



UNIVERSIDAD NACIONAL DE COLOMBIA

The Graph Pattern Matching Problem through Parameterized Matching

Juan Carlos Mendivelso Moreno

Universidad Nacional de Colombia
School of Engineering, Department of Computer Science and Industrial Engineering
Bogotá, Colombia
2015

The Graph Pattern Matching Problem through Parameterized Matching

Juan Carlos Mendivelso Moreno

Dissertation presented as partial requirement to grant the title of:
Doctor of Philosophy – Computer Science

Advisors:

Yoan Pinzón, PhD and Sameh Elnikety, PhD

Research Area:

Theoretical Computer Science

Research Group:

ALGOS-UN

Universidad Nacional de Colombia

School of Engineering, Department of Computer Science and Industrial Engineering

Bogotá, Colombia

2015

(Dedication)

In the loving memory of my dear grandpa Martín Mendivelso. I thank him very much for his love and his scientific genes.

Also, I could not have made this without my dear mother. Her infinite love and support is what has brought me this far. I will never be able to give back to her all the incredible things she has done for me, but I will try. My love for her is infinitely deep.

I am so lucky to have another wonderful mother: Libby. We have had a really special connection since I was born. I thank her for her unconditional love and support. My love for her has no boundaries.

I love my Dad infinitely. Feeling his love and that he is proud of me fills my heart. I appreciate that my dear brother Diego is also proud of me. I deeply love him.

I thank Anna Sylvia for her incredible love and support since the day I was born. Also, I thank Fucci, my baby, for teaching me what love, beauty, tenderness, loyalty and joy are. I love Isaurah, Starr, Marujah, Johanna, Linna, Matthew and all my family too.

It is a true blessing to have been friends with Fábiko for more than 13 years already. His friendship has pulled me through more than once. I truly love him and appreciate him.

I will always be grateful with Mónica Sierra Riveros for what she did for me, no matter what. I will love her forever. I cannot thank Patricia Muethé enough. She loved me back to life. She has been there when I have needed her most. I love her.

I thank Patricia Torres and Marcela Pabón for their wonderful support.

I also thank John Paul (X2), Myrtha, Andrew and Helen, for their friendship. I appreciate my dear friends at the E-Theater. Their impact in my life has been enormous. I have done some of the things I love the most with them. And all in a nurturing environment.

Acknowledgements

First of all, I want to thank Yoan Pinzón. Besides being an excellent advisor and professor, he is a wonderful person. I have the deepest respect and gratitude for him. I thank him for his incredible guidance and support throughout my MSc and my PhD. And of course, I thank him for introducing me into the research world and helping me launch my career.

Also, I want to thank my co-advisor Sameh Elnikety. I thank him very much for everything. Working with him has taken my career to another level. I thank him for his guidance, insight, patience, support, generosity and kindness. He is a great person; I feel fortunate to have him as the co-advisor of this thesis and to keep learning from him.

I want to thank all the co-authors throughout my career, including Seung-won Hwang, Yuxiong He, Inbok Lee, Xing Xie, Luis F. Niño, Juana Córdoba, David Becerra, Camilo Pino and Sunghwan Kim. I have learned much from working with them and I appreciate all their incredible contributions.

I truly appreciate the wonderful work of the professors at the Computer Science Department of the National University of Colombia. I appreciate the great quality of the education they provide. They definitely do much for the students' lives and for the development of the country.

I also thank the marvellous research groups I have been a part of for long time: ALGOS-UN and LISI.

Furthermore, I want to thank everyone at Fundación Universitaria Konrad Lorenz. I have felt really well working with them. Also, I appreciate your support in the final stage of this thesis.

I deeply thank Microsoft Research for the opportunity of doing a research internship at the Redmond labs. I learned a lot there and also met very nice people.

I am grateful with the National University of Colombia for all my undergraduate and graduate education. I owe a lot of what I am to this wonderful university that has allowed me to grow as a computer scientist, researcher and professor.

Finally, I want to thank different entities of the National University of Colombia, such as the Research Direction of Bogotá, the Postgraduate Curricular Direction and the Vice-dean of Research, for the funding provided for the academic trips. Also, I want to thank the Academic Direction of Bogotá for the Assistant Professor Scholarship throughout my MSc and PhD.

Abstract

We propose a new approach to solve graph isomorphism using parameterized matching. Parameterized matching is a string matching problem where two strings parameterized-match if there exists a bijective function, on the symbols of the alphabet, that maps one of the strings into the other. Given that parameterized matching is defined for linear structures, we define the concept of *graph linearization* to represent the topology of a graph as a walk on it. Then, our approach to determine whether two graphs are isomorphic consists of determining whether there exists a walk in one of the graphs that parameterized-matches a linearization of the other graph.

Our solution has two main steps: linearization and matching. We develop an efficient linearization algorithm, that generates short linearizations with an approximation guarantee, and develop a graph matching algorithm. We show that this solution also works for subgraph isomorphism, which is the problem of determining whether an input graph H is isomorphic to a subgraph of another input graph G . We evaluate our approach experimentally on graphs of different types and sizes, and compare to the performance of VF2, which is a prominent algorithm for graph isomorphism. Our empirical measurements show that graph linearization finds a matching graph faster than VF2 in many cases, especially in Miyazaki-constructed graphs which are known to be one of the hardest cases for graph isomorphism algorithms.

We extend this approach to query attributed graphs. An attributed graph is a graph data structure, in which nodes and edges may have identifiers, types and other attributes. Attributed graphs are used in many application domains, for example to model social networks in which nodes represent people, photos, and postings and edges represent friendship, person-tagged-in-photo and mentioned-in-post relationships. Queries are used to extract information from such graphs. Several graph queries are expressed as graph pattern matching, which is the problem of finding all instances of pattern match query P in a larger attributed graph G . A pattern match query may specify both a graph structure and predicates on the attributes of the graph elements. Clearly, this problem is associated to subgraph isomorphism.

Furthermore, we define a more general class of graph queries called *generalized pattern queries* on attributed multigraphs. The goal of this class is to find paths and subgraphs that satisfy query reachability and predicates. The query language is expressive: It allows (i) using regular expression operators (e.g., Kleene star and union); (ii) specifying structural predicates on graph nodes and edges; and (iii) using attribute predicates on nodes and edges. Pattern match queries, reachability queries, their combination, and even more queries can be expressed through generalized pattern queries. We use our approach to solve this new type of queries.

The proposed technique has two phases. First, the query is linearized, i.e., represented as a graph walk that covers all nodes and edges. There are several linearizations for a given query; we derive

heuristics to produce a good linearization that is short and places selective predicates early in the linearization. Second, we search for a bijective function that maps each element of the query to an element of the attributed multigraph that satisfies the reachability requirements and the predicates. Specifically, we develop an algorithm that matches the linearization by traversing the attributed graph in a manner similar to a breadth first traversal constrained by the linearization. We evaluate our solution experimentally using a real graph (the DBLP citation network) to assess its practicality and efficiency. Our results show that our techniques and optimizations are effective in querying attributed graphs, offering several factors of reduction in query response time when graph statistics are utilized.

Keywords: parameterized matching, graph theory, graph algorithms, graph matching, pattern matching, graph isomorphism, subgraph isomorphism, attributed graphs, graph queries, social networks.

Resumen

En esta tesis se propone un nuevo enfoque de solución para resolver el problema de isomorfismo de grafos usando búsqueda parametrizada. La búsqueda parametrizada es un problema de búsqueda de cadenas de texto en el cual dos cadenas coinciden si existe una biyección que mapee los símbolos de una cadena en los símbolos de la otra. Dado que la búsqueda parametrizada está definida para estructuras lineales, se define el concepto de linearización de grafos para representar la topología de un grafo como un camino sobre este. Entonces, la solución para determinar si dos grafos son isomorfos consiste en determinar si existe un camino en uno de los grafos que haga coincidencia parametrizada con la linearización del otro grafo.

La solución propuesta tiene dos pasos: linearización y búsqueda. Se presenta un algoritmo eficiente que genera linearizaciones aproximadamente óptimas en longitud, y un algoritmo de búsqueda. Se demuestra que esta solución también resuelve el problema de isomorfismo de subgrafos, en el cual se determina si un grafo H es isomorfo a un subgrafo de otro grafo G . Se evaluó experimentalmente la solución con grafos de diferentes tipos y tamaños. Se comparó su desempeño con el de VF2, el cual es un algoritmo competitivo de isomorfismo de grafos. Los resultados experimentales muestran que la solución propuesta es más eficiente que VF2 en varios casos, en especial en grafos Miyazaki, los cuales se caracterizan por constituir uno de los casos más difíciles para los algoritmos de isomorfismo de grafos.

Este enfoque de solución se extiende para resolver consultas sobre grafos semánticos. Un grafo semántico es un grafo en el cual los nodos y arcos pueden tener identificadores, tipos y otros

atributos. Estos grafos tienen aplicaciones importantes en diversas áreas, como por ejemplo para modelar redes sociales en las que los nodos representan personas, fotos y publicaciones y los arcos representan relaciones de amistad, etiquetado y mención. Se usan consultas para extraer información de estos grafos. Varias de estas consultas se expresan como búsqueda de patrones, la cual consiste en encontrar las coincidencias del grafo patrón P en un grafo semántico G . El grafo patrón especifica tanto la estructura de las coincidencias a encontrar, como los predicados sobre los atributos que deben cumplir los nodos y los arcos de las mismas. Claramente, este problema está asociado al isomorfismo de subgrafos.

Además, se define un tipo de consultas más general sobre grafos semánticos. Estos nuevos patrones se denominan *grafos patrón generalizados*. El objetivo de estos es encontrar caminos y subgrafos que satisfagan ciertos requisitos semánticos, de estructura y de alcance. Estos patrones son expresivos, pues permiten (i) usar operadores de expresiones regulares (e.g., la estrella de Kleene y la unión); (ii) especificar predicados estructurales en los nodos y arcos; y (iii) evaluar predicados sobre los atributos de los nodos y arcos. Los patrones grafo tradicionales, las consultas de alcance, la combinación de estos y más se pueden representar a través de grafos patrón generalizados. Se usa el enfoque de solución propuesto para resolver los grafos patrón generalizados.

La solución tiene dos fases. Primero, el patrón es linearizado, i.e., representado como un camino que incluye todos sus nodos y arcos. Hay muchas linearizaciones para un patrón dado; se proponen heurísticas para producir linearizaciones cortas que ubican los predicados selectivos al comienzo. Segundo, se busca una función biyectiva que mapee cada nodo en el patrón a un nodo en el grafo semántico que satisfaga los requisitos de alcance y los predicados. Específicamente, se propone un algoritmo de búsqueda que recorre el grafo semántico siguiendo una búsqueda en amplitud restringida por la linearización. La solución se evaluó experimentalmente usando un grafo semántico real (la red de citas DBLP) para evaluar su practicidad y eficiencia. Los resultados experimentales muestran que las técnicas y optimizaciones propuestas son efectivas en consultar grafos semánticos, ofreciendo un alto factor de reducción en el tiempo de ejecución cuando se utilizan las estadísticas del grafo semántico.

Palabras clave: búsqueda parametrizada, teoría de grafos, algoritmos de grafos, búsquedas en grafos, búsqueda de patrones, isomorfismo de grafos, isomorfismo de subgrafos, grafos semánticos, redes sociales.

Contents

| | |
|---|------------|
| Acknowledgements | VII |
| Abstract | IX |
| List of Figures | XVI |
| List of Tables | 1 |
| 1. Introduction | 2 |
| 1.1. Graphs: Concepts and Applications | 2 |
| 1.2. Graph Matching Problems | 4 |
| 1.3. Applications of Graph Matching | 8 |
| 1.4. Parameterized Matching | 9 |
| 1.5. Our Contributions | 11 |
| 2. Related Work | 13 |
| 2.1. Solutions for Graph and Subgraph Isomorphism | 13 |
| 2.1.1. Ullmann’s Algorithm | 13 |
| 2.1.2. The VF2 Algorithm | 15 |
| 2.2. Queries on Attributed Graphs | 18 |
| 2.2.1. Reachability Queries | 18 |
| 2.2.2. Pattern Match Queries | 21 |
| 2.2.3. Pattern Queries | 21 |
| 2.3. Parameterized Matching | 22 |
| 2.3.1. Definition of the Basic Problems | 22 |
| 2.3.2. Solutions | 24 |
| 2.3.3. Extensions | 30 |
| 2.3.4. Applications | 33 |
| I. Graph Isomorphism through Parameterized Matching | 36 |
| 3. Our Approach: Graph Linearization | 37 |
| 3.1. Definition of Graph Linearization | 37 |

| | | |
|------------|--|-----------|
| 3.2. | Characteristics and Algorithms for Graph Linearization | 39 |
| 3.3. | Graph Linearization Algorithm - GLA | 40 |
| 3.3.1. | Key Ideas | 40 |
| 3.3.2. | Algorithm | 41 |
| 3.3.3. | Correctness Proof | 43 |
| 3.3.4. | Length of GLA Linearization | 45 |
| 3.3.5. | Empirical Comparison on the Length of Different Linearization Algorithms | 46 |
| 3.3.6. | Complexity Analysis | 47 |
| 4. | Algorithm for Graph Isomorphism | 48 |
| 4.1. | Key Ideas | 48 |
| 4.2. | Algorithm | 49 |
| 4.3. | Correctness Proof | 50 |
| 4.4. | Complexity Analysis | 52 |
| 4.5. | Experimental Evaluation | 53 |
| 4.5.1. | Benchmark Graphs | 54 |
| 4.5.2. | Synthetic Graphs | 55 |
| 4.6. | PMG-SI: Solution for Subgraph Isomorphism | 57 |
| 4.6.1. | Algorithm | 58 |
| 4.6.2. | Experimental Evaluation | 58 |
| II. | Queries on Attributed Graphs Solved through Parameterized Matching | 63 |
| 5. | Generalized Pattern Queries | 64 |
| 5.1. | Graph Model | 64 |
| 5.2. | Query Model | 65 |
| 5.2.1. | Prerequisites | 65 |
| 5.2.2. | Definition of Generalized Pattern Queries | 67 |
| 5.2.3. | Example | 68 |
| 5.2.4. | Discussion | 68 |
| 6. | Linearization on Generalized Pattern Queries | 71 |
| 6.1. | Query Linearization | 71 |
| 6.2. | Enhanced Graph Linearization Algorithm — E-GLA | 75 |
| 6.2.1. | Baseline: GLA for Length-Optimality | 75 |
| 6.2.2. | Key Ideas | 76 |
| 6.2.3. | Algorithm | 77 |
| 6.2.4. | Correctness Proof | 78 |

| | |
|---|-----------|
| 6.2.5. Length of E-GLA Linearization | 79 |
| 6.2.6. Complexity Analysis | 80 |
| 7. Solution of Generalized Pattern Queries | 81 |
| 7.1. Key Ideas | 81 |
| 7.2. Algorithm | 82 |
| 7.3. Correctness Proof | 86 |
| 7.4. Complexity Analysis | 87 |
| 7.5. Experimental Evaluation | 89 |
| 7.5.1. Experimental Setup | 89 |
| 7.5.2. Queries on the Complete DBLP Graph | 90 |
| 7.5.3. Varying Graph and Query Sizes | 92 |
| 7.5.4. Efficiency of E-GLA | 94 |
| 8. Conclusions | 96 |
| Bibliography | 99 |

