A New Approach to Isomorphism in Attributed Graphs

Juan Mendivelso [#] and Yoan Pinzón [*]

***Abstract*—Attributed graphs are widely used in many application domains, for example to model social networks. An attributed graph is a graph in which vertices and edges may have types and other attributes. Different query models have been developed to obtain information from attributed graphs. One of the most important is graph pattern matching, which is the problem of finding all the instances of the pattern graph P in the attributed graph G under graph isomorphism. A pattern graph may specify both structural requirements and predicates on attributes of the graph elements. We propose a novel technique that linearizes the pattern graph and matches such linearization against the attributed graph. We derive heuristics to produce a linearization that places selective predicates at the beginning. We implement the algorithm and our results show that our optimizations based on the attributed graph statistics are effective in querying attributed graphs.**

***Keywords*—Social Networks, Semantic Web, Information Retrieval, Databases**

I. Introduction

Graphs are highly interesting data structures due to their considerable expressive power that allows them to represent real-word phenomena in diverse areas [1], [2], [3], [4], [5], [6], [7], [8]. A graph $G(V, E)$ consists of a set of vertices V and a set of edges E, where the edges are ordered pairs of the vertices that represent links between them. A graph is *attributed* if its vertices and/or edges contain *attributes*. For instance, in Figure 1, we show an example of a social network where vertices represent people and photos while edges establish friendship and person-tagged-in-photo relationships.

In recent years, the burgeoning use of graphs has considerably increased the need for efficient graph matching algorithms. The *graph pattern matching* problem consists of finding all the subgraphs in an attributed graph, called the *data graph*, that satisfy the structural requirements and predicates specified by a query graph, called the *pattern graph*. The structural requirements establish a graph isomorphism requirement for the output subgraphs with respect to the pattern graph. Furthermore, the predicates on the node attributes and the edge attributes of the pattern graph further restrict the output subgraphs. For example, considering the social network presented in Figure 1, we may want to retrieve the pairs of friends that are tagged in a photo where one of them is a female. Then, the pattern graph can be defined as in Figure 2(a) and the output obtained is shown in Figure 2(b) and (c).

Pattern matching (also called subgraph matching and subgraph isomorphism) is an NP-Complete problem [9]. One of the first algorithms for solving this problem was proposed by Ullmann in 1976. There are several techniques to improve different aspects of the algorithm, for example by applying search-pruning methods [10], [2]. More recently, a graph linearization technique was developed to solve graph isomorphism [11]; in particular, a *Graph Linearization Algorithm* – GLA to solve the problem was proposed. However, most of these approaches consider graphs without attributes and only match the graph topology. In this paper, we extend the graph linearization technique to solve the graph pattern matching problem in attributed graphs; especially, we use the statistics of the data graph to produce linearizations that work efficiently in practice.

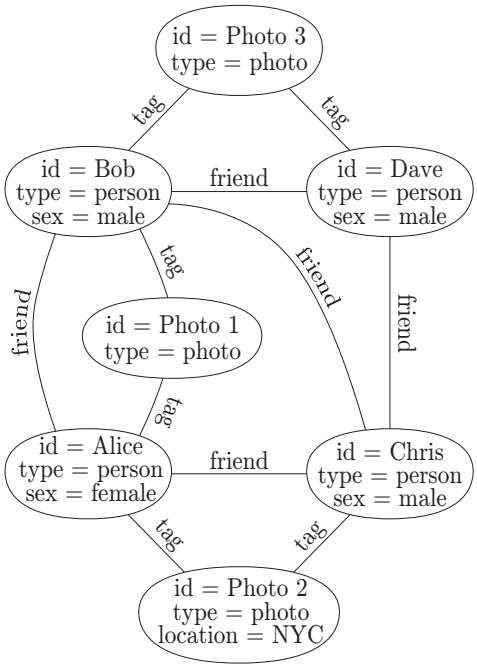

Fig. 1. Example of a social network represented as an attributed graph.

