertex \( v' \) and the leaf node \( v \) (which corresponds to a vertex in the WAG), compares their divergences with that of smallest divergence seen so far, and updates it if necessary. When the active branch list is empty, the nearest neighbor of \( v' \) is returned.

5. **iPaM-MATCH**

In this section, we discuss an efficient, optimal algorithm (referred to as iPaM in the experiments) for top-\( k \) pattern matching, building upon our iPaM-Index. “Optimality” here is with respect to answer quality, and not necessarily with respect to query processing time. A naive algorithm will enumerate over all possible candidate results before it determines the final top-\( k \) results. This
is prohibitive for even moderately large graphs.

Given any pattern query, the algorithm executes the following tasks: (1) It uses the getNext() interface of iPaM-Index and retrieves the next best candidate node for every query node. (2) It tries to establish all query edges by joining vertices of the WAG that represent the endpoints of a query edge. The inverted-list of the iPaM-Index is used for this purpose. (3) When k answers are not fully computed, the algorithm also judiciously determines whether to expand structurally (i.e., introduce additional nodes to connect the candidate nodes that represent the endpoints of a query edge), or to issue getNext() calls to retrieve the next best candidate node from iPaM-Index. (4) It returns the top-k matches ranked by the increasing order of the overall divergence score. 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-answer may have. The algorithm achieves early-termination when the threshold is not smaller than the score of the k-th best result thus far. In order to accomplish tasks (3) and (4) described above, iPaM-Match exploits the overall divergence score as a threshold and treats it as monotonic.

Keen readers may observe that the high-level intuition of leveraging the threshold bears resemblance to the top-k family of algorithms in [6]. However, our proposed technique requires non-trivial extensions to make such schemes applicable to graphs, and considers divergence measures in the place of scoring functions.

**Range Query:** Algorithm 1 describes the pseudocode of iPaM. Given the query nodes, the task is to gradually retrieve the candidate vertices in a round-robin manner by issuing getNext() calls to the index, and “stitch” the nodes together to recover the graph structure in the query. We define the idea of candidate edge to that end: a candidate edge is a path (or simply an edge) which is a 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 nodes, or else the candidate edge is partially complete. A complete candidate edge does not need to be expanded further, whereas, the partially complete ones may needed to be expanded. For every candidate edge (complete or partially complete), the algorithm keeps track of its current structural divergence.

Note that for the range queries, all candidate nodes that satisfy the specified range constraints of the corresponding query node are equally desirable. Therefore, the overall divergence score primarily counts the additional nodes (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.

At a given point during query processing, when the k-results are not fully computed, iPaM (Algorithm 1) issues additional calls (in a round robin manner) to the index to retrieve the next best candidate node that may be used to form the query pattern. If the iPaM-Index no longer returns a new candidate node with the getNext() call, the algorithm attempts to expand the retrieved candidate nodes 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, as we describe next.

The value of the threshold is the minimum ranking (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 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 node from the iPaM-Index or expands the partially computed answers structurally by an edge, it updates the threshold value.

**Point Query:** The optimal algorithm for point query matching is similar in principle to that for range queries. The main differences here are that (1) ranking of a point query requires stricter matches on the weighted attributes; and (2) during query processing, if top-k answers are not fully computed, then the algorithm has to judiciously decide whether to issue another getNext() call, or to expand structurally. This decision is taken by analyzing the respective threshold values, and choosing the one that results in a smaller threshold. Recall that with every new getNext() call, or structural expansion, we update the threshold, and note that a smaller threshold is better, since it implies that the unseen results will have smaller divergence. We omit detailed discussion for brevity.