List of Figures

| | | |
|------|--|----|
| 1-1. | Examples of graphs and graph isomorphism | 3 |
| 1-2. | Example of an attributed multigraph | 4 |
| 1-3. | Example of graph isomorphism | 5 |
| 1-4. | Example of a pattern match query | 7 |
| 1-5. | Example of a parameterized-match between strings | 10 |
| 1-6. | Illustration of the structure of parameterized-matching strings | 11 |
| 2-1. | Procedure <i>prev</i> as an aid for parameterized matching | 26 |
| 2-2. | Example of a parameterized-suffix tree | 27 |
| 2-3. | Concept map of the parameterized matching algorithms | 30 |
| 2-4. | Example of the application of parameterized matching in software maintenance | 34 |
| 3-1. | Pseudocode for the GLA algorithm | 42 |
| 3-2. | Pseudocode for the TRAVERSEGRAPH() procedure | 43 |
| 3-3. | Examples of graphs with different topology | 46 |
| 4-1. | Pseudocode for the PMG algorithm | 50 |
| 4-2. | Pseudocode for the EXTENDMATCH() function | 51 |
| 4-3. | Response time of GLA and VF2 on the benchmark graphs | 54 |
| 4-4. | Response time of GLA and VF2 on sparse and dense synthetic graphs | 57 |
| 4-5. | Pseudocode for the PMG-SI algorithm | 58 |
| 4-6. | Pseudocode for the EXTENDMATCHSI() procedure | 59 |
| 4-7. | Experimental results of subgraph isomorphism for complete graphs | 61 |
| 4-8. | Experimental results of subgraph isomorphism for path graphs | 61 |
| 4-9. | Experimental results of subgraph isomorphism on graphs of different sizes | 61 |
| 5-1. | Example of an attributed multigraph | 65 |
| 5-2. | Example of a generalized pattern query | 69 |
| 5-3. | Example of a reachability query expressed as a generalized pattern query | 70 |
| 5-4. | Example of a reachability query with intermediate nodes of interest | 70 |
| 6-3. | Pseudocode for the E-GLA algorithm | 77 |
| 6-4. | Pseudocode for the STATSTRAVERSE() procedure | 78 |
| 7-1. | Pseudocode for the GPQM algorithm | 83 |

| | | |
|-------------|--|----|
| 7-2. | Pseudocode for the <code>PROCESSNODE()</code> procedure | 84 |
| 7-3. | Pseudocode for the <code>FINDREACHABLENODES()</code> function | 84 |
| 7-4. | Example of a DFS search tree traversed by GPQM | 85 |
| 7-5. | Example of the DFA for a (u, v, ρ) -reachability requirement | 86 |
| 7-6. | Experimental results of GPQM on the complete DBLP graph | 93 |
| 7-7. | Experimental results of GPQM for different graph and query sizes | 94 |
| 7-8. | Experimental evaluation of the effectiveness of E-GLA linearizations | 95 |

List de Tables

| | | |
|-------------|---|----|
| 2-1. | Worst-case complexity of solutions for reachability queries | 19 |
| 3-1. | Comparison of the output length of different linearization algorithms | 47 |
| 4-1. | Number of winning cases of GLA and VF2 on the benchmark graphs | 55 |
| 4-2. | Ratio of short-running cases of GLA and VF2 on the benchmark graphs | 56 |
| 5-1. | Reversal of reachability expressions | 67 |
| 6-1. | Example of the calculation of node selectivity | 78 |
| 7-1. | Examples of generalized pattern queries on the DBLP graph | 91 |

1. Introduction

Graphs are interesting data structures due to their expressive power that allows them to represent real-world phenomena in diverse areas. Graphs' expressive power lies in their ability to represent different kinds of concepts and the relationships among such concepts. Nowadays, graph-based models are found in different domains where both concepts and relationships contain rich and diverse information expressed by types and attributes. In some applications, graphs contain millions of concepts and it is required to support queries efficiently. However, the growing size of graph databases and the richness and the variety of their data have made it difficult to resolve this problem in reasonable time. Given that existing solutions are not able to deal with the current demands, it is quite pertinent to keep exploring new solutions from different approaches in order to efficiently query large graphs. In this thesis, we apply a string matching technique called parameterized matching to evaluate the isomorphism and containment relations between graphs with or without attributes.

In the rest of this chapter, we examine these problems and their relationships more closely. Specifically, in Section 1.1, we consider some of the most fundamental concepts and application areas of graphs. The different variants of the graph matching problem are introduced in Section 1.2. Then, in Section 1.3, some applications of querying graphs are discussed. The basic concepts of parameterized matching are presented in Section 1.4. Finally, in Section 1.5 our contributions are outlined.

1.1. Graphs: Concepts and Applications

A graph $G = (V, E)$ consists of a set V of nodes (or vertices), $n = |V|$, and a set E of edges, $m = |E|$, where the edges are ordered pairs of the nodes that represent links between them, i.e., $E \subseteq V \times V$. Let $\mathcal{E}_G = V \cup E$ denote the set of *graph elements* of G , i.e., the set of nodes and edges in G . In Figure 1-1, three examples of graphs are presented. For more generality, in this thesis we consider multigraphs. A multigraph is a graph where multiple edges between two distinct nodes and self loops are permitted. We distinguish the edges that have the same end nodes by the notation of the edge; for example, $e = (u, v)$ and $e' = (u, v)$.

Multigraphs are useful for representing sets of entities of different kinds and their relationships. Some of the domains where multigraphs can be found include the Semantic Web [131], the worldwide web [3], communication networking [20], social networking [114], interactive gaming [91],

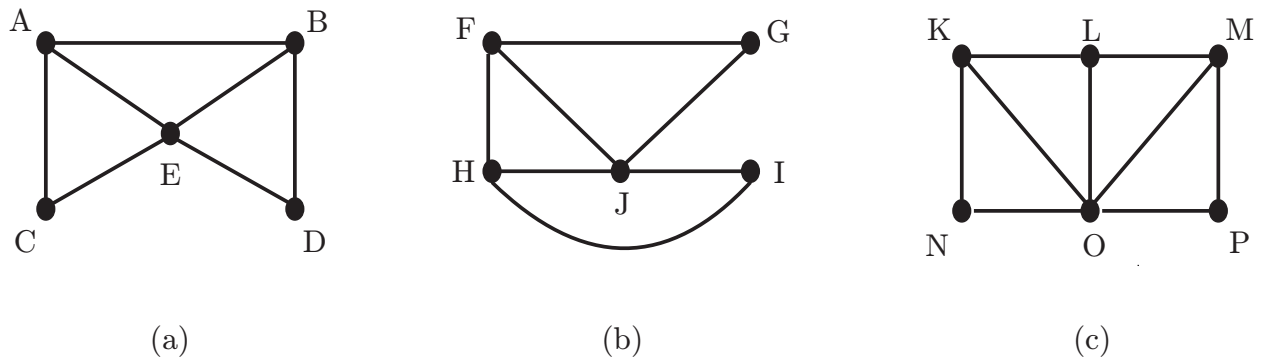


Figure 1-1.: Examples of graphs. Graphs (a) and (b) are isomorphic. There exists a subgraph in graph (c) that is isomorphic to graph (a) and (b).

geographic information systems [98], pattern recognition [126], pattern analysis [146], computer vision [65], artificial intelligence [118], information retrieval [119], knowledge discovery [100], data mining [153], electronics [138], computer aided design [86], chemoinformatics [67] and bioinformatics [92]. For instance, multigraphs are used to provide structural descriptions of images by decomposing them into different components that are modelled through nodes and the relationships of such components are modelled through edges [44]. Some of the types of images that have been described in this way are handwritten characters, ideograms and symbols [45]. On the other hand, in bioinformatics, several types of information can be represented through multigraphs: a protein structure, considering the set of residues as the nodes and their spacial proximity as the edges, or a protein interaction network where the nodes represent the proteins and the edges represent the physical interactions [78].

However, multigraphs are not only useful to represent real-world phenomena with nodes of the same type connected by edges of the same type. In many applications, it is necessary to use multigraphs with attributes, called *attributed multigraphs*, which are multigraphs where both the set of nodes and the set of edges are sets of entities with different types and characteristics. For example, if the domain is music, we may have an information network where node types are *singer*, *song* and *album*, and edge types are *performs* and *containedIn* to connect *singer* with *song* and *song* with *album*, respectively. The attributes of *singer* may be *name*, *birthday* and *website* while the attributes of *album* may be *name*, *length* and *year*. Moreover, there is an *ontology* associated to an attributed multigraph that establishes the possible concepts, the types of relationships permitted between two types of concepts and the restrictions on the attributes of both concepts and relationships [69]. For instance, the ontology associated to the multigraph of our running example would forbid a relationship of type *performs* between concepts of types *album* and *song*.

Another important area where attributed multigraphs are used is social networks. For instance, in Figure 1-2, we show an example of a social network where nodes represent people and photos

while edges establish friendship and person-tagged-in-photo relationships.

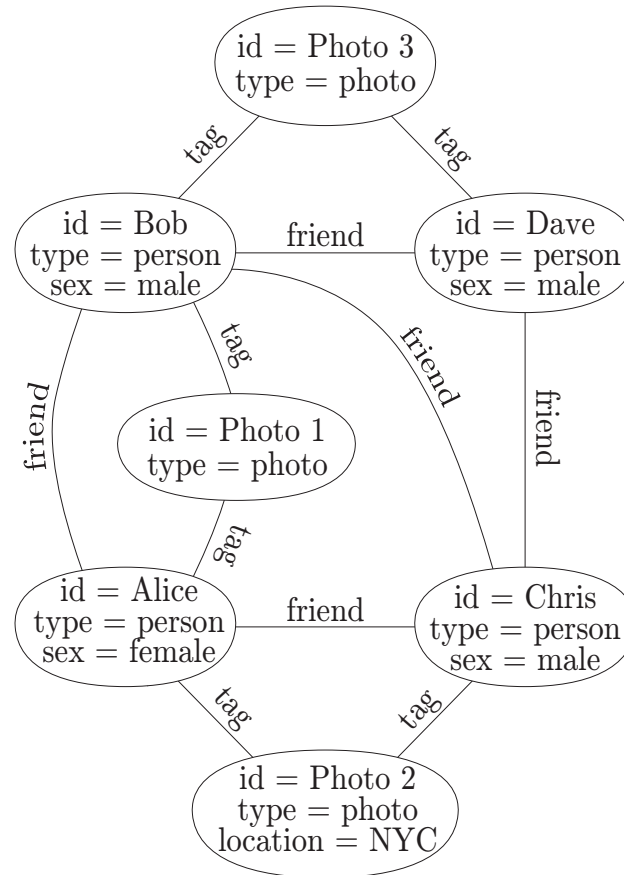


Figure 1-2.: Example of a social network represented as an attributed multigraph.

Considering the great amount of information that is represented through multigraphs nowadays, in these and in several other areas, the problem of querying multigraphs has significantly gained importance in recent years. In the next section, we introduce the main concepts of graph matching.

1.2. Graph Matching Problems

There are different problems associated to matching multigraphs. We first consider the general problems tackled in theoretical computer science and then the problems of practical interest for attributed multigraphs. The most basic problem is *graph isomorphism* which consists of determining whether two multigraphs have the same structure, i.e., there exists a bijection that associates the nodes/edges of the two multigraphs such that the adjacency relation is preserved. More formally, graph isomorphism can be defined as follows.

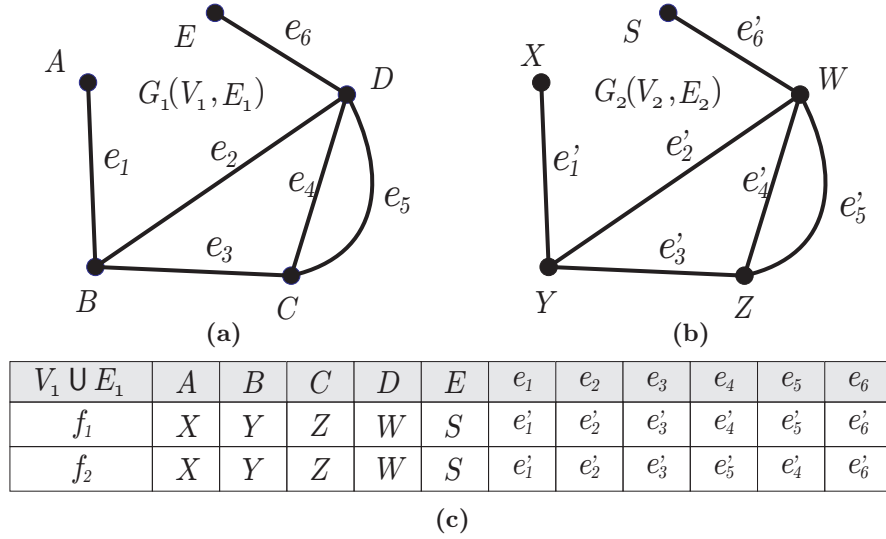


Figure 1-3.: Isomorphism example: the multigraphs presented in (a) and (b) are isomorphic; the functions that define the isomorphism are presented in (c). The difference between f_1 and f_2 is that $f_1(e_4) = e'_4$ and $f_1(e_5) = e'_5$ while $f_2(e_4) = e'_5$ and $f_2(e_5) = e'_4$.