Furthermore, we also use a string matching technique, called parameterized matching, to solve graph pattern matching. Parameterized matching allows to find text substrings that have the same structure of a given pattern string. Our approach consists of linearizing the pattern graph into a walk p so that we can look for the walks in the data graph that parameterized-match p. If such walks also satisfy the predicates, then we report the subgraphs they represent as matches.

The contributions of this work are the following: (1) We propose a novel technique to answer pattern matching queries over an attributed graph (Section II). (2) We develop a 2-approximate length-optimal linearization algorithm that takes

[#] Fundación Universitaria Konrad Lorenz
[*] Universidad Nacional de Colombia

978-1-4799-6717-9/14/$31.00

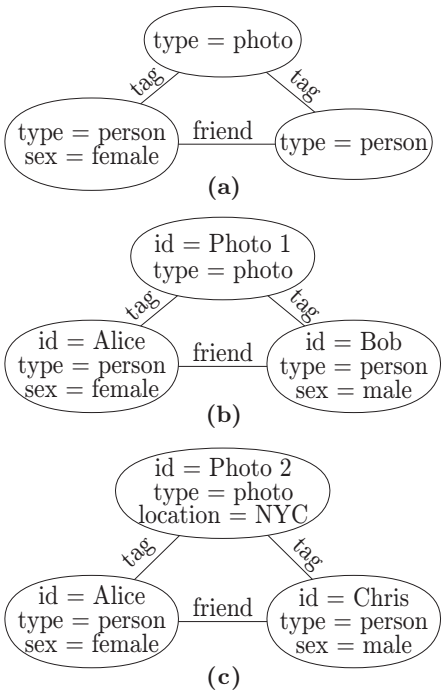

Fig. 2. Example of the graph matching problem for the data graph presented in Figure 1. (a) Pattern graph. (b, c) Output reported.

into account the data graph statistics to produce cost-effective linearizations (Section III). (3) We introduce a matching algorithm that finds the matches of the linearized pattern graph in the data graph (Section IV). (4) We evaluate the proposed techniques experimentally and show that they are effective in reducing query execution time (Section V).

II. Our Approach

We propose to solve the graph pattern matching problem by pattern graph linearization. To perform pattern graph matching on a data graph, our algorithm consists of two phases — linearization phase and matching phase. At the linearization phase, the linearization algorithm transforms the pattern graph into an equivalent parameterized walk; and at the matching phase, our matching algorithm searches for matches of such parameterized walk in the data graph. These matches also constitute matches of the pattern graph. This section gives an overview of the linearization and matching algorithm with examples.

A linearization algorithm traverses all the vertices and edges of the pattern graph and produces a linear sequence of them, which we call the parameterized walk or linearization of the pattern graph. A linearization is a walk of the entire pattern graph thus no information of the pattern graph is lost. A linearization must include all the vertices and edges of the pattern graph at least once. However, as a graph traversal may have to visit some vertices or edges more than once, a linearization may include those vertices and edges more than once. The same vertex/edge that has been visited for multiple times are represented by the same parameter; different vertices/edges are represented by different parameters. Moreover, when we choose different starting vertices and visiting sequences, we can produce many different linearizations for the same pattern graph, which is similar as that the same SQL query at a relational database can have many different physical execution plans.

For example, the pattern graph in Figure 2(a) has six linearizations and we list two of them for demonstration:

```
L1 = P1(vertex,type=photo) P2(edge,type=tag)
       P3(vertex,type=person)
       P4(edge, type=friend)
       P5(vertex,type=person,sex=female)
       P6(edge,type=tag) P1
L2 = P1(vertex,type=person,sex=female)
       P2(edge,type=tag) P3(vertex,type=photo)
       P4(edge,type=tag) P5(vertex,type=person)
       P6(edge,type=friend) P1
```

Note that at both L1 and L2, there are vertices that we have to visit more than once, and each of these vertices is represented by the same parameter.