**LEMMA 1.** The top-k results returned by the query processing algorithm are optimal in the context of answer quality.

**PROOF.** (Sketch) The above lemma is proved by contradiction. Without loss of generality, let us assume that the above claim is incorrect—i.e., given a query, iPaM does not return the top-k results in the context of answer quality. This means, after iPaM terminates, there is at least one result not returned by iPaM whose ranking score is smaller than the ranking score of the k-th result, returned by iPaM (recall, low ranking score denotes better match). Using the definition of threshold, this denotes, the value of threshold is then smaller than the score of the k-th result, \(^2\) when the algorithm terminates. Note this is a contradiction, because iPaM will terminate only when threshold is at least as large as the score of the k-th result. Hence the proof. \(\square\)

### 6. EXPERIMENTAL EVALUATION

iPaM was 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 [17] of iPaM is implemented and appropriately modified, using Matlab 7.12.0 and the same system configuration.

\(^2\)Recall that threshold contains the best (minimum) score of any unseen candidate result.
6.1 Data

Table 2 lists the real-world data sets used in our experiments. We have three smaller datasets with thousands of nodes and edges; and two large datasets with millions of nodes and edges.

Our smaller datasets include two DBLP co-authorship graphs, one from databases (DB) and another from data mining (DM). Each author constitutes a node in the WAG. The attributes for DB are SIGMOD, VLDB, and ICDE, and that of DM are KDD, ICDM, and SDM. Given a node (author) and an attribute (conference), the weight of that attribute is computed by calculating the percentage of the author’s publications in the corresponding conference.

Our other smaller dataset is an IP communication network from Lawrence Berkeley National Laboratory (LBL) that contains the communication information between different IP addresses in a 24-hour time window. A graph is constructed from this dataset, where each unique IP address corresponds to a node in that graph. An edge between a pair of nodes exists if they have had any communication (with or without scanning) between them. We extract three different attributes (and the respective weight of that on each node) by considering how many of those communications fall within different port number ranges: well-known ports [0, 1023], registered ports [1024, 49151], and ephemeral ports [49152, 65535].

Our large graphs are a Yahoo! IP communication network and an Amazon product co-purchasing network. The Yahoo! graph consists of IP nodes and communication edges, and the attributes (as well their respective weights on each node) are extracted following the same procedure described above for LBL dataset. Our last dataset was collected by crawling the Amazon website. 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 node in the graph, and an edge exists if two products are co-purchased. The attributes are extracted from product ratings (in [1, 5]), and categorized further: excellent (rating 5), good (ratings 3, 4), and bad (ratings 1, 2). The respective weight of a product in an attribute is calculated by computing the normalized ratio of the total number of helpful votes on a given attribute over the total number of votes.

<table>
<thead>
<tr>
<th>Dataset</th>
<th># Nodes</th>
<th># Edges</th>
<th>Sparsity</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP-DM</td>
<td>2104</td>
<td>88,16</td>
<td>0.33</td>
</tr>
<tr>
<td>DBLP-DB</td>
<td>6413</td>
<td>40,493</td>
<td>0.32</td>
</tr>
<tr>
<td>LBL</td>
<td>3055</td>
<td>15,577</td>
<td>0.53</td>
</tr>
<tr>
<td>Amazon</td>
<td>548,100</td>
<td>1,788,725</td>
<td>0.01</td>
</tr>
<tr>
<td>Yahoo!</td>
<td>1,941,878</td>
<td>3,929,607</td>
<td>0.52</td>
</tr>
</tbody>
</table>

Table 2: Datasets used in our experiments. Sparsity is measured by:

\[
\frac{\text{# edges in } W}{\text{# nodes in } V} 
\]

6.2 Setup

To the best of our knowledge, no previous methods can deal with pattern queries on multi-attribute, weighted graphs. However, rather than just comparing against naive baselines, we appropriately modified an existing work [17] that efficiently supports unweighted single-attribute graphs (i.e., each node has a single attribute label and no weights). We primarily report comparative studies between iPAM and this modified competitor, which we call WAG-ray. The performance of other baseline methods is typically even worse. Before proceeding to the results, we will describe WAG-ray.