Problem 1 (Graph Isomorphism). *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two multigraphs such that $n = |V_1| = |V_2|$ and $m = |E_1| = |E_2|$. The Graph Isomorphism problem determines whether there exists a bijective mapping function $f : \mathcal{E}_{G_1} \rightarrow \mathcal{E}_{G_2}$, such that*

$$\forall_{u,v \in V_1}, e = (u, v) \in E_1 \iff f(u), f(v) \in V_2 \wedge f(e) = (f(u), f(v)) \in E_2 \quad (1-1)$$

For example, the multigraph in Figure 1-1(a) is isomorphic to the multigraph in Figure 1-1(b) under the bijection $f : (A, B, C, D, E) \rightarrow (H, F, I, G, J)$. Other example of graph isomorphism is presented in Figure 1-3(a,b); furthermore there are two possible mapping functions that define the isomorphism (see Figure 1-3(c)). A closely related problem is *subgraph isomorphism*.

Problem 2 (Subgraph Isomorphism). *Let G_1 and G_2 be two multigraphs. The Subgraph Isomorphism problem consists of determining whether there exists a subgraph in G_2 isomorphic to G_1 .*

For example, if we remove the node P and its adjacent edges from the multigraph in Figure 1-1(c), we obtain a multigraph that is isomorphic to the multigraphs presented in Figures 1-1(a) and (b). A naive solution for these problems could search for all the possible mappings; however, its search space is exponential.

A lot of research about both graph and subgraph isomorphism has been carried out. Interestingly, even though subgraph isomorphism has been proven to be NP-Complete, the exact complexity of graph isomorphism has not been determined yet [44]. Due to the similarity of the problems, most of the existing solutions solve both of them. In particular, Ullmann's algorithm [148] is the traditional solution. Notwithstanding, a more recent algorithm, called VF2, experimentally outperformed

Ullmann's algorithm for many cases [44]. On the other hand, the NAUTY algorithm [102] is another traditional solution for only graph isomorphism. These algorithms have exponential worst-case performance since isomorphism is a hard problem. Except for some easy cases, solving isomorphism generally takes much longer time if there is no match; in such case, all the possible mappings are progressively searched until shown not to lead to an isomorphism. Several heuristics, however, are employed to find likely mappings quickly. A good algorithm for determining isomorphism should quickly find isomorphic multigraphs in many cases. In this thesis, we propose a new approach to solve both graph and subgraph isomorphism that makes use of some heuristics to detect isomorphism at an early stage of the search.

On the other hand, some models of interesting queries on attributed multigraphs have been proposed. Particularly, *reachability queries* consist of determining whether two nodes in the multigraph are somehow connected through an unrestricted path. For example, considering the multigraph of Figure 1-2, we can say that *Photo 3* is reachable from Alice as one of her friends is tagged in such photo. Besides large-scale social networking, reachability queries have important applications in several other areas. For instance, on biological multigraphs, it is relevant to find genes whose expressions are influenced by a given molecule [149]. Moreover, reachability queries are also useful to query XML databases and domain ontologies [84].

In recent years, special types of constrains, like the permitted edge types on the connecting path, have been included in reachability queries [83]. For example, considering the attributed multigraph of Figure 1-2, we might want to know whether *Dave* is connected to a female using *friend* edges, i.e., whether there is a female in his network. The output is *true* as Alice can be reached through paths coming from either Bob or Chris. Later, reachability queries were extended to support regular expressions that establish the edges types on the connecting path [59]. However, such regular expressions have limited expressive power as they do not support the Kleene operator nor predicates on intermediate nodes. Then, a model that supports these features was developed [127, 128].

Other type of queries on attributed multigraphs is *pattern match queries*. Each pattern match query is a query multigraph that searches for matches in an attributed multigraph such that: (i) the adjacency relation of the matches is the same as the one of the query; and (ii) each node/edge in the query specifies a predicate to be satisfied by its corresponding node/edge in the match. More formally, the problem of finding all the matches can be defined as follows.

Problem 3 (Pattern Match Query Problem). *Let P be a pattern match query and G be an attributed multigraph. The Pattern Match Query problem consists of finding the set of subgraphs of G that are isomorphic to P , and whose graph elements satisfy the predicates on the corresponding graph elements in P .*

This problem is clearly associated to subgraph isomorphism; thus, it can be solved with straight-

forward adaptations of subgraph isomorphism algorithms that include predicate evaluations. For example, in Figure 1-4(a), we show a pattern match query that aims to find a pair of friends, where one of them is a female, that are tagged in a photo. The output of this query on the attributed multigraph of Figure 1-2 is presented in Figure 1-4(b) and (c).

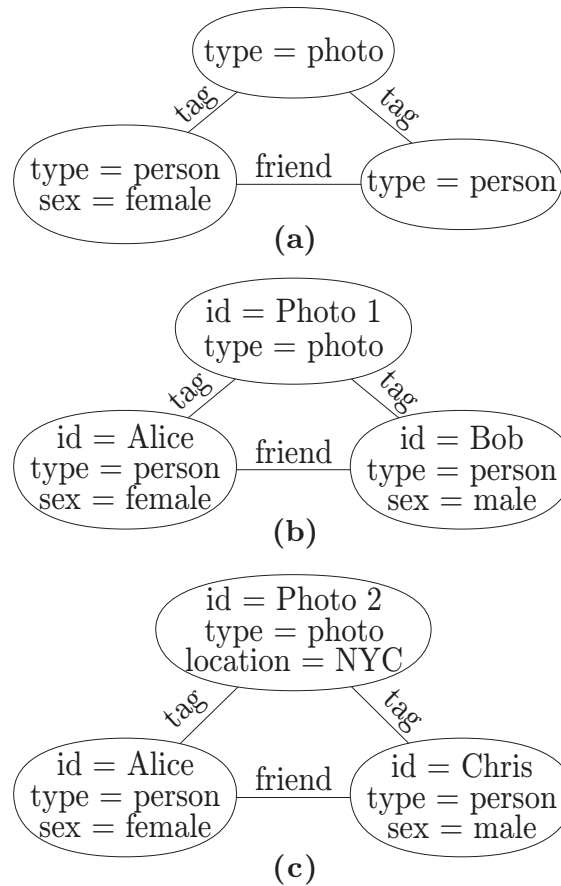


Figure 1-4.: Example of a pattern match query for the attributed multigraph presented in Figure 1-2. (a) Pattern match query. (b, c) Output reported.

Later, a new type of query on attributed multigraphs was introduced [59]. These queries, called *pattern queries*, constitute a combination of reachability and pattern match queries. Specifically, the queries are multigraphs where each node is associated to a predicate and each edge, along with its end nodes, establishes a reachability query. The output is associated to the set of nodes in the attributed multigraph that correspondingly satisfy both the predicates and the reachability queries in the query graph. However the output expressed corresponds to the set of global matches for each edge, i.e., reachability query in the query graph [59]. Then, the relative relationships between the nodes presented in the output is not easy to interpret. Moreover, these queries do not support the Kleene operator nor predicates on intermediate nodes [59].

In this thesis, we introduce *generalized pattern queries* as a new type of queries that evaluates attribute predicates, structural requirements and reachability requirements. These queries, besides allowing edge-to-path mappings, also support predicates on intermediate nodes/edges and operators like union and the Kleene star in the reachability requirements. Moreover, the output produced is easy to interpret: it consists of the set of all the solution instances where each instance is an ordered set of nodes that correspondingly satisfy the nodes in the query; thus, the relative relationships between the output nodes from each solution instance and the query is straightforward to determine. Then, the graph matching problems we consider in this thesis are graph isomorphism, subgraph isomorphism, solving pattern match queries and solving generalized pattern queries. In the next section, we discuss some applications of these problems.

1.3. Applications of Graph Matching

The graph matching problems have applications in different domains [78, 159]. Some examples are shown below:

- To find all heterocyclic chemical compounds that contain a given aromatic ring and a side chain. In this context, chemical compounds are modelled as graphs where the nodes represent atoms and the edges represent bonds.
- To find all protein structures that contain an α - β -barrel motif that is specified as a cycle of β strands embraced by another cycle of α helices [27].
- To determine whether a protein complex query from one species is functionally conserved in another species. The protein complex can be represented as a graph where the nodes are proteins labelled by Gene Ontology.
- To find all the instances from a Resource Description Framework (RDF) graph where two departments of a company share the same shipping company. The nodes are of type *department* and *company* and the edges of type *shipping*.
- To locate the occurrences of a suspicious bug that arises as a distortion in the control flow within a large software system that can be represented as a large static or dynamic call graph [55].
- To find all the co-authors from a bibliographic information network, such as DBLP, in a specified set of conference proceedings.

Furthermore, there are many other applications of graph matching in different areas including: 2D and 3D image analysis [135, 96, 150, 144], image database [80, 121], video analysis [133, 71, 125],

document processing [62, 66, 94], biometric identification [140, 57] and biomedical engineering [152, 64]. More information on such applications is provided in a recent survey [43].

1.4. Parameterized Matching

In this section, we introduce string matching as we propose to use it to solve the graph matching problems. String matching is definitely one of the foremost and most basic and useful computational primitives [8]. The input to the string pattern matching problem consists of two strings: the *pattern* $P = P_{1\dots m}$ and the *text* $T = T_{1\dots n}$. The output should list all the occurrences of the pattern in the text, i.e., all the positions i in T such that $P_j = T_{i+j-1}$ for all $1 \leq j \leq m$. Note that the symbols in the strings are chosen from some set which is called an *alphabet*. An alphabet could be any collection of symbols and it is normally drawn from a set of pre-existing characters which is habitually designated as the common ASCII¹ code set. Over the years, several variants of this problem have been proposed in order to support a wider range of applications. For instance, in the early nineties, a string matching variant called *parameterized matching* was proposed as an aid to detect duplicate code in large software systems.

Duplication in code occurs when there are some sections of code that are exactly equal (such as literals and reserved keywords) and some other sections of code that are the same, except for a systematic change of parameters (such as identifiers or constants). Then, the code can be seen as a string of tokens, where each token belongs to either of the following alphabets: (i) an alphabet Σ_C of constant symbols for the tokens of code sections that remain exactly the same; and (ii) an alphabet Σ_P of parameter symbols for the tokens of code sections that could have the mentioned systematic change. Then, the parameterized matching problem can be defined as follows.

Definition 1 (Parameterized-Match). *Let Σ_C be the constant symbol alphabet and Σ_P be the parameter symbol alphabet, where Σ_C and Σ_P are disjoint. Two length- ℓ strings $X = X_{1\dots\ell}$ and $Y = Y_{1\dots\ell}$, defined over $(\Sigma_C \cup \Sigma_P)^*$, are said to be a parameterized-match, or a p-match, if there exists a bijective function $g : \Sigma_C \cup \Sigma_P \mapsto \Sigma_C \cup \Sigma_P$ such that $g(Y_i) = X_i$, $1 \leq i \leq \ell$ so that g is identity for the the symbols from Σ_C .*

In other words, X and Y are a p-match if one string can be transformed into the other by renaming its parameters through a bijective function $g : \Sigma_C \cup \Sigma_P \mapsto \Sigma_C \cup \Sigma_P$, such that g is identity for the constant symbols. Note that, g can be chosen from $|\Sigma_P|!$ different possible mapping functions. As an example, Figure 1-5 shows two equal-length strings $X = xbyyxbx$ and $Y = zbxxzbxz$ defined over $\Sigma_C \cup \Sigma_P$, where $\Sigma_C = \{b\}$ and $\Sigma_P = \{x, y, z\}$. In Figure 1-5(a), the 6 possible bijective functions from the symbols in Y to the symbols in X are shown. We conclude that X and Y

¹American Standard Code for Information Interchange.

parameterized-match because there is a function, specifically r , such that $r(Y_i) = X_i$ for every $1 \leq i \leq 7$, and the only constant symbol, b , has an identity mapping, i.e., $r(b) = b$.

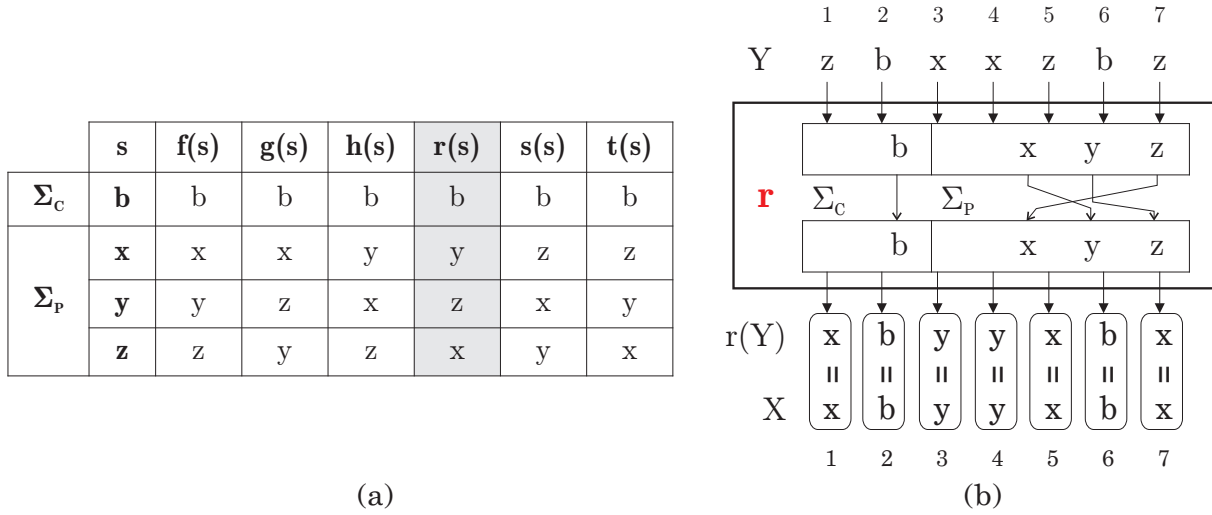


Figure 1-5.: Example of a parameterized-match between the strings $X = xbyyxbx$ and $Y = zbxzbxz$ both defined over $\Sigma_C \cup \Sigma_P$, where $\Sigma_C = \{b\}$ is the constant alphabet and $\Sigma_P = \{x, y, z\}$ is the parameter alphabet. (a) All the 6 possible bijective functions from the symbols in Y to the symbols in X such that the constant symbol has an identity mapping. (b) Successful attempt to transform Y into X through r .