The matching algorithm searches the linearization of the pattern graph on the data graph to find the walks that satisfy the query, which are associated to subgraphs of the data graph that match the pattern graph. Every match is a bijection from all of the graph elements of the pattern graph to a set of graph elements in the data graph. For example, Figure 2(b) and (c) are the matching outputs of Figure 2(a) where there is a bijection of all graph elements in each output to those of the pattern graph.

Suppose that a linearization has length m; in theory, the matching algorithm must compare it with all length-m walks in the data graph. However, many of these walks are pruned from the very beginning. For example, if we produce linearization L1 to represent the pattern graph at Figure 2(a) and match it with the data graph in Figure 1, the matching algorithm only needs to start from the vertices of type "photo" as the walks starting from other vertices will not match. If we produce linearization L2, we can do even better as we only need to start the search from vertex "Alice". This is analogue to relational databases where several execution plans of the same SQL query produce the same output but may encounter different execution cost. Here different linearized walks of the same pattern graph lead to the same matching outputs but can incur in different costs. In relational databases, the query optimizer uses data statistics and optimization heuristics to decide a better plan with lower expected cost. Here we use the statistics of the data graph to choose the linearization which is likely to be cost effective. For example, between the above two linearizations, L2 has lower matching cost than L1 as the starting vertex of L2 is more selective and it effectively prunes the search space by starting from a single vertex "Alice" only.

In the remainder of the paper, we will describe our linearization and matching algorithms in more details and show how to produce an effective linearization of a pattern graph leading to low matching cost.

III. Linearization Exploiting Graph Statistics

We linearize a pattern graph into a linear sequence of vertices and edges, and we use this linearized sequence as an input to the matching algorithm to find matching subgraphs. This section proposes an algorithm that, besides producing

short linearizations, utilizes graph statistics to further reduce the matching cost.

A. High-Level Ideas

We use an example to illustrate why graph statistics help to produce efficient linearizations. Let us consider a data graph $G(V, E)$ and a the pattern graph $P(V_P, E_P)$. Let *color* be a type for the vertices in $G(V, E)$; we assume that 20% of the vertices in the graph are $green$, 30% are $blue$ and 50% are red. Let us consider a pattern graph where a vertex A must be green and a vertex B must be red. We can start our linearization from either of them, but starting from A is likely to produce a more efficient matching process as A has less matches than B; this helps to prune the search space early. This idea is similar to the query optimizer in a database system, which often pushes the most restrictive conditions down to the evaluation tree of a query plan such that the selective predicates are evaluated early in the execution.

To formalize these ideas, we define the *selectivity* degree, for each graph element ge in the pattern graph. Let $ge.Pred$ denote the set of the predicates associated to ge and let $P(c, G(V, E))$ represent the probability of selecting a graph element from the data graph $G(V, E)$ that satisfies a predicate (or condition) c. Thus, the *selectivity* degree of the graph element ge can be defined as:

$$ge.selectivity = \prod_{c \in ge.Pred} P(c, G(V, E)) \qquad (1)$$

Using the selectivity degree of the pattern graph elements on the data graph, we present a linearization algorithm, called E-GLA, which gives preference to the most selective graph elements.

B. Enhanced-Graph Linearization Algorithm - E-GLA

The *Enhanced-Graph Linearization Algorithm*, *i.e.* the E-GLA algorithm for short, starts the linearization from the most selective vertex (the vertex with the lowest selectivity), and at each of the following step, it greedily chooses the most selective adjacent edge-vertex pair. In particular, the selectivity associated to an edge-vertex pair $e = (u, v)$ and v is calculated as the product of the edge selectivity and the selectivity of the vertex that this edge leads to, *i.e.* $e.selectivity * v.selectivity$. Having more selective edge-vertex pair at the beginning of the linearization helps to prune the search space at an early stage, reducing the total matching cost. However, the length of the linearization is also minimized by E-GLA; specifically, the strategy for that consists of visiting edges that lead to already explored vertices before the edges that lead to unexplored vertices (which are sorted on their selectivity).