WAG-ray: We appropriately modify the existing work G-ray [17], which was originally designed for pattern matching 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 pattern query. G-ray uses a scalable algorithm for finding and ranking qualifying subgraphs by finding seeds, expanding neighborhoods around seeds, and then finding bridges to connect the neighborhoods. We refer to the modified G-ray as WAG-ray. Our modification is as follows. We create an augmented graph by adding ‘dummy’ nodes to the WAG; then we connect each original node to all ‘dummy’ nodes with edge weights that correspond to the weights of the attributes on that node. This process introduces \(|V| \times l|d|\) additional edges to the original graph. Next, given a query node 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 match function in WAG-ray needs to be modified to consider approximate match 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 [17].

Additional Baselines: In addition to the experiments reported in detail 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 finding the top-\(k\) answers of a pattern query. Given a pattern query, it first tries to find a set of mapping candidate nodes for every node in the query, with a linear scan over \(W\). It then tries to establish edges between every pair of candidate nodes that represents 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 small graphs, and does not terminate within 30 minutes on large graphs.

With a similar ranking function that ranks a query edge (instead of a query node) relative to an edge in the WAG, we extended the proposed indexing and the query processing approach to accommodate edge-based ranking. In this case, instead of a node, the iPAM-Index index 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 node), and performs the matching accordingly. Our experimental study demonstrates that this method improves the query processing time by less than 30% on average, while it takes significantly longer time to build the iPAM-Index. For brevity, those results are omitted.

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

Queries: We primarily consider four different types of structures in a pattern query—namely, star, path, loop, and clique. For the performance experiments, a point query node is generated from the underlying \(W\) matrix uniformly at random. A range query node is generated from a point query, where we arbitrarily introduce ranges on the specific attribute weights. In either case, the structural pat-
Table 3: Build-time (clock time in sec) of iPaM-Index and WAG-ray over different datasets. Naturally, WAG-ray is faster than iPaM-Index in build time, since the number of links it creates in the augmented graph is linear to the number of nodes (when number of attributes is constant). The last column in the table lists the # of queries for WAG-ray after which iPaM recovers the difference in build time. The smaller values in this column suggests that iPaM-Index is a better query processing algorithm, even though its build time is slightly higher than WAG-ray.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Number of Nodes</th>
<th>Number of Edges</th>
<th>Sparsity</th>
<th>Build Time (iPaM sec)</th>
<th>Build Time (WAG-ray/sec)</th>
<th># of queries after which iPaM recovers the difference in build time with WAG-ray</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP-DM</td>
<td>2104</td>
<td>8816</td>
<td>0.33</td>
<td>6</td>
<td>0.02</td>
<td>1</td>
</tr>
<tr>
<td>DBLP-DB</td>
<td>6413</td>
<td>40493</td>
<td>0.32</td>
<td>18.6</td>
<td>0.06</td>
<td>5</td>
</tr>
<tr>
<td>LBL</td>
<td>3055</td>
<td>15577</td>
<td>0.53</td>
<td>11.4</td>
<td>0.03</td>
<td>2</td>
</tr>
<tr>
<td>Yahoo!</td>
<td>1,941,878</td>
<td>3,929,607</td>
<td>0.52</td>
<td>486</td>
<td>18.91</td>
<td>66</td>
</tr>
<tr>
<td>Amazon</td>
<td>548,000</td>
<td>1,788,725</td>
<td>0.01</td>
<td>327.6</td>
<td>5.23</td>
<td>32</td>
</tr>
</tbody>
</table>

6.3 Build Time Experiments

What is the build time of iPaM-Index on different datasets? Table 3 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 “shallow index”. As evident, iPaM-Index is efficient and scales well with increasing network size. Given similar sparsity value, build times are comparable (between DBLP-DB and DBLP-DM) in iPaM-Index, even when the number of nodes increases 3 times. With a 302 increase in node-size, build-time increases only by 26 times (between DBLP-DB and Yahoo!) in iPaM-Index. Amazon, despite being a large graph with very low sparsity, has a small build time (< 6 minutes). iPaM-Index takes smaller time to build sparse W matrix, as it needs to create a smaller number of intermediate nodes in the iPaM-Index to index the nodes of the WAG.