Furthermore, two equal-length strings X and Y that parameterized-match have the same structure. Let us suppose that i and j are the only occurrences of the symbol α in Y . Then, the existence of a bijective function g that maps the symbols in Y to the symbols in X implies that $g(\alpha) = X_i = X_j = \beta$ and that β has no other occurrences in X . As this applies for all the distinct symbols α in Y , we can conclude that the following facts hold: (i) X and Y have the same number of distinct symbols; (ii) the first occurrence of each distinct symbol α in Y takes place in the same position of the first occurrence of the symbol $g(\alpha)$ in X ; and (iii) the relative distances among the different occurrences of each α in Y are the same relative distances among the occurrences of $g(\alpha)$ in X . Therefore, two strings that parameterized-match have the same structure, i.e., they are the same except for a systematic change of the symbols. We illustrate these properties for our running example in Figure 1-6. For instance, notice that the first occurrence of x in Y is at position 3 and its second occurrence is one position away; the occurrences of y in X take place at corresponding positions.

In 1993, Brenda Baker [15] was the first researcher to have addressed this problem, and many others [5, 9, 16, 17, 18, 41, 76, 68, 90, 53, 124] since have followed Baker’s work. She did, indeed, open up a wide field of extensive research that soon was generalized to other fields. Over the years,

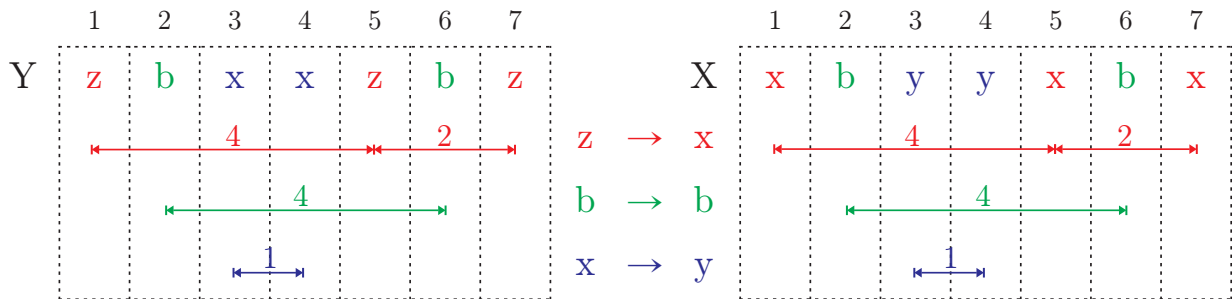


Figure 1-6.: Two strings $X = xbyyxbx$ and $Y = zbxzxbz$ that parameterized-match. It is shown that the structure of both strings is the same as the relative distances among the occurrences of their distinct symbols are equal.

other lines of research that have been pursued are: parameterized matching under edit distance [19], parameterized matching under Hamming distance [77, 7], parameterized matching under LCS distance [87], multiple parameterized matching [81], 2-dimensional parameterized matching [4] and function matching [4, 6, 106]. This accelerated research could only be justified by the usefulness of its practical applications such as in software maintenance [15], image processing [139, 11] and computational biology [4].

1.5. Our Contributions

In this thesis, we propose a new application for parameterized matching: the solution of the graph matching problems. Our initial motivation for this is based on the fact that in parameterized matching we must determine whether two strings that have the same structure, i.e., the relative distances among the occurrences of the distinct symbols in the strings are equal (see Figure 1-6). Similarly, in graph matching, we must determine if two graphs have the same structure, i.e., if the graphs are isomorphic. Furthermore, in parameterized matching (and graph isomorphism) the match depends upon the existence of a bijection from the symbols (nodes) in one string (graph) to the symbols (nodes) in the other string (graph). Therefore, this thesis defines a new model for solving graph matching based on parameterized matching. Specifically, given two graphs G_1 and G_2 , we represent G_1 in a linear manner which we call *graph linearization*. Then, we evaluate if this representation parameterized-matches a walk on G_2 .

This document is comprised by two parts. In Part I, we propose a new solution for both graph and subgraph isomorphism. Then, in Part II, we adapt this solution to answer different types of queries on attributed graphs. In particular, we solve pattern match queries and generalized pattern queries.

More specifically, in Part I, we present our approach to solve isomorphism in multigraphs through parameterized matching as follows:

- We define graph linearization, as a linear representation of graphs, and formally show how it can be used to solve graph and subgraph isomorphism. We develop the *Graph Linearization Algorithm* – GLA, an asymptotically length-optimal algorithm that efficiently linearizes a graph through greedy heuristics (Chapter 3).
- We propose a matching algorithm, called PMG, that solves graph isomorphism. Given two multigraphs G_1 and G_2 , PMG calculates a linearization of G_1 and determines whether there exists a walk on G_2 that parameterized-matches such linearization. If so, the graphs are isomorphic (Chapter 4).
- We adapt the PMG algorithm to solve subgraph isomorphism; this adaptation is called PMG-SI (Section 4.6).

Then, in Part II, we redefine our approach to efficiently solve queries of interest on attributed graphs as follows:

- We define the data model and the query model. Especially, we define generalized pattern queries, a new type of queries with high expressive power that supports structural predicates, attribute predicates and reachability evaluation (Chapter 5).
- We extend the concept of graph linearization to represent generalized pattern queries. Moreover, we show how the linearization of a generalized pattern query can be used to find its solution on an attributed graph. We present the *Enhanced- Graph Linearization Algorithm* – E-GLA, an algorithm that exploits the attributed graph statistics to generate linearizations of the query that will incur in low matching cost (Chapter 6).
- We develop a solution for generalized pattern queries on attributed graphs also based on query linearization through E-GLA (Chapter 7).

Some contents of this thesis have already been presented in [105, 110, 104, 106, 103, 109, 93]. Other articles also developed during the PhD program, that are related to this thesis, are [22, 108, 107, 111, 112, 23, 21, 46].

2. Related Work

In this thesis we propose a new solution for the graph matching problems based on parameterized matching. Thus, in this chapter we present a literature review of these topics. In particular, Section 2.1 describes the most efficient algorithms for graph isomorphism and subgraph isomorphism. Then, Section 2.2 considers graph matching in attributed graphs. Specifically, we cover different types of queries on attributed graphs, including reachability queries, pattern match queries and pattern queries. Finally, Section 2.3 includes the basic problems, solutions and extensions of parameterized matching.

2.1. Solutions for Graph and Subgraph Isomorphism

It has been proven that subgraph isomorphism is a NP-Complete problem [70]; however, the exact complexity of graph isomorphism is still an open question [70, 44]. Because the two problems are very related, most of the solutions for one of them works for the other. In this section, we describe the most efficient algorithms: Ullmann’s algorithm (Section 2.1.1) and VF2 (Section 2.1.2).

2.1.1. Ullmann’s Algorithm

One of the first algorithms for solving both problems was proposed by Ullmann back in 1976 [148]. His solution is similar to Corneil and Gottlieb’s algorithm for graph isomorphism [47] but differs from it in that graphs are not processed separately. Ullmann’s solution consists of backtracking in a search tree using a mechanism to prune the search space called *refinement procedure*. For generality, we describe the algorithm for subgraph isomorphism, but it also solves graph isomorphism. First, we show how the possible mappings are enumerated and then how to apply the refinement procedure.

Ullmann’s algorithm determines whether graph $G_1 = (V_1, E_1)$, where $n_1 = |V_1|$ and $m_1 = |E_1|$, is isomorphic to a subgraph in $G_2 = (V_2, E_2)$, where $n = |V_2|$ and $m = |E_2|$. Let us denote the adjacency matrices of G_1 and G_2 as A and B , respectively. The possible mappings are stored in a binary matrix M of n_1 rows and n columns. In particular, $M[i][j]$, for $1 \leq i \leq n_1$ and $1 \leq j \leq n$, is equal to 1 if node $v_i \in V_1$ can be mapped to node $v_j \in V_2$; otherwise, $M[i][j] = 0$. An important property of this matrix is that it must contain exactly one 1 in each row and at most one 1 in each column. This is to ensure the injective mapping from the nodes of G_1 (i.e., the rows) to a subset of

the nodes of G_2 (i.e., the columns).

The idea of the algorithm is exploring all the possible mappings by permuting the rows and columns of B and comparing adjacency with A . This can be done by multiplying B with the possible mappings M . Specifically, for a given M , the multiplication MB moves row j to row i , for all $M[i][j] = 1$. Thus, $(MB)^T$ moves column j to column i . Similarly, $C = M(MB)^T$ moves column j to column i and row j to row i for every $M[i][j] = 1$. Then, in order to evaluate if a given mapping M is an isomorphism, we check whether the adjacency relation defined in A is contained in C . In other words, if M is an isomorphism, the following condition must be satisfied:

$$\forall_{i,j}(A[i][j] = 1) \Rightarrow (C[i][j] = 1) \quad (2-1)$$

Notice that, in case of graph isomorphism, \Rightarrow is replaced by \Leftrightarrow as the adjacency matrices must be equal.

The possible mappings are explored as follows. An initial matrix M^0 of n_1 rows and n columns is constructed by setting $M^0[i][j] = 1$ if mapping $v_i \in V_1$ to node $v_j \in V_2$ is possible. Specifically, it is possible to do such mapping if the degree of v_j is greater or equal to the degree of v_i in the case of subgraph isomorphism; in the case of graph isomorphism, the degrees must be equal. Otherwise, we set $M^0[i][j] = 0$. Then, M^0 becomes the root of the search space of the permutation matrices. In particular, a Depth-First Search (DFS) approach is used: at level k of the DFS search tree, the matrix M^k only keeps one of the 1's at the row k of M^{k-1} ; the others are reassigned to 0. Thus, at level n_1 , the mapping for all nodes in V_1 should have been found. Therefore, for each matrix M^k , we evaluate condition 2-1 to determine if it is an isomorphism. Note that if, at any point, a given M^k has a row with no 1's, it is not necessary to explore its successors as it will not lead to an isomorphism.

Now Ullmann's refinement procedure, which is the key idea of his algorithm to prune the search space, is introduced. If, at any point, a node $v_j \in V_2$ is among the possible mappings of a node $v_i \in V_1$, then every neighbour of v_i must have at least one possible mapping among the neighbours of v_j . If this condition does not hold, we can safely remove v_j from the possible mappings of v_i as we know that forthcoming mappings cannot be established in this direction. Recall that, in terms of the mapping matrix M , the possible mappings for v_i are the nodes v_j for which $M[i][j] = 1$. Then, for each $M[i][j] = 1$ in matrix M , the refinement procedure must be evaluated as follows:

$$\forall_x(A[i][x] = 1) \Rightarrow \exists_y(M[x][y] \cdot B[y][j] = 1) \quad (2-2)$$

If this condition does not hold, then we set $M[i][j] = 0$. This procedure is performed for every $M[i][j] = 1$. However, changing a 1 to 0 can make condition 2-2 is no longer satisfied for other 1's in M . Then this process must be repeated over and over again until there is an iteration where no 1 in the matrix is changed to 0. The refinement procedure is applied for each matrix M^k in the DFS

search tree, including M^0 [148]. The space complexity of Ullmann's algorithm is $\Theta(n^3)$; its time complexity in the best case is $\Theta(n^3)$ while its worst-case time complexity is $\Theta(n! n^2)$ [44].

Other refinement procedures to reduce the search space have also been considered. Specifically, Haralick and Elliot proposed forward-checking and looking-ahead [74], and Kim and Kak used discrete relaxation [88]. Another approach that has been taken consists of a reduction to the maximal clique detection problem [56, 115]. Besides, Blake proposed a solution based on a partition according to lattice theory to reduce computational complexity [24].

However, Ullmann's algorithm is one of the most used solution because of its good performance. Ullmann's algorithm was compared against other graph matching solutions and it turned out to be the one with best matching time [113]. Furthermore, this algorithm also permits the comparison of semantic information during the process. Notwithstanding, for large graphs, the time required by Ullmann's algorithm is still too high to be tractable. During the last three decades, there have been many attempts for solving the graph matching problem on large graphs. Most of them achieve low time complexity by imposing restrictions on the topology of the graphs. Some of the most important contributions in this direction are polynomial algorithms for trees, planar graphs and bounded valence graphs [95]. On the other hand, some algorithms based on continuous optimization methods like neural networks, simulated annealing [79], genetic algorithms [28] and probabilistic relaxation [39] have been proposed. They find solutions in a reasonable time without imposing constraints on the topology of the graph; the drawback is that their solutions are *approximate*.

2.1.2. The VF2 Algorithm

More recently, new mechanisms to query large graphs have been devised. Specifically, Cordella *et al.* proposed a deterministic algorithm, called VF2, that does not impose any restrictions on the topology of the graph and supports attributed graphs [44]. This algorithm achieves a reduced computational complexity by using a set of feasibility rules during the matching process. Furthermore, sophisticated data structures are used to reduce space complexity. Experimental results prove that VF2 is a competitive graph matching algorithm.

In particular, VF2 determines if graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic by constructing a mapping set $M \subset V_1 \times V_2$, where each pair (v_1, v_2) represents the mapping of a node v_1 in G_1 to a node v_2 in G_2 . Then, if G_1 and G_2 are isomorphic, the mapping set M is a bijection that preserves the adjacency relation of the graphs. The process to find M can be described using a State Space Representation (SSR) [117] where each state is a partial mapping. Specifically, a partial mapping $M(s)$ leads to a new state $M(s')$ by inserting a new pair $(v_1, v_2) \in V_1 \times V_2$ that maintains consistency, i.e., that does not preclude the possibility of obtaining a complete mapping. A partial mapping is consistent if the subgraphs that its nodes constitute in G_1 and G_2 are isomorphic.

Let $M_1(s)$ and $M_2(s)$ be the set of nodes of V_1 and V_2 in the mapping $M(s)$, respectively. Also, let $T_1^{in}(s)$ be the set of nodes in V_1 that are not in $M_1(s)$ but are the origin of edges ending into nodes in V_1 . Likewise, let $T_1^{out}(s)$ be the set of nodes in V_1 that are not in $M_1(s)$ but are the destination of edges starting from nodes in V_1 . Furthermore, let $R_1(s)$ denote the nodes in V_1 that are neither in $M_1(s)$, $T_1^{in}(s)$ or $T_1^{out}(s)$. Sets $T_2^{in}(s)$, $T_2^{out}(s)$ and $R_2(s)$ are defined in a similar way. Then, consistency maintenance is verified by five feasibility rules:

- For each predecessor v'_1 of v_1 in $M_1(s)$, there must be a predecessor v'_2 of v_2 in $M_2(s)$ such that $(v'_1, v'_2) \in M(s)$, and vice versa.
- For each successor v'_1 of v_1 in $M_1(s)$, there must be a successor v'_2 of v_2 in $M_2(s)$ such that $(v'_1, v'_2) \in M(s)$, and vice versa.
- The number of successors of v_1 that are in $T_1^{in}(s)$ must be equal to the number of successors of v_2 that are in $T_2^{in}(s)$. Similarly, the number of predecessors of v_1 that are in $T_1^{in}(s)$ must be equal to the number of predecessors of v_2 that are in $T_2^{in}(s)$.
- The number of successors of v_1 that are in $T_1^{out}(s)$ must be equal to the number of successors of v_2 that are in $T_2^{out}(s)$. Similarly, the number of predecessors of v_1 that are in $T_1^{out}(s)$ must be equal to the number of predecessors of v_2 that are in $T_2^{out}(s)$.
- The number of successors of v_1 that are in $R_1(s)$ must be equal to the number of successors of v_2 that are in $R_2(s)$. Likewise, the number of predecessors of v_1 that are in $R_1(s)$ must be equal to the number of predecessors of v_2 that are in $R_2(s)$.