1) Pseudocode: The pseudo-code of the E-GLA algorithm is presented in Figure 3. The selectivity of vertices are computed by procedure COMPUTESELECTIVITY() in Figure 4 using the definition of Equation 1. Then E-GLA visits the vertex and its neighbors in a recursive manner by calling the procedure STATSTRAVERSE(), which is presented in Figure 5. The selectivity computations and comparisons of adjacent edge-vertex pairs are processed using procedure FINDBESTNEIGHBOR() in Figure 6.

Algorithm 1: E-GLA **Algorithm**
Input: $P(V_P, E_P)$ **Output**: p

```
1.  for every e ∈ E_P do e.Explored ← false
2.  for every u ∈ V_P do
3.      u.Explored ← false
4.      E ← {(u, v) | v ∈ V_P ∧ (u, v) ∈ E_P}
5.      u.NumUnexploredEdges ← |E|
6.  ComputeSelectivity(P, G)
7.  choose v ∈ V_P with min(v.select)
8.  StatsTraverse(P, v, p)
9.  return p
```

Fig. 3. E-GLA Algorithm.

Algorithm 4: COMPUTESELECTIVITY **Procedure**
Input: $P(V_P, E_P), G(V, E)$

```
1.  for every ge ∈ (V_P ∪ E_P) do
2.      ge.selectivity ← 1
3.      for every c ∈ ge.SemReq do
4.          ge.selectivity ← ge.selectivity * P(c, G(V, E))
```

Fig. 4. COMPUTESELECTIVITY Procedure.

Algorithm 5: STATSTRAVERSE **Procedure**
Input: $P(V_P, E_P), u, p$

```
1.  p.Add(u), u.Explored ← true
2.  for every e = (u, v) ∈ E_P do
3.      if !e.Explored ∧ v.Explored then
4.          p.Add(e), e.Explored ← true, p.Add(v)
5.          Decrease NumUnexploredEdges for u and v
6.          if there are unexplored graph elements do
7.              p.Add(e), p.Add(u)
8.  while there are unexplored edges e = (u, v) ∈ E_P
9.      e = (u, v) ← FindBestNeighbor(u, P)
10.     p.Add(posE, e), e.Explored ← true
11.     Decrease NumUnexploredEdges for u and v
12.     TraverseGraph(P, v, p)
13.     if all graph elements are explored then break
14.     p.Add(e), p.Add(u)
```

Fig. 5. STATSTRAVERSE Procedure.

Algorithm 6: FINDBESTNEIGHBOR **Function**
Input: u, $P(V_P, E_P)$ **Output**: $edge$

```
1.  min = ∞
2.  for every e = (u, v) ∈ E_P do
3.      if !e.Explored then
3.          assocSelect = e.selectivity * v.selectivity
4.          if assocSelect < min then
5.              min = assocSelect, edge = e(u, v)
6.  return edge
```

Fig. 6. FINDBESTNEIGHBOR Function.

2) Length of E-GLA Linearizations: Any linearization algorithm, including length-optimal algorithms, must traverse

each edge of the pattern graph at least once. Given that E-GLA traverses each edge at most twice, in the worst case, the length of the walk query generated by E-GLA is at most 2 times the length of a length-optimal linearization. Thus, E-GLA is asymptotically length-optimal. However, for many graphs, even a length-optimal algorithm has to visit some vertices and edges more than once. Then, the difference between E-GLA and the optimal solution for a given graph is typically much less than the worst-case ratio presented. In comparison to GLA, E-GLA may produce a linearization with longer length but by exploring selective graph elements earlier, fewer explorations are required at the matching phase as the mismatches are likely to be discovered earlier; this observation is supported by the experiments presented in Section V-B.

3) Complexity Analysis: The complexity of E-GLA is established by the length of p. As presented in Section III-B2, p has at most $2|E_P|$ edges and $2|E_P|+1$ vertices. Each insertion takes constant time as it is always done at the end of the list. When a vertex is inserted for the first time, it is necessary to sort its unexplored adjacent vertex-edge pairs on their selectivity (Figure 5, lines $8-9$); this operation takes $O(|V_P|\lg|V_P|)$. Thus, the total time complexity of E-GLA is $O(|V_P|^2\lg|V_P|)$.