WAG-ray’s simpler index takes lesser time to build than iPaM-Index. However, as shown in the very last column of Table 3, iPaM is able to recover this time after processing a small number of queries: 1 query for DBLP-DM, and 66 queries for Yahoo!.

6.4 Runtime Experiments

We report the efficiency of iPaM against WAG-ray. First, we report average query processing time (Avg QPT) by varying result size (i.e., k). Next, we report Avg QPT by varying the number of nodes in the query. (Note that since the queries have specific patterns, the number of edges are also known accordingly.) For these runtime experiments, we consider 4 different types of pattern queries. Our query workload consists of 80 queries (20 queries of a particular pattern), and we report the average query processing time. For brevity, we report our results on one small graph (DBLP-DB), and two large graphs (Amazon and Yahoo!). The omitted results on the small graphs are similar to the ones depicted.

Query-processing time as a function of result-size: We compare query-processing time of iPaM and WAG-ray by varying k (result size). iPaM outperforms WAG-ray both for small and large graphs. The results are listed in Figure 4. Clearly, iPaM significantly outperforms WAG-ray in all the cases; Among the two large graphs, the difference in query processing time is more significant in Amazon than in Yahoo!. The very low sparsity of the weighted attribute matrix (W) of Amazon graph makes the computation favorable to iPaM, while WAG-ray was primarily designed for graphs that contain one attribute per node (i.e., very high sparsity of the weighted attribute matrix). Naturally, the efficiency of iPaM is observed to be maximum for Amazon graph.

Query-processing time as a function of number of nodes: We compare query-processing time of iPaM and WAG-ray by varying number of nodes in the query. iPaM outperforms WAG-ray both for small and large graphs. The results are listed in Figure 5. Both iPaM and WAG-ray scale well with the increasing number of nodes. Among the two large graphs in the dataset, Amazon exhibits the best performance with increasing number of nodes. This observation is due to the very low sparsity value of the weighted attribute matrix (W) of Amazon, which helps it scale extremely well with the increasing number of nodes in the query.

7. RELATED WORK

To the best of our knowledge, ours is the first work towards pattern matching on weighted-attributed graphs. In this section, we compare and contrast our work with existing literature on pattern matching in graphs, graph indexing, keyword search on databases, and evaluating inference queries in graphical models.

Pattern Matching in Graphs: Many researchers have studied the problem of pattern matching on large data graphs [17, 4, 25, 16, 12, 7, 13]. What sets our work apart is the fact that our graphs

---

3Recall low sparsity signifies small no of 0’s in the W matrix.
lem here, since the pattern matching on the WAG needs to consider

tensively studied the problem of keyword search over relational
approaches, and evaluation techniques can be found in [8].

different graph matching problems, general and specific solution
attributes. A comprehensive survey describing variations among
is tangentially different due to the existence of weights over node
lem. While we also aim at doing pattern matching, our problem

tance

authors in [25] extend the problem by designing constraints on

rected graph (based on

vides a comparative list between previous works and iPaM. The

outperforms

of nodes. This observation is most prominent for Amazon dataset that has a very low sparsity value of 0.01. This demonstrates the effectiveness of iPaM on the WAGs.

Figure 4: Comparison of query processing time between iPaM and WAG-ray when varying result-size $k$. Number of nodes is set to 5, and the original sparsity of the respective $W$ matrices are used. Both algorithms scale linearly with increasing $k$. iPaM outperforms WAG-ray in all the cases. The maximum difference in query processing is observed for Amazon dataset that has a very small sparsity value of 0.01. This demonstrates the effectiveness of iPaM on the WAGs.