The last three rules should be adapted for subgraph isomorphism. In particular, the equal constraints must be replaced for inequalities. In case of isomorphism in attributed graphs, semantic feasibility is also evaluated. Specifically, when a new pair $(v_1, v_2) \in V_1 \times V_2$ is considered to extend the partial mapping $M(s)$, it is verified that v_1 and v_2 are semantically compatible. Furthermore, for each edge (v_1, v'_1) where $v'_1 \in M_1(s)$, there must be a node $v'_2 \in M_2(s)$ such that (v_2, v'_2) is semantically compatible with (v_1, v'_1) . Similarly, for each edge (v'_1, v_1) where $v'_1 \in M_1(s)$, there must be a node $v'_2 \in M_2(s)$ such that (v'_2, v_2) is semantically compatible with (v'_1, v_1) .

From a global perspective, the algorithm works as follows. For a given state $M(s)$, a recursive procedure called `MATCHVF2()`, is performed. In this procedure, all the pairs $(v_1, v_2) \in V_1 \times V_2$ that can be considered for inclusion in $M(s)$ are inserted into the set $P(s)$. For each pair $p \in P(s)$ that satisfies the feasibility and semantic rules, a new state $M(s')$ is generated by inserting (v_1, v_2) into $M(s)$. Then, the procedure `MATCHVF2()` is recursively called for each $M(s')$. The process ends when a complete mapping is found or when all the feasible mappings are explored. Note that `VF2` explores the search space in a DFS fashion starting from an empty mapping set.

In order to reduce space requirements, and then make it scalable to large graphs, VF2 makes use of the following data structures:

- Vectors $core_1$ and $core_2$, whose length is n and m , respectively. Specifically, $core_1[i]$ contains the index of the current mapping of node $v_i \in M_1(s)$ into G_2 ; in case v_i is not in the mapping yet, then $core_1[i] = null$. Vector $core_2$ is defined in the same manner to establish the mappings of $M_2(s)$ into G_1 .
- Vectors in_1 and in_2 , whose length is n and m , respectively. In particular, $in_1[i]$ is non-*null* if v_1 is either in $M_1(s)$ or $T_1^{in}(s)$; the value stored in $in_1[i]$ is the depth in the SSR tree of the state in which the node entered the corresponding set. Vector in_2 is defined in the same manner with respect to G_2 .
- Vectors out_1 and out_2 , whose length is n and m , respectively. They are defined in a similar way as in_1 and in_2 . For instance, $out_1[i]$ contains the depth in the SSR tree in which v_i entered either $M_1(s)$ or $T_1^{out}(s)$. In case v_i has been inserted in neither of these sets, $out_1[i]$ is equal to *null*.

These vectors allow to perform different operations of the algorithm efficiently. For instance, membership queries can be evaluated in constant time. Furthermore, VF2 does not require to store a copy of these vectors at each state of the search. Just one instance of each vector must be stored because if an element is non-*null* in a given state, it will have the same value in the descending states from s . This, along with the traversal order of DFS, allows to restore the previous values of the vectors when the algorithm backtracks. Then, the space complexity of VF2 is $\Theta(n)$, i.e., the size of the vectors. In the best case, the algorithm finds a complete mapping on the first attempt, i.e., just one length- n path in the SSR is explored. Then, the best-case time complexity of VF2 is $\Theta(n^2)$. On the other hand, the worst-case time complexity of VF2 is $\Theta(n!n)$ given that the algorithm visits $n!$ states in the worst case, i.e., the case where the complete mapping is the last one explored or when there is no complete mapping at all.

The VF2 algorithm was experimentally compared against Ullmann's algorithm and NAUTY to evaluate its performance on graph isomorphism. In 56 % of the cases, the best results were achieved by VF2 while, in 44 % of the cases, NAUTY was the most efficient. Moreover, the results indicate that NAUTY is more convenient for randomly connected graphs, while VF2 is more efficient for graphs with a more regular structure, especially for large graphs. It is important to remark that, in most applications, graphs follows certain regularity. This makes VF2 a good candidate to consider in practical environments. Furthermore, VF2 was compared against a straightforward adaptation of Ullmann's algorithm that supports attributes to evaluate subgraph isomorphism. Results show that VF2 performs significantly better. While the matching time rapidly grows with the size of the subgraph for Ullmann's algorithm, it remains almost independent of such size for VF2 [44].

2.2. Queries on Attributed Graphs

Several techniques have been proposed in the literature for querying graphs. However, the existing graph querying methods mainly focus on querying the topological structure of the graphs [122, 156, 159] and very few of them have considered the use of attributed graphs [127, 143]. In practice, it is more common that the querying requirements for the applications of graph databases (e.g., social networks or bibliographical networks) would involve querying the graph data (attributes on nodes/edges) in addition to the graph topology.

Answering queries that involve predicates on the attributes of the graphs (nodes or edges) is more challenging as it requires extra memory consumption for building indices over the graph attributes in addition to the structural indices in order to accelerate the query evaluation process. Furthermore, it makes the query evaluation and optimization process more complex (e.g., evaluation and join orders). Existing graph optimization techniques focus on either building index structures [156, 159] or on developing estimation modules for certain graph queries [120, 158]. These techniques are complementary to this work. They can be used to accelerate query processing in our approach, which exploits both attribute histograms and the topology of the graph to produce efficient linearizations that yield lower matching time. At the same time, these index structures cannot be used to answer all queries as they may require graph exploration. Two types of query which are widely used in many applications are *reachability queries* [40, 151, 84], and *pattern match queries* [159, 161]. In this section, we review the related work on reachability queries (Section 2.2.1), pattern match queries (Section 2.2.2), and a recently defined type of queries called *pattern queries* (Section 2.2.3).

2.2.1. Reachability Queries

Given a directed attributed graph $G = (V, E)$ and two nodes $u, v \in V$, a *reachability query* determines whether there exists a directed path in G from u to v . Graph reachability is closely related to the concept of *transitive closure*. Specifically, the transitive closure of G , denoted as $TC(G)$, is defined as the set of the pairs of nodes (u, v) such that v is reachable from u . Then, in order to determine reachability from u to v , it suffices to evaluate the membership of (u, v) in $TC(G)$. However, the transitive closure of large and dense graphs can be very large.

A reachability query on a directed graph G can be evaluated by using an associated directed acyclic graph (DAG) $G' = (V', E')$ of G . Such graph can be obtained by finding the strongly connected components of G , which takes $O(|V| + |E|)$. Each node in V' represents one of the components. Each edge $(u, v) \in E$, such that u and v belong to different components, is associated to an edge in E' between the nodes in V' that represent the corresponding components of u and v . Then, given the nodes $u, v \in V$, we can say that v is reachable from u if they are in the same component or if the component of v is reachable from the component of u . In the next algorithms, we assume that

the directed graph has been transformed into a DAG.

There are two main approaches to evaluate a reachability query on a graph $G = (V, E)$: (i) performing a DFS/BFS traversal; and (ii) pre-computing and maintaining the transitive closure of the graph. The former has high time complexity (i.e., $O(|V| + |E|)$). The latter allows to answer reachability queries in constant time but incurs high space requirements (i.e., $O(|V|^2)$). Existing solutions attempt to obtain a convenient trade-off between time and space requirements where the query time complexity varies between $O(1)$ and $O(|V| + |E|)$ (the complexity of approaches (i) and (ii)). This trade-off is achieved by constructing different types of indices that reduce the space storage of the transitive closure [157]. Table 2-1 summarizes the complexity of constructing indices under different approaches, their size and the corresponding query response time [84].

Table 2-1.: Complexity of solutions for reachability queries on a DAG $G = (V, E)$ [84]. **Notation:** t is the number of non-tree edges in the spanning-tree-based solutions.

| Algorithm | Query Time | Index Size | Index Construction |
|--------------------------|-------------------|--------------------|-----------------------------|
| DFS/BFS | $O(V + E)$ | - | - |
| Transitive Closure [136] | $O(1)$ | $O(V ^2)$ | $O(V \times E)$ |
| Tree Cover [1] | $O(\log V)$ | $O(V ^2)$ | $O(V \times E)$ |
| Labelling + SSPI [35] | $O(E - V)$ | $O(V + E)$ | $O(V + E)$ |
| GRIPP [145] | $O(E - V)$ | $O(V + E)$ | $O(V + E)$ |
| Dual-Labelling [151] | $O(1)$ | $O(V + t^2)$ | $O(V + E + t^3)$ |
| Chain Cover [36] | $O(\log k)$ | $O(k V)$ | $O(V ^2 + k V \sqrt{k})$ |
| Path-Tree Cover [85] | $O(\log^2 k')$ | $O(k' V)$ | $O(k' E + V \times E)$ |
| 2-Hop Cover [40] | $O(\sqrt{ E })$ | $O(V \sqrt{ E })$ | $O(V ^3 \times TC(G))$ |
| 3-Hop Cover [84] | $O(\log V + k)$ | $O(k V)$ | $O(k V ^2 Con(G))$ |

In the following, we describe selected solutions of Table 2-1 as presented in a recent survey [157]. In the late eighties, an algorithm that computes the transitive closure without any compression in $O(|V| \times |E|)$ time was developed by Simon [136]. In particular, this algorithm is an improvement on the solution proposed by Goralčíková and Koubek [72] a decade earlier.

Also, in the late eighties, Agrawal et al. propose the use of a spanning tree of the DAG to compress its transitive closure; such spanning tree is called the *tree cover* [1]. In this approach, the edges of the graph are classified based on their membership in the tree: if an edge appears in the tree, it is called a *tree edge*; otherwise, it is called a *non-tree edge*. First, let us consider the case where the tree cover is comprised of only tree edges, i.e. the graph is a tree. Each node u is associated to an interval-based label $[i, j]$, where i is the index and j is the *postorder number* of the node. The

postorder number of u is its relative position in the postorder traversal of the tree. The index of u is the lowest postorder number among its descendants or, in case it is a leaf, its own postorder number. Using these intervals, reachability can be checked with the following lemma: a node v , with postorder number k , is reachable from u iff $i \leq k < j$, i.e. the interval of v is contained in the interval of u [1].

This index is applied to general graphs as follows. A spanning tree of the graph is computed. In case the graph contains more than one connected component, a virtual root node can be added. Then, each node is associated to a set of intervals rather than a single interval. The set of each node initially contains only the interval $[i, j]$ with its index and postorder number, respectively, just like in the case of trees. Then, the nodes are sorted in reverse topological order. For each node u , all its outgoing edges are considered. Specifically, for each edge (u, v) , all the intervals associated to the node v are inserted into the set of intervals of node u . If an interval is subsumed by another interval in the set, it is discarded. It can be concluded that a node u' can reach node v' if the postorder number of v' is contained in one of the intervals of node u' . An algorithm that computes an optimal spanning tree, in the sense that it minimizes the required storage of the transitive closure, was also presented [1].

More recently, different variants of this approach, based on a spanning tree, have been developed. For instance, Chen et al. proposed a set of stack-based algorithms in 2005 [35]. In their approach, a label is assigned to each node in the tree; we denote the label of node u as $label(u)$. A predicate $\mathcal{P}(label(u), label(v))$ is used to determine graph reachability from node u to node v in the DAG. Specifically, if $\mathcal{P}(label(u), label(v)) = true$, then v is reachable from u . However, because not all the edges appear in the tree, $\mathcal{P}(label(u), label(v)) = false$ does not imply that v is not reachable from u . In order to tackle these cases, another data structure called *Surrogate & Surplus Predecessor Index* (SSPI) is used [157]. Another solution that makes use of a spanning tree of the DAG and labelling was devised by Trißl and Leser [145]. This approach, called *GGraph Indexing based on Pre- and Postorder numbering* (GRIPP), extends the pre- and postorder numbering scheme to support graphs. The search phase is performed by means of a hop technique and a set of pruning strategies. Notice that the complexity of this solution is the same as the one of Chen et al.

Also following the spanning-tree based approach, Wang et al. proposed a solution suitable for large sparse graph where graph reachability needs to be evaluated [151]. This approach also constructs a spanning tree where each node u is associated to an interval-based label $[start, end)$, where $start$ and $end - 1$ are the preorder and postorder number of u in the spanning tree. This labelling scheme is called *Dual-I*. Furthermore, the non-tree edges are stored in a *link table*; such table is desired to be small. The selection of a convenient spanning tree and strategies to avoid superfluous non-tree edges have been studied [151]. This solution achieves constant query time while reduces the size of the index to $O(|V| + t^2)$, where t is the number of non-tree edges. Notice that this is efficient only if t is much lower than $|V|$, which does not occur in many real graphs [84].

Most of the algorithms that efficiently solve *shortest distance queries* [40, 32, 37, 154], and reachability queries for graphs do not support predicates on the connecting paths [40, 32, 151, 37, 154]. Because of the need for supporting semantic restrictions in the queries, without sacrificing the response time, recent work has been developed in this direction. For instance, a special type of reachability query, called *label-constraint reachability query*, that only accepts edge labels from a given set was proposed [83]. More recently, a revised definition of reachability query, where a regular expression of edge labels is used to specify the connecting path in the query, was proposed [59]; however such regular expression has limited expressiveness: neither the Kleene operator nor predicates on intermediate nodes are supported.

2.2.2. Pattern Match Queries

Pattern match queries have been defined and solved in terms of subgraph isomorphism due to its appropriateness for practical applications; some of the most relevant solutions following this approach are presented in [29, 35, 143, 38, 160, 159, 161], as outlined in the surveys [69, 132]. In order to make the pattern match queries more flexible and support more applications, the edges of the query have been allowed to map to paths in the graph [58, 61, 161]. The main challenge of the techniques based on subgraph isomorphism is scalability on the graph size given that the problem is NP-Complete. To address this issue, a new version of pattern match query was proposed [59].