IV. Matching Algorithm

The matching algorithm uses the parameterized walk query — linearization of the pattern graph — on the data graph to find the walks that satisfy the query. Denote $G(V_G, E_G)$ as the data graph, $P(V_P, E_P)$ as the pattern graph, and p as the linearization of P with length m. Suppose that q is a length-m path in $G(V_G, E_G)$. We say that q matches p iff there exists a function f such that $f(p_i) = q_i.id$, for all $1 \le i \le m$. Let Σ_p denote the set of distinct parameters in p. If any parameter $X \in \Sigma_p$ occurs at multiple positions i_1, i_2, …, i_k in p, then the corresponding positions in q must refer to the same graph element in $G(V, E)$, *i.e.* $f(X) = q_{i_1}.id = \ldots = q_{i_k}.id$. Furthermore, $f(X)$ must satisfy the predicates of X and no other parameter $Y \in \Sigma_p$ has a mapping $f(Y) = f(X)$. In other words, a graph element can map to at most one parameter in the linearization. Every walk q that matches p represents a subgraph in $G(V_G, E_G)$ that matches the pattern graph P as the matching conditions address both the structural requirements and attribute predicates of the pattern graph. This section presents a pattern graph matching algorithm, called PMG, that finds all the matches of the pattern graph in the data graph.

A. The PMG Algorithm

1) Main Ideas: To find all the matches of a pattern graph in a data graph, PMG performs two tasks: (1) Decide if a given length-m walk of the data graph matches the linearization p of the pattern graph, and (2) Explore all the length-m paths in the data graph.

The first task of PMG is to determine whether $p = p_{1\ldots m}$ and a length-m path in $G(V, E)$ match. That is, comparing p_i and q_i for $i = 1, \ldots, m$. Suppose that $p_i = X$ and q_i corresponds to a graph element ge in $G(V_G, E_G)$. Two cases are considered to match p_i and q_i:

(i) The position i is the first occurrence of X in p. Then, we must verify that ge satisfies the predicates specified by X and that $f(Y) \neq ge.id$ for all $Y \in \Sigma_p$. If so, it is a match, and we establish a new mapping $f(X) = ge.id$; otherwise, it does not match.

(ii) The position i is not the first occurrence of X. In this case, if ge is the same graph element that was formerly assigned to X, *i.e.* $f(X) = ge.id$, it is match; otherwise, it does not match.

If the verification of either case (i) or (ii) was successful, we check the next position on if p_{i+1} matches q_{i+1}; otherwise, we can conclude that p and q do not match without further comparisons. If p_i and q_i satisfy these conditions for all $1 \le i \le m$ we conclude that p and q matches.

To perform the second task, PMG traverses the graph in a breadth-first-search fashion starting from each vertex in general. (If an index is available to restrict the set of starting vertices, then matching starts at a subset of the graph vertices). Suppose that we start from vertex v_1 in the data graph. We first compare p_1 and v_1. If they match, we compare p_2 with each one of the adjacent edges e_j of v_1. For the edges e_j that match p_2, we try to extend the search by comparing p_3 with the vertices that they lead to. The process continues until p_m is compared against the adjacent elements of $f(p_{m-1})$ for the remaining matching walks, if there is any.

Note that at each level i of the BFS tree, we compare p_i with the adjacent elements of $f(p_{i-1})$ to extend the length-$(i-1)$ walks that match $p_{1\ldots i-1}$. If there is a mismatch between p_i and the corresponding graph element in the data graph, we stop the exploration along this branch of the BFS tree. Since the mapping functions of the different branches of the BFS tree are associated with different mapping functions, we need to record a mapping for each active branch. Whenever a length-m walk that matches $p_{1\ldots m}$ is found, its corresponding mapping function f is reported as a match.