Figure 5: Comparison of query processing time between iPaM and WAG-ray when varying number of nodes. $k$ is set to 5, and the original sparsity of the respective $W$ matrices are used. Unlike WAG-ray, iPaM scales almost linearly with increasing number of nodes. This observation is most prominent for Amazon dataset that has a very low sparsity value of 0.01. iPaM significantly outperforms WAG-ray in all the cases.

have multiple weighted attributes on their nodes. Table 4 pro-

vides a comparative list between previous works and iPaM. The

authors in [17] propose a pattern matching problem over a large di-

rected graph (based on reachability constraints) and a corre-

ponding approximate matching algorithm to select the best set of pat-

terns when an exact match is not possible. Subsequently, the

authors in [25] extend the problem by designing constraints on dis-

tance instead of reachability and design algorithms for that prob-

tem. While we also aim at doing pattern matching, our problem

is tangentially different due to the existence of weights over node

attributes. A comprehensive survey describing variations among
different graph matching problems, general and specific solution
approaches, and evaluation techniques can be found in [8].

<table>
<thead>
<tr>
<th>Graphs without attributes</th>
<th>Tian &amp; Patel (2008)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphs with single attribute</td>
<td>Tong et al. (2007)</td>
</tr>
<tr>
<td></td>
<td>Kahn et al. (2011)</td>
</tr>
<tr>
<td>Graphs with multiple weighted attributes</td>
<td>iPaM</td>
</tr>
</tbody>
</table>

Table 4: Various pattern-matching approaches on graphs. Only iPaM considers the problem of weighted-attribute graph matching.

Keyword Search on Databases: The database literature has ex-
tensively studied the problem of keyword search over relational
databases [1, 11, 2]. Fundamentally we consider a different prob-
lem here, since the pattern matching on the WAG needs to consider

the structural matching of the input graph, which the former body of

works does not need to consider. Therefore, the algorithm sug-
ggested for the former problem do not lend itself to perform pattern
matching on the WAGs.

Graph Indexing: Given a graph database, graph indexing litera-
ture aims at indexing the graphs based on the frequent substructures
present in them [24, 20, 18]. To that end, GraphGrep [15] is a fa-
mous representative of path-based indexing approach. GraphGrep
enumerates all existing paths up to a certain length $l_p$ in a graph
database $G$ and selects them as indexing features. In comparison
to the path-based indexing approach, [19] uses graphs as basic in-
dexing features, which is often referred to as graph-based indexing
approach. Later on, [22] proposes tree-based indexing features and
demonstrates how that can circumvent the disadvantages that are
present in previous two approaches. Observe that, our work is dif-
f erent in principle here as well, as we wish to perform approxi-
mate pattern match in a graph where the nodes contain probability
distributions.

Inference Queries in Graphical Models: Efficiently evaluating
inference queries [5] has been a major research area in the prob-
abilistic reasoning community. Observe that, our problem is dif-
f erent in nature. Unlike the former body of work, the existence of
an edge in our weighted attributed graph is not probabilistic. More-
over, unlike this formed body of work, the nodes of a WAG contains
multiple weighted attributes.

OLAP in Multidimensional graphs: A very recent work pro-
poses OLAP functionalities in multidimensional graphs [21]. The
authors proposes 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 proposed indexing scheme is useful to answer any subgraph query in the weighted attributed graphs. Again, unlike [21], the nodes in our problem contain multiple attribute weights that makes our problem significantly different in principle from existing literature.

8. CONCLUSIONS

We describe Weighted Attribute Graphs (WAGs), which can model a wide range of data arising in diverse applications. We investigate the problem of pattern matching on WAGs. Although, as we prove, finding the optimal match for a given pattern query on a WAG is NP-complete, we introduce iPaM, that can perform efficient and effective pattern matching on WAGs. iPaM has a novel hybrid indexing scheme that incorporates both the weighted attributes and the graph structure. iPaM uses a novel algorithm to efficiently return the best answers to a pattern query. We demonstrate the effectiveness and scalability of iPaM based on extensive experiments on real-world data, exhibiting up to $7 \times$ better query response times. Future work includes the extension of iPaM for time-evolving graphs.

9. REFERENCES


Figure 6: Comparison of query processing time between iPaM and WAG-ray when varying sparsity. $k$ is set to 5, and number of nodes in the query is 5. With increasing sparsity, WAG-ray processes queries with less time, whereas iPaM requires more time to process queries on extremely sparse WAGs. However for both large datasets, iPaM outperforms WAG-ray under any sparsity values.