On the other hand, given that graphs can be considered as databases, the database community introduced a formal language called *GraphQL* to query semantic graphs [78]. Moreover, different techniques are applied on *GraphQL* to support large graphs: use of neighborhood subgraphs, joint reduction of the search space and optimization of the search order. Experimental results on a biological database of thousands of nodes showed that *GraphQL* outperforms an *SQL*-based implementation [78]. Later, Zhao and Han proposed an indexing mechanism called *SPath* that leverages decomposed shortest paths around node neighborhoods as basic indexing units [159]. This mechanism achieves effectiveness in pruning the search space and scalability in index construction and deployment. Two experimental tests were performed: one on the same biological database used for testing *GraphQL*, and another one on a synthetic database that contains one million nodes; in both tests *SPath* significantly outperformed *GraphQL* [159].

2.2.3. Pattern Queries

To address this issue, a new version of pattern match query was proposed [59]. This variant is based on graph bounded simulation rather than subgraph isomorphism. Graph bounded simulation [60] is an extension of graph simulation for pattern match queries where bounds on the number of permitted hops are imposed. This restriction allows to solve pattern match queries in polynomial time, even when we allow that edges in the query map to paths in the graph. The corresponding

output presents in batch the global matches for each edge (reachability query) in the query.

However, like in the reachability queries also studied in [59], pattern match queries using the Kleene operator or containing predicates on intermediate nodes cannot be solved with such approach. In this paper, we introduce *generalized pattern queries* as a new type of queries that evaluates both attribute predicates and structural requirements. These queries, besides allowing edge-to-path mappings, also support predicates on intermediate nodes/edges and operators like union and the Kleene star in the reachability requirements. Moreover, the output produced is easy to interpret: it consists of the set of all the solution instances where each instance is an ordered set of nodes that correspondingly satisfy the nodes in the query; thus, the relative relations among the output nodes from each solution instance and the query is straightforward to determine.

On the other hand, graph query languages, based on either regular expressions [42, 51, 73], SQL-like languages [10, 123, 134], or procedural languages [78], to solve graph queries have been proposed. Such languages have limited expressive power and lack the support of declarative query interfaces. For instance, comparing with well-known query languages such as SPARQL and SQL, we support queries that cannot be expressed by either of them. (1) The SPARQL query language expresses pattern match queries over RDF data, and cannot express general reachability queries. In contrast, we target both reachability queries and pattern match queries. A recent specification of SPARQL allows a limited form of reachability with the triple pattern (subject, verb+, object). (2) SQL cannot express paths of arbitrary length, unless it is extended with recursion [69] to support closure operators. Notwithstanding, we use an idea that is similar to the query optimizer in a database system: the most restrictive conditions are often pushed down to the evaluation tree of a query plan; thus, the selective predicates are evaluated early in the execution.

2.3. Parameterized Matching

As an aid in software maintenance, *parameterized matching* was first defined by Brenda Baker to detect duplicate code in large software systems [15]. Later, the study on this problem was further extended due to its practical applications in different areas. In this section, we cover some of this research. Specifically, in Section 2.3.1, the formal definition of parameterized matching and its variants are given. Some parameterized matching algorithms are reviewed in Section 2.3.2 and some extensions are presented in Section 2.3.3. Finally, some of the most important applications of this pattern matching variant are shown in Section 2.3.4.

2.3.1. Definition of the Basic Problems

Let Σ_C be the constant symbol alphabet and Σ_P be the parameter symbol alphabet. We assume that Σ_C and Σ_P are disjoint from each other and the set of nonnegative integers. A *parameterized string* or a *p-string* is defined as a string of symbols in $(\Sigma_C \cup \Sigma_P)^*$. Furthermore, two length- m

p-strings $X = X_{1\dots m}$ and $Y = Y_{1\dots m}$ are said to be a *parameterized-match* or a *p-match*, if one p-string can be transformed into the other by renaming its parameters through a bijective function $g : \Sigma_C \cup \Sigma_P \mapsto \Sigma_C \cup \Sigma_P$, such that g is identity for the symbols from Σ_C (see Definition 1). Note that, g can be chosen from $|\Sigma_P|!$ different possible mapping functions (an example is given in Figure 1-5).

Parameterized Matching is the problem of finding all the parameterized-matches of a pattern in a text. More formally, let us consider two p-strings: the pattern $P = P_{1\dots m}$ and the text $T = T_{1\dots n}$, both defined over $\Sigma_C \cup \Sigma_P$. Also, let T^i denote the length- m text window starting at position i of T , i.e., $T^i = T_{i\dots i+m-1}$. Then, pattern P is said to parameterized-match T^i iff there exists a bijective mapping function g_i such that $g_i(P_j) = T_{i+j-1}$, $1 \leq j \leq m$, so that g_i is identity for the symbols from Σ_C . Notice that if there exists a function $g_i(P_j) = T_{i+j-1}$, $1 \leq j \leq m$, the inverse of g_i also exists given that g_i is bijective. So we can equivalently say that P parameterized-matches T^i if there exists a bijective mapping function $g'_i(T_{i+j-1}) = P_j$ so that g'_i is identity for the symbols from Σ_C . Note that, at each position i of T , a different g_i can be considered to determine the existence of a parameterized-match between the pattern and the text window starting at position i . The output of the problem is the set of indices i , $1 \leq i \leq n - m + 1$, such that P parameterized-matches T^i . This problem is also referred as *Parameterized Fixed Pattern Matching (PFPM)* [81].

Some other problems related to parameterized matching have been defined to be able to support more applications. One of them is finding the *maximal p-matches over a threshold length* of a p-string text, defined as follows. Let $T = T_{1\dots n}$ be a p-string and $T_{i\dots i+k}$ and $T_{j\dots j+k}$ two p-substrings of it that p-match. This p-match is said to be *left-extensible* if $T_{i-1\dots i+k}$ and $T_{j-1\dots j+k}$ are a p-match and is *right-extensible* if $T_{i\dots i+k+1}$ and $T_{j\dots j+k+1}$ are a p-match, where $1 \leq i \leq i+k \leq n$, $1 \leq j \leq j+k \leq n$ and $i \neq j$. If a p-match is neither left-extensible or right-extensible, it is said to be a *maximal p-match*. Maximal p-matches are not an equivalence relation, because they are not transitive, so the output of the maximal p-matches problem must list pairs of p-substrings rather than an equivalence class. Thus, the output of the maximal p-matches of a p-string text $T = T_{1\dots n}$ over a threshold length t problem must report the set of all pairs of p-substrings of T that p-match and whose length is at least t .

On the other hand, the searching of multiple patterns has been extended to parameterized matching [81]. For a given fixed set D of p-string patterns over $\Sigma_C \cup \Sigma_P$, the *Parameterized Multiple Pattern Matching (PMPM)* problem consists of preprocessing D as an aid to later determine the p-matches (for all of the patterns in D) in a query text T . A dynamic variant of this problem, called *Parameterized Dynamic Dictionary Matching (PDDM)*, has also been considered [81]. In this problem, a dictionary D of p-string patterns is preprocessed and maintained with available operations of inserting/deleting patterns into/from D and searching a query text T for p-matches for the patterns currently in D .

2.3.2. Solutions

The maximal parameterized matching over a threshold length problem was the first parameterized matching problem to ever be considered, even before some of the basic definitions of parameterized matching were formalized. Baker tackled this problem motivated by the observation that there was a considerable amount of duplicate code in large software systems [13]. Therefore, she presented a program, called DUP, as an aid to find all the duplicate sections of code with a minimum length, specified by the user, in a large software system. DUP simplifies the problem to an exact matching problem replacing all the parameters by a determined symbol and then looks for the p-matches among the exact matches found. The algorithm is based on recursions over the suffix tree of the text.

A suffix tree of a string X is a compacted trie¹ defined on the set of the suffixes of $X\$$, where $\$$ is a unique end marker so that no suffix is a prefix of another suffix (*cf.* [101, 147, 155]). A compacted trie is a tree data structure defined on a set of strings (in this case, the set of the suffixes of a string) such that: (i) every inner node has at least two children; (ii) every edge is labelled with a substring of one of the strings in the given set; and (iii) the concatenation of the labels on the path from the root to each leaf is a distinct string in the set [41]. The key property of compacted tries is that, for every pair of leaves, the string formed by concatenating the labels from the root to their lowest common ancestor is the longest common prefix of the strings in the set associated with these leaves. For each node v of the trie, the *pathstring* of v is the concatenation of the edge labels on the path from the root to v and the length of the pathstring of v is called the *pathlength* of v .

For constructing the suffix tree for the text, Baker suggests McCreight's Algorithm [101]. This algorithm builds a suffix tree for a string $X = X_{1\dots m}$ in m stages each one of which corresponds to the insertion (from left to right) of a suffix $X_{i\dots m}$, $1 \leq i \leq m$, of X . The insertion of the i -th suffix is made so that the first part of the path coincides with the path of the longest common prefix of $X_{i\dots m}$ and $X_{j\dots m}$ for some $j < i$ (a previously inserted suffix). One of the key points to make this algorithm efficient is the use of *suffix links*. If an internal node has pathstring aX , where a is a symbol and X is a string, its suffix link points to an internal node with pathstring X (which is guaranteed to exist due to the Common Prefix Property and the Distinct Right Context Property of strings). Suffix links are also useful for pattern matching in space proportional to the size of the pattern [34].

Experiments with real data proved that DUP is highly useful in software maintenance but also showed that the algorithm is inefficient given that just a few of the found exact matches correspond to p-matches. For this reason, the same author proposed a more elaborate theory [15, 18] aiming to find better solutions and support a wider range of applications. This theory includes the definition of the parameterized pattern matching problem.

¹*a.k.a.* Multiway Patricia Trie.

Some core aspects of parameterized matching are discussed, as follows. For the case of the string comparison problem, a naive way to determine whether two length- m p-strings $X = X_{1\dots m}$ and $Y = Y_{1\dots m}$ are a p-match was proposed [18]. It consists of the following steps. Traverse both X and Y from left to right while constructing a table that establishes the mapping function that allows to transform one of the p-strings into the other one. Continue with this procedure until a mismatch is found. A mismatch between two corresponding symbols occurs in any of the following three cases: (i) one symbol is a parameter (from Σ_P) and the other is a non-parameter (from Σ_C); (ii) both symbols are non-parameters and they are different; and (iii) both symbols are parameters but any of them has previously been assigned to a different parameter in the mapping table. If no mismatch occurs, then X and Y are a p-match. The time complexity is $O(m)$ and the space complexity is $O(|\Sigma_P|)$. Nevertheless, this approach is not proper for the pattern matching problem.

A procedure called *prev* was defined to yield more efficient solutions for parameterized matching [18]. Given the constant alphabet Σ_C , the parameter alphabet Σ_P and a length- m p-string $X = X_{1\dots m}$, $prev(X)$ is a string in $(\Sigma_C \cup \mathbb{N})^*$ where every constant symbol in X remains the same in $prev(X)$ but the parameter symbols are replaced by nonnegative integers: the leftmost occurrence of a determined parameter is represented by a 0 and the other occurrences are represented by the difference in position compared to the previous occurrence of this parameter. The numbers that represent difference in position are called *parameter pointers*. The time complexity of the computation of *prev* is $O(m)$ and the space complexity is $O(|\Sigma_P|)$ by means of a table containing the last occurrence position of each parameter. Notice that $prev(X)$ is calculated in such a way that it does not matter what the parameters of X are; what is really relevant is the relative distance among the different occurrences of the same parameter (represented by the parameter pointers) which provides valuable information about the structure of the p-string. Thus, two p-strings X and Y are a p-match, iff $prev(X) = prev(Y)$.

Example. For the example presented in Figure 1-5, where $\Sigma_C = \{b\}$, $\Sigma_P = \{x, y, z\}$, $X = xbyyxbx$ and $Y = zbxzbxz$, we find that $prev(X) = 0b014b2 = prev(Y)$ and therefore X and Y are a p-match (see Figure 2-1).

The *prev* of any substring of a p-string X can be calculated from $prev(X)$ given that any symbol of the substring is the same as in $prev(X)$ except when it is a parameter pointer that points to a position before i ; in such case, it will correspond to the first occurrence of the parameter in the substring so it must be replaced by a 0. On the other hand, the parameterized pattern matching problem could be defined, in terms of *prev*, in the following manner: Given the pattern $P = P_{1\dots m}$ and the text $T = T_{1\dots n}$, both defined over $\Sigma_C \cup \Sigma_P$, P is said to parameterized-match T^i iff $prev(P) = prev(T^i)$ (recall that $T^i = T_{i\dots i+m-1}$). Thus, using *prev* is a convenient approach for the parameterized pattern matching case, given that any $prev(T^i)$ can be calculated as follows.

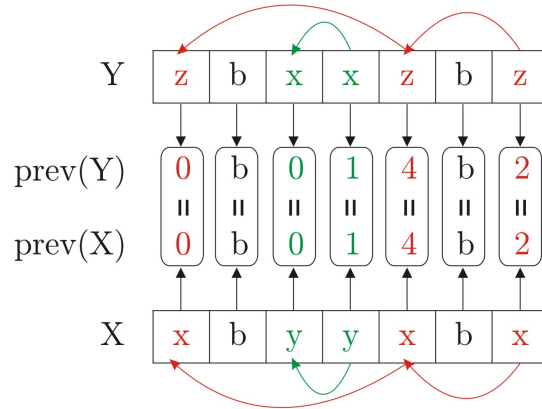


Figure 2-1.: Determination of a p-match between $X = xbyyxbx$ and $Y = zbxzbxz$ through the $prev$ procedure, where $\Sigma_C = \{b\}$, $\Sigma_P = \{x, y, z\}$.

$$prev(T^i)_j = \begin{cases} 0 & \text{if } prev(T)_{i+j-1} > j - 1 \\ prev(T)_{i+j-1} & \text{otherwise} \end{cases}, \quad \text{for } 1 \leq j \leq m.$$

In this sense, with the use of $prev$, parameterized matching can be seen as a standard exact matching problem without losing information about the chains of parameters. Reminiscing about the use of suffix trees for exact matches in DUP, Baker defined a new data structure called *parameterized suffix tree* to aid in directly searching for parameterized-matches [18]. Parameterized suffix trees are a generalization of suffix trees for strings.