2) Pseudocode: Figure 7 presents the pseudo-code of the PMG algorithm, which takes the linearized pattern graph and the data graph as inputs and produces all the matched walks in the data graph as output. The variable ν, called the *valuation*, is a mapping of parameters at the linearized pattern graph and graph elements in the data graph for a given traversed walk; thus, a valuation represents one matched walk to p. The set $\mathcal{R}$ is the set of all the valuations, which is the output set. The algorithm starts the BFS exploration from every vertex v in the data graph by calling the recursive procedure ExtendMatching() for v. This procedure attempts to extend the current walk by determining whether the next parameter on the linearization and a given graph element in the data graph can be associated; this is evaluated using the function Match() (see Figure 9).

3) Complexity Analysis: The time complexity of PMG is given by the number of executions of the recursive procedure ExtendMatching where each one requires constant time. This number is equal to the number of vertices and edges in the BFS search trees. In the worst case, $P(V_P, E_P)$ has no types and attributes to apply pruning of the tree. Let n denote the number of vertices in the data graph and d the maximum number of edges associated to a vertex. The BFS tree rooted at one of the vertices of the data graph has $\lceil m/2 \rceil$ levels of

Algorithm 7: PMG **Algorithm**

Input: $p = p_{1..m}, G(V, E)$ **Output**: $\mathcal{R}$

1. $\nu = \{\}, \mathcal{R} = \{\}$
2. **for every** $v \in V$ **do**
3. $\quad ExtendMatching(v, p, 1, \nu, \mathcal{R})$
4. **return** $\mathcal{R}$

Fig. 7. PMG Algorithm.

Algorithm 8: EXTENDMATCHING **Procedure**

Input: ge, $p = p_{1..m}$, pos, ν, $\mathcal{R}$

1. **if** $Match(ge, p_{pos}, \nu) = true$ **then**
2. $\quad$ **if** $i = m$ **do**
3. $\quad\quad \mathcal{R}.Add(copyOf(\nu))$
4. $\quad$ **else if** $i < m$ **do**
5. $\quad\quad$ **for each** $ch \in childrenOf(ge)$ **do**
6. $\quad\quad\quad ExtendMatching(ch, p, pos + 1, copyOf(\nu), \mathcal{R})$

Fig. 8. EXTENDMATCHING Procedure.

Algorithm 9: MATCH **Function**

Input: ge, p_{pos}, ν **Output**: $true/false$

1. **if** $p_{pos}.Predicates(ge)$ **and**
2. $\quad \nu.IsCompatible(\{p_{pos}/ge.ID)\})$ **then**
3. $\quad\quad \nu = \nu \cup \{p_{pos}/ge.ID\}$
4. $\quad\quad$ **return** $true$
5. **else**
6. $\quad\quad$ **return** $false$

Fig. 9. MATCH Function.

vertices and $\lfloor m/2 \rfloor$ levels of edges; we just consider the levels of vertices. The root has 1 vertex and the second level has d vertices. Each of these d vertices is associated to $d-1$ vertices in the third level (as the edges that lead to vertices in upper levels of the tree are not considered); thus, the third level has $d(d-1)$ vertices. In general, level i of the tree has $\prod_{j=0}^{i-2}(d-j)$ vertices. Thus, the total number of vertices of a BFS tree is:

$$1 + \sum_{i=2}^{\lceil m/2 \rceil} \prod_{j=0}^{i-2}(d-j) \approx O(d^{\lceil m/2 \rceil - 1})$$

Since a linearized walk alternates between vertices and edges while starting and ending at a vertex, m is odd. Thus, $O(d^{\lceil m/2 \rceil - 1}) = O(d^{\lfloor m/2 \rfloor})$. As we have a BFS tree starting at each vertex in the data graph, the total number of vertices visited, and hence the time complexity of PMG, is $O(nd^{\lfloor m/2 \rfloor})$.

V. EXPERIMENTAL RESULTS

We evaluate the performance of our approach experimentally to assess the benefits of the proposed techniques. We show the effectiveness of using graph statistics to optimize graph linearizations under a variety of data graph sizes and query graph patterns.