To generalize suffix trees to parameterized suffix trees, it is necessary to review the definition of *p-suffix* [18]. The i -th *p-suffix* of a p-string $X = X_{1\dots m}$ is defined as $psuffix(X, i) = prev(X_{i\dots m})$. So we can calculate each p-suffix, just like the $prev$ of any substring of X , by copying the corresponding symbols of $prev(X)$ except when they are parameter pointers that point to a symbol outside the substring (in which case they are replaced by 0). Then, p-suffix trees are defined as follows. If X is a p-string that ends with a unique end marker $\$$ in Σ_C , a *parameterized suffix tree*, also called *p-suffix tree*, for X is a compacted trie (multiway Patricia trie) that stores the p-suffixes of X [18]. Following, we give an example, as it appears in [18], of the p-suffixes that the p-suffix tree of a given string must store.

Example. Let $\Sigma_C = \{b, \$\}$ be the constant alphabet, $\Sigma_P = \{x, y\}$ be the parameter alphabet and $X = xbyyxbx\$$ be a p-string. Then, $prev(X) = 0b014b2$ so the p-suffix tree of X must encode $0b014b2\$, b010b2\$, 010b2\$, 00b2\$, 0b2\$, b0\$, 0\%$ and $\$$ (see Fig. 2-2).

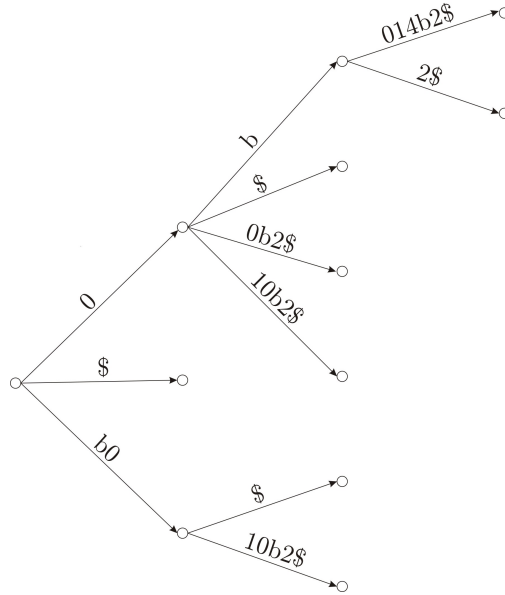


Figure 2-2.: A p-suffix tree for $X = xbyyxbx\$$ where $\Sigma_C = \{b, \$\}$ and $\Sigma_P = \{x, y\}$.

An algorithm to construct p-suffix trees, called *LAZY*, was proposed [18]. It is based on McCreight's algorithm for constructing suffix trees [101]. Nevertheless, in this case, a suffix link for a node with pathstring aX cannot point to a node with pathstring X because that node may not exist. This is because the Distinct Right Context Property does not hold for p-strings. Therefore, suffix links were redefined in such a way that, for a node with pathstring aX , the suffix link points to the node whose pathstring is the longest prefix of X among all the nodes in the tree. This algorithm is linear in the p-string length in both time and space for fixed alphabets. For variable alphabets, the time complexity is $O(n(|\Sigma_P| \log(|\Sigma_C| + |\Sigma_P|)))$.

Later, Baker proposed a new algorithm to build p-suffix trees, called *EAGER*, where the suffix links for a node with pathstring aX point to the node whose pathstring is the shortest of all those for which X is a prefix [15]. This idea is more convenient for the structure of p-suffix trees. Even though for fixed alphabets the time and space complexity remain linear, for variable alphabets the time complexity is $O(n(|\Sigma_P| + \log(|\Sigma_C| + |\Sigma_P|)))$. Nevertheless, for both *LAZY* and *EAGER*, the time complexity of the variable alphabet case can be reduced to $O(n \log n)$ by using auxiliary data structures like concatenable queues [2] and Sleator-Tarjan dynamic trees [137]. However, the use of these structures makes the algorithms not practical.

Other authors have worked on developing faster algorithms for constructing p-suffix trees. Kosaraju proposed an algorithm whose time complexity is $O(n \log(|\Sigma_P| + |\Sigma_C|))$ [90]. Furthermore, a randomized algorithm to construct suffix trees for cases where there are missing suffix links, such as p-suffix trees and suffix trees for two-dimensional arrays, was proposed [41]. It was the first algorithm whose time complexity is $O(n)$ even for variable alphabets. It is based on adding

a back-propagation component to McCreight's Algorithm and using a high probability hashing scheme for large degrees.

Two solutions for the parameterized matching problem that use p-suffix trees were developed [15]. Given the pattern p-string $P = P_{1\dots m}$ and the text p-string $T = T_{1\dots n}$, one of the algorithms consists of following the path determined by the symbols of $prev(P)$ on the p-suffix tree of $T\$$ to find out if $prev(P)$ is identical to a length- m substring of T . For fixed alphabets, to determine all the positions in T where there is a p-match with P takes $O(m + occ)$ time and $O(n)$ space, where occ is the number of p-matches. For variable alphabets, the time complexity is $O(m \log(|\Sigma_C| + |\Sigma_P|) + occ)$. The other algorithm consists of searching in a p-suffix tree for P through an adaptation of the corresponding algorithm for strings [33]. Its space complexity is $O(m)$ and its time complexity is $O(n)$ for fixed alphabets; for variable alphabets, its time complexity is $O(n(|\Sigma_P| + \log(|\Sigma_C| + |\Sigma_P|)))$. Nevertheless, it could also be improved to $O(n \log(|\Sigma_C| + |\Sigma_P|))$ by using some auxiliary data structures for computing lowest common ancestors [75, 130].

On the other hand, an algorithm, called PDUP, for finding the maximal p-matches over a threshold length of a text $T = T_{1\dots n}$ was devised [18]. PDUP is similar to DUP, but constructs a p-suffix tree of the text instead of a suffix tree. This algorithm generalizes to p-strings the algorithm for finding maximal p-matches over a threshold length in a string [14]. In this generalization, it is necessary to augment the p-suffix tree with lists that store valuable data that makes possible to determine whether there is left-extensibility in the p-matching substrings. The time complexity of PDUP is $O(n + occ)$, where occ is the number of maximal p-matches found, even for variable alphabets.

Soon after Baker proposed the parameterized matching theory and its first algorithms, other researchers started to work on this topic. For instance, Amir *et al.* analysed Baker's theory and defined a related model called *Mapped Matching* which is a special case of parameterized matching where all symbols are in the parameter alphabet Σ_P [5]. Through this model, an algorithm that extends the KMP algorithm [89] to parameterized matching and runs in $O(n \log \min(m, |\Sigma_P|))$ time was proposed [5]. This was the first parameterized matching algorithm independent from the size of the constant alphabet Σ_C . Furthermore, it was proven that the $\log \min(m, |\Sigma_P|)$ factor is inherent to any algorithm for parameterized matching in the comparison model and, consequently, that the provided algorithm is optimal. This demonstration was achieved through a reduction from the element distinctness problem to parameterized matching.

This new research may have motivated Baker to look for parameterized matching solutions based on classical exact string matching algorithms [16]. Given that the BOYER-MOORE algorithm [26] is one of the most efficient, she attempted to generalize it to p-strings but found its worst case performance was poor. Therefore she turned to one of its variants, TURBOBM [49]. Her non-trivial generalization of TURBOBM to p-strings, called PTURBOBM, runs in $O(n \log \min(m, |\Sigma_P|))$ time and $O(n)$ space; the preprocessing time is $O(m \log \min(m, p))$. Its time complexity is the sa-

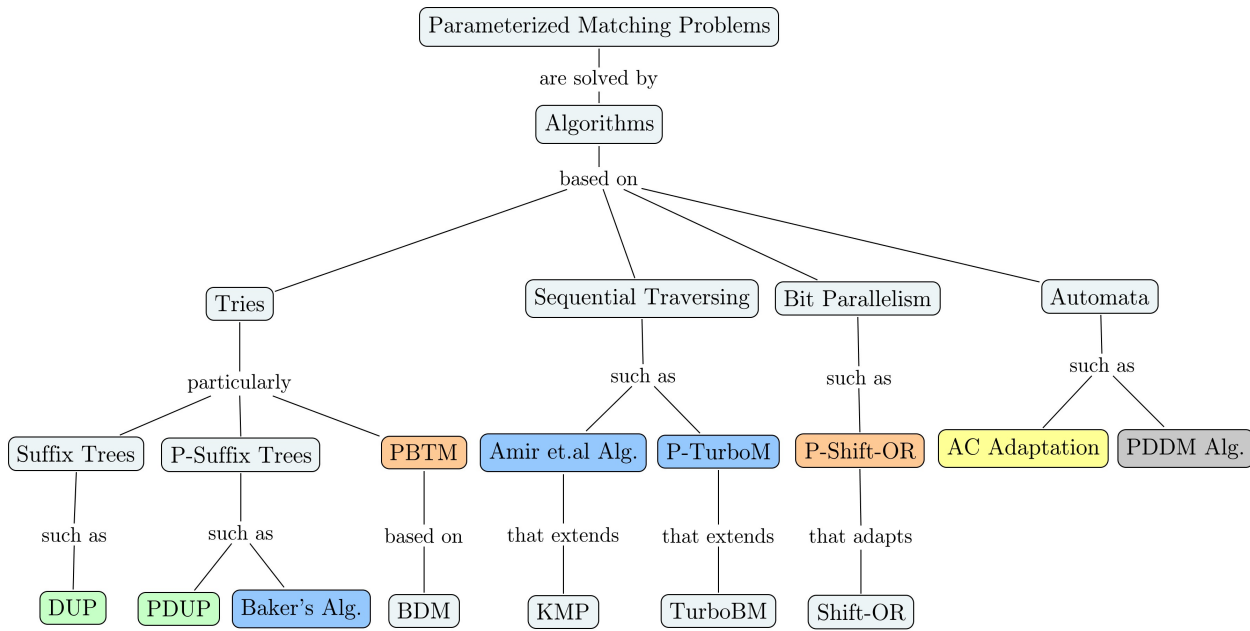
me as the generalization of KMP complexity so it is optimal [5]. Nevertheless some experiments show that PTURBOBM works better for long patterns over different alphabet sizes. Anyhow, for variable alphabets, both of these algorithms are notably better than then p-suffix tree based parameterized matching algorithms.

Other important contributions were made by Idury and Schäffer who proposed some variants of the basic problem (see Section 2.3.1) and solutions for all of them [81]. For the Parameterized Multiple Pattern Matching Problem, they proposed an algorithm that uses a modified Aho–Corasick automaton and runs in $O(n \log(|\Sigma_C| + |\Sigma_P|) + occ)$ time, where occ is the number of occurrences of all the patterns. As for the Parameterized Dynamic Dictionary Problem, they devised an automaton algorithm that supports different operations with the following time complexity: (i) $O((n + occ)(\log(|\Sigma_C| + |\Sigma_P|) + \log d))$ for searching the p-string patterns of the dictionary in a p-string text $T = T_{1..n}$; (ii) $O(m(\log(|\Sigma_C| + |\Sigma_P|)) + \log^2 d)$ for inserting a new pattern $P = P_{1..m}$ into the dictionary; and (iii) $O(m(\log(|\Sigma_C| + |\Sigma_P|)) + \log d)$ for deleting a pattern $P = P_{1..m}$ from the dictionary, where d is the total size of all the patterns.

More recently, Fredriksson and Mozgovoy proposed two new algorithms for both the single and multiple parameterized matching problems [68]. Both of them make use of Baker’s lemma to compute the *prev* of a text substring through the *prev* of the container p-string [18]. One of them is a bit-parallelism based algorithm called P-SHIFT-OR. It is a generalization of the SHIFT-OR algorithm [12] to p-strings and runs in $O(n \lceil m/w \rceil)$ worst case time and $O(n)$ average time. This algorithm can be extended to solve the multiple parameterized matching problem.

Fredriksson and Mozgovoy also devised an algorithm called Parameterized Backward Trie Matching (PBTM) [68] based on the Backward DAWG Matching (BDM) algorithm [25, 49]. First, the set of p-suffixes of the reverse of the pattern are stored in a trie. Then, this trie is used to fastly determine whether there is a p-match in the current text window; otherwise, the trie is used to calculate the length of the shift to consider the next text window where a p-match could be found. The average time complexity of PBTM is $O(n \log(m)/m)$. This process could also make use of a suffix array [99] instead of a trie, in which case the algorithm is called Parameterized Backward Array Matching (PBAM). PBTM and PBAM are also extensible for the multiple parameterized matching problem. It is remarkable that these algorithms are the first parameterized matching algorithms for which an average time complexity analysis has been made. They have optimal average case running for both single and multiple patterns, as confirmed by experimental results.

The diagram in Figure 2-3 shows the algorithms for solving the different parameterized matching problems presented in this section organized by the nature of their approaches.



The background color of each algorithm indicates which Parameterized Matching problem it solves, as follows:

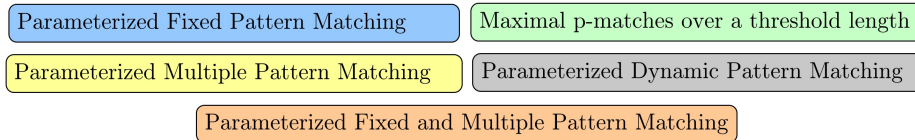


Figure 2-3.: Concept map of the algorithms for solving the main parameterized matching problems organized by the nature of their approaches.

2.3.3. Extensions

Parameterized Matching has been studied in many directions. For instance, an investigation about the periodicity of parameterized strings was done [9]. They attempted to generalize to p-strings two of the periodicity lemmas of strings: the Lyndon and Schützenberger lemma (referred as *Weak Version*) [97], and the Fine and Wilf lemma [63]. They found out that only the Weak Version holds for p-strings only when the two mappings inducing the periodicity commute. These results and some other studies about the repetitions in p-strings showed considerable differences between p-strings and ordinary strings. Nevertheless, binary p-strings behave in a very similar way as ordinary strings with respect to periodicity and repetitions.

On the other hand, parameterized matching was extended to the two dimensional case by considering matrices of symbols instead of p-strings. *Two-dimensional parameterized matching* consists of finding all the p-matches of a pattern of size $m \times m$ in a text of size $n \times n$. An algorithm for the problem that runs in $O(n^2 + m^{2.5} \text{polylog } m)$ time was proposed [76]. Other solutions include a

$O(n^2 \log^2 m)$ deterministic algorithm and a $O(n^2 \log n)$ randomized algorithm that reports all the p-matches [4]. Nevertheless, it may report a mismatch as match with probability of $1/n^k$, where k is a given constant.