A. Experimental Setup

Implementation. We implement GLA [11], E-GLA, selectivity estimation, graph statistics gathering, and matching algorithms in C#.

Query pattern graphs. We employ pattern graphs with different topologies and sizes. We use complete graphs, path graphs, cyclic graphs and star graphs. We add predicates to graph vertices. Specifically, the predicates include the following: (i) the type of the vertex; and (ii) an interval of possible values for the attribute.

Data graphs. We generate attributed graphs $G(V, E)$ using the *Recursive Matrix* (RMAT) model [12] that generates scale-free graphs, similar to the types of graphs used in many applications. We use data graphs with different sizes: $|V| =$ 1024, 16384, 131072, 524288, 1048576. The number of edges of each graph is $|E| = 5 \times |V|$, representing sparse graphs. For each graph, vertex types are drawn from a set Σ_t where $|\Sigma_t| = 1\% \times |V|$. Vertex type follows a zipf distribution which models the popularity of vertex types. We also associate with each vertex an attribute with a value sampled uniformly from a set Σ_a where $|\Sigma_a| = 100$.

Hardware. We perform the experiments on a commodity server with 3.30 GHz Intel Xeon X5680 CPU with 24 GB RAM running Windows Server 2008R2. **Metrics.** Our main performance metric is the query response time. We report the number of graph element comparisons as this is the dominant factor in time complexity. For reference, on our sever, 400000 comparisons are performed per second. We also report the length of pattern query linearization.

We use data graphs with predicates (types and attributes). We evaluate how exploiting graph statistics is useful to produce optimized linearizations and improve query response time.

B. Optimized Linearization Using Graph Statistics

We linearize all the pattern graphs using GLA which does not consider graph statistics [11], and using E-GLA which does utilize them. We compare the cost of matching a query on a data graph using the two linearization algorithms.

Figure 10(a) compares the length of the linearization by both algorithms for the different queries. Notice that E-GLA linearizations are equal or slightly longer than the ones of GLA. For instance, the linearization of the *Path 6* pattern graph is longer for E-GLA; however, this is a good trade-off as we show next.

Figure 10(b) shows the number of comparisons required for the matching process using GLA and E-GLA for pattern graphs of different topology and different sizes on the data graph with $|V| = 524288$. Notice that the number of comparisons required in the matching process is significantly lower when using the linearization produced by E-GLA. This is because such linearization starts with the vertices associated with the most selective predicates, the rarest to find in the data graph. Thus, the search space is considerably pruned. In particular, although E-GLA may produce optimized linearizations that are longer than the non-optimized linearizations, the query processing time (matching time) is much smaller with the optimized linearization.

The benefits of E-GLA linearization are even more significant in bigger data graphs. For example, notice that the greatest

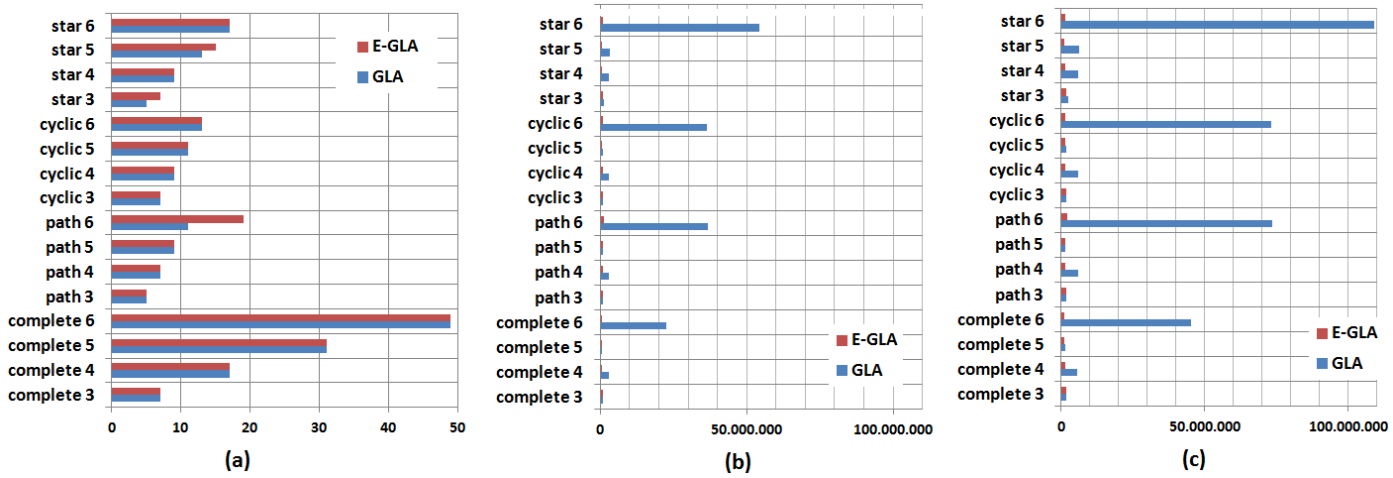

Fig. 10. Experimental comparison of the matching process using GLA and E-GLA linearizations. The pattern graphs are graphs with different topology (complete, path, cyclic and star graphs) and of different sizes ($|V_P| = 3, \ldots, 6$). (a) Linearization length on the same data graph. (b) Number of comparisons using data graph with $|V| = 524288$. (c) Number of comparisons using data graph with $|V| = 1048576$.

number of comparisons for the data graph with $|V| = 524288$ in Figure 10(b) is almost 60 million using GLA; for the data graph with $|V| = 1048576$ in Figure 10(c), the greatest number of comparisons is more that 100 million comparisons. In contrast, with E-GLA the number of comparisons is only 0.6 and 1.5 million, respectively.

VI. Conclusions

This paper proposes a new approach to solve graph pattern matching over attributed graphs: Query graphs are linearized into a walk with parameters. Using selectivity estimation and graph statistics, an enhanced linearization algorithm optimizes the order of predicates in the linearized walk, placing selective predicates early in the linearization. We derive a matching algorithm to match the linearized walk by traversing the attributed graph. We present linearization and matching algorithms, and analyse their time complexity. We implement these algorithms and evaluate them over a set of data graphs and pattern graphs of different sizes and structures. The experimental results show that our techniques perform well, and in particular the enhanced linearization algorithm, which exploits the graph statistics, reducing query response times.

References

[1] N. Shadbolt, W. Hall, and T. Berners-Lee, "The semantic web revisited," *Intelligent Systems*, vol. 21, no. 3, pp. 96–101, 2006.

[2] L. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub) graph isomorphism algorithm for matching large graphs." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 10, pp. 1367–1372, 2004.

[3] L. Cordella and M. Vento, "Symbol recognition in documents: a collection of techniques?" *J. on Doc. Analysis and Recognition*, vol. 3, no. 2, pp. 73–88, 2000.

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

[5] B. Gallagher, "Matching structure and semantics: A survey on graph-based pattern matching," in *AAAI*, 2006.

[6] P. Zhao and J. Han, "On graph query optimization in large networks," in *VLDB*, vol. 3, September 2010, pp. 340–351.

[7] C. Branden and J. Tooze, *Introduction to protein structure*, Garland, Ed. Garland New York, 1998, vol. 17.

[8] F. Eichinger, K. Bohm, K.hm, and M. Huber, "Mining edge-weighted call graphs to localise software bugs," in *KDD*, 2008.

[9] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*. WH Freeman & Co., San Francisco, 1979.

[10] B. Falkenhainer, K. Forbus, and D. Gentner, "The structure-mapping engine: Algorithm and examples," *Artificial intelligence*, vol. 41, no. 1, pp. 1–63, 1989.

[11] J. Mendivelso, S. Kim, S. Elnikety, Y. He, S.-w. Hwang, and Y. Pinzón, "Solving graph isomorphism using parameterized matching," in *String Processing and Information Retrieval*. Springer, 2013, pp. 230–242.

[12] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," in *SDM*, 2004.