Other topic that arose as a matter of interest was the calculation of similarity between two p-strings. In particular, Baker defined the *parameterized edit distance* or *p-edit distance* of two p-strings as the cost of a minimal edit script, called *p-edit script*, that transforms one p-string into the other [19]. The valid operations are insertions, deletions and parameterized replacements (the replacement of a substring with a p-string that p-matches it). Moreover, Baker proposed an algorithm [19] for calculating the p-edit distance D of two *prev*-encoded p-strings, $X = X_{1\dots m}$ and $Y = Y_{1\dots n}$, by generalizing Myers's algorithm for finding the LCS of two strings [116]. The algorithm runs in $O(D(n + m))$ time and $O(n + m)$ space. However, the complexities can be improved by using p-suffix trees [18] and the lowest common ancestor [75, 130]. Furthermore, a divide-and-conquer based algorithm for reporting the minimal p-edit script was proposed [19]. It also runs in $O(D(n + m))$ and $O(n + m)$ space. Finally, it is shown that these techniques can be extended to solve the *approximate parameterized problem under the p-edit distance* [19], defined as follows. For a given p-string pattern $P = P_{1\dots m}$, a p-string text $T = T_{1\dots n}$ and an integer k , the goal is to report all the positions $1 \leq i \leq n$ such that T^i are within the p-edit distance k of P . This can be done in $O((k + \log|\Sigma_C| + \log|\Sigma_P|)(n + m))$ time and $O(n + m)$ space.

There have been some works about *approximate parameterized problem under hamming distance*. In particular, the π -match between two p-strings $X = X_{1\dots m}$ and $Y = Y_{1\dots m}$ was defined as the number of matches between $\pi(Y_i)$ and X_i , for $1 \leq i \leq m$ [7]. For two equal-length p-strings, the *approximate parameterized matching problem*, also called *parameterized matching with mismatches*, consists of finding a π of maximal π -match. Given a p-string pattern $P = P_{1\dots m}$ and a p-string text $T = T_{1\dots n}$, the *approximate parameterized searching problem under hamming distance* consists of computing the approximate parameterized matching between P and every length- m p-substring of T . It is not necessary to choose the same π for every text window, as in standard parameterized matching. Furthermore, a linear algorithm to solve this problem, for the case where both P and T are run-length encoded and one of them is a binary p-string, was devised [7].

Further studies about parameterized matching and hamming distance have been developed [77, 76]. Specifically, a related problem, called *parameterized matching with a threshold of k mismatches*, was proposed. Its goal is finding all the p-matches of a pattern $P = P_{1\dots m}$ in a text $T = T_{1\dots n}$ with at most k mismatches. For two equal-length p-strings $X = X_{1\dots m}$ and $Y = Y_{1\dots m}$, they proposed a $O(m + k^{1.5})$ time algorithm and a $O(m^{1.5})$ time algorithm for the cases when k is considered and when it is not considered, respectively. These solutions are based on maximum matching algorithms; furthermore, it was demonstrated that the maximum matching problem is reducible to the approximate parameterized matching problem. For a p-string pattern $P = P_{1\dots m}$, a p-string text $T = T_{1\dots n}$ and a given k , a $O(nk^{1.5} + mk \log m)$ time algorithm for the parameterized matching

with k mismatches problem was also proposed. It is shown that this could be extended to the two dimensional case in $O(n^2mk^{1.5} + m^2k \log m)$ time.

Another approximate version of parameterized matching is based on δ - and γ - distances. Specifically, we defined $\delta\gamma$ -approximate parameterized matching [93, 103]. Given two equal-length integer strings $X = X_{1\dots m}$ and $Y = Y_{1\dots m}$, string X is said to $\delta\gamma$ -parameterized match string Y if X can be transformed in a string X' , via a bijection π (i.e., $X'_i = \pi(X_i)$ for $1 \leq i \leq m$), such that X' $\delta\gamma$ -matches Y . Constants δ and γ are bounds for the local and global errors, respectively, on the difference between the corresponding symbols of the strings. A $O(nm)$ algorithm to report the $\delta\gamma$ -parameterized matches of a pattern $P = P_{1\dots m}$ in a text $T = T_{1\dots n}$ was proposed [93, 103]. This variant is defined as a combination of two string matching paradigms: parameterized matching and $\delta\gamma$ -matching. The latter, is very effective in searching for all similar but not necessarily identical occurrences of a given pattern. This problem has been well-studied (cf. [30, 50, 48]) due to its applications in bioinformatics [107, 111] and music information retrieval [30].

The parameterized matching problem under the LCS distance problem has also been considered. The *longest common parameterized subsequence (LCPS)* for two p-strings $X = X_{1\dots m}$ and $Y = Y_{1\dots n}$ was defined as the pair of sequences I and J of maximum length, such that I is a subsequence of the p-string X , J is a subsequence of the p-string Y , and I and J are a p-match [87]. It is important to remark that it is not required that the symbols in I and J are consecutive in X and Y . The LCPS could be useful as a similarity measure between code sections; nevertheless, this problem has been proven to be NP-hard. Then, an approximate algorithm was proposed [87]. On the other hand, in [82] some algorithms for computing the longest parameterized common subsequences are presented; nevertheless it is important to mention that their definition of parameterization is considerably different from the one developed by Baker and tackled in this thesis.

Another parameterized paradigm, called *parameterized pattern queries*, that does not correspond to Baker's initial definition, was proposed [53]. However, this model is indeed closely related to the theory developed by Baker. They use a set of symbols and a set of variables that correspond to Baker's constant alphabet and parameter alphabet. They also defined a concept of *valuation* that could be associated with the mapping bijection and the p-match definition. The parameterized pattern queries paradigm was conceived as an extension of traditional pattern expressions to enhance the querying and clustering operations over sequence databases. Thus, the definition of a set of predicates on the variables (constraints) is also permitted under this new model. Furthermore, a KMP-based algorithm for this problem is also proposed. Experimental results showed that it notably decreases the query evaluation time compared to a naive approach.

In order to support more applications, parameterized matching was generalized to *function matching* by allowing the mapping function to be of any type, and not just bijections as in parameterized matching [4]. In other words, many symbols of the pattern can be mapped to the same text

symbol. A deterministic solution for the function matching problem, that runs in $O(n|\Sigma_P| \log m)$ time, was devised [4]. Furthermore, they proposed a Monte Carlo algorithm that runs in $O(n \log m)$ time with failure probability of $1/n^k$, where k is a given constant. Function matching was also extended for the two-dimensional case and a randomized algorithm that runs in $O(kn^2 \log n)$ time was proposed [4]. This algorithm has a $1/n^k$ probability of reporting a false positive.

We derived an approximate version of function matching to permit certain degree of error. In particular, we proposed $\delta\gamma$ -approximate function matching [106]. Given two integer strings, $X = X_{1\dots m}$ and $Y = Y_{1\dots m}$, and two given constants, $\delta, \gamma \in \mathbb{N}$, we say that there is a match from X to Y if X can be transformed into a string X' , by means of a function f , such that X' is δ -equal and γ -equal to Y . Two equal-length integer strings are δ -equal if the maximum difference between their corresponding symbols is at most δ ; they are γ -equal if the sum of such differences is at most γ . A $O(nm)$ algorithm to find the $\delta\gamma$ -function matches of a pattern $P = P_{1\dots m}$ in a text $T = T_{1\dots n}$ was proposed [106].

To support even a much wider range of applications, function matching was extended to the *generalized function matching with don't cares* problem [6]. In this problem, the image of the mapping function can be any substring in $(\Sigma_C \cup \Sigma_P)^*$ and not just a single symbol as in function matching. Furthermore, an extra symbol ϕ , called the don't care character, can be present in the strings. A ϕ in the text matches any pattern symbol; a ϕ in the pattern matches any text substring. This problem represents many pattern searching types but, as a consequence, it is much more complex. It was shown that the alphabet sizes appear in the exponent of the naive solution's complexity which leads to a considerable difference between the cases of finite and infinite alphabets. A polynomial algorithm for the finite alphabet case was presented; for the case of infinite alphabets, it was demonstrated that the problem is *NP-hard* [6]. This is the first problem, so far, for which there is a polynomial solution for the finite alphabet case and there is not one for the infinite alphabet case.

2.3.4. Applications

Parameterized matching was initially defined as a tool for software maintenance [13]. This was motivated by the observation that programmers introduce duplicate code into large software systems when they are adding new features or fixing bugs possibly generated for not having considered special cases in the initial programming. Instead of adapting working sections of code, the programmers prefer to copy and slightly modify new instances of those sections in order to avoid making major revisions and introducing new bugs. They do it specially when the working sections were written by another programmer. With time, the amount of duplicate code is highly increased and the code gets larger, more complex and more difficult to maintain. For instance, when a new issue in a determined part of the program is fixed, it will not be automatically fixed in the other copies of that section of code and sometimes they may be hard to find.

The definition of parameterized matching assumes that some sections of code are copied and modified through text editors such that the corresponding copies are mostly the same, except for a systematic change of the variables and procedures' names. Then, the code is considered as a sequence of tokens (variables, constants, operands, reserved keywords and procedure names) where the constant alphabet Σ_C is comprised by the operands and the reserved keywords while the parameter alphabet Σ_P is comprised by the variables, constants and procedures' names.

Example. In Fig. 2-4 two edited code excerpts from the X Window [129] source code are presented. These two fragments are a p-match given that they are identical except for a correspondence between *pfi* and *pfh*, *lbearing* and *left*, and *rbearing* and *right*. Notice that the p-matching sections are like expansions of the same macro with different parameters.

```

copy_number(&pmin, &pmax, pfi->min_bounds.lbearing,
            pfi->max_bounds.lbearing);
*pmin++ = *pmax++ = ',';
copy_number(&pmin, &pmax, pfi->min_bounds.rbearing,
            pfi->max_bounds.rbearing);
*pmin++ = *pmax++ = ',';

copy_number(&pmin, &pmax, pfh->min_bounds.left,
            pfh->max_bounds.left);
*pmin++ = *pmax++ = ',';
copy_number(&pmin, &pmax, pfh->min_bounds.right,
            pfh->max_bounds.right);
*pmin++ = *pmax++ = ',';

```

Figure 2-4.: Two sections of code that parameterized-match. Taken from [15].

In that sense, if it is required to look for all the copies of a determined section of code, the problem can be solved through a *parameterized fixed matching* algorithm by considering that section of code as the pattern and the code where copies are searched as the text. If the goal is finding all the copies of many sections of code (multiple patterns), then a *parameterized multiple matching* algorithm would be useful. If there exists no pattern, but the goal is finding all the pairs of duplicate code that have at least a determined length specified by the user, then the problem can be solved with a *maximal p-matches over a threshold length* algorithm, like DUP or PDUP.

Experiments with DUP on a large subsystem of over a million lines of code showed that 22% of the lines were involved in parameterized matching. This is a great amount of duplicate code, given that a proportional percentage of the code could be shrunk by using better programming techniques like procedures and functions. A reduction of this magnitude would make the code much more simple and easier to maintain. In general, all the parameterized matching problems and the approximate parameterized matching problems (under the p-edit and hamming distance) produce important results that facilitate the analysis of the code and provide useful information to simplify

it and shrink it. This is the reason why software maintenance is still one of the main areas where parameterized matching is most useful at.

Other area of application of parameterized matching is image processing [77, 4]. Searching for color images on the web is an interesting problem [11, 139]. The Human–Computer Interaction Lab at the University of Maryland tackled the problem of searching for an icon in the screen. If the colors are fixed, the problem can be solved with an exact two-dimensional pattern matching algorithm. Nevertheless, sometimes the pattern image appears in other ranges of colors within the text, which makes impossible for exact–matching algorithms to find these occurrences. In this kind of cases it is proper to use two dimensional *parameterized matching* algorithms. However, images often have some errors resulting from distortion and loss of resolution, so such occurrences of a pattern image could not be reported by parameterized matching algorithms either (due to the absence of perfect bijections). But occurrences with these errors can indeed be found by taking either a function matching approach [4, 106] or an approximate parameterized matching approach under the hamming, p–edit, or $\delta\gamma$ distance [19, 76, 77, 93, 103].

On the other hand, parameterized matching has important applications in databases. For instance, in a database that contains urls of the pages visited by different users, parameterized pattern queries can be used to retrieve useful information for improving the ergonomoy of the site and finding the best places for advertisement ads [53]. For example, given the symbol a and the variable x where both represent urls, the query of the parameterized pattern expression axa would retrieve the set of urls that the users have visited before coming back to the previously visited page represented by a . In a similar fashion, this idea can be used in computational biology to retrieve all the amino acids substrings that follow a determined structure where the presence of determined amino acids at certain positions are a constraint. This is also applicable to databases of any type, where the analysis over the sequential occurrence of elements is a matter of interest.

In general, parameterized matching and its related problems are considerably useful in any area where patterns are defined in terms of structural correlation across the positions. This motivates us to extrapolate its use to the solution of graph matching.

Part I.

**Graph Isomorphism through
Parameterized Matching**

3. Our Approach: Graph Linearization

Our approach to determine whether two graphs, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, are isomorphic consists of two main steps: (i) linearizing G_1 into a walk $p = p_{1\dots\ell}$; and (ii) exploring all the walks in G_2 to determine whether there is one that parameterized-matches $p = p_{1\dots\ell}$. In this chapter, we define graph linearization and parameterized matching on graph walks (Section 3.1). Then, we discuss characteristics and algorithms for linearization (Section 3.2). Finally, we propose an efficient algorithm that produces asymptotically length-optimal linearizations (Section 3.3).

3.1. Definition of Graph Linearization

Definition 2 (Graph Linearization). *Let $G = (V, E)$ be a connected undirected multigraph. A walk $p = p_{1\dots\ell}$ of nodes and edges is a linearization of G iff:*

1. p_i is a node $v \in V$ if i is odd, $1 \leq i \leq \ell$.
2. p_i is an edge $e \in E$ if i is even, $1 \leq i \leq \ell$, such that $e = (p_{i-1}, p_{i+1})$.
3. Each node $v \in V$ and each edge $e \in E$ appears at least once in p .

In other words, the linearization p of a connected undirected graph $G = (V, E)$ is an alternating sequence of nodes $v \in V$ and edges $e \in E$ that starts and ends at a node. Each intermediate occurrence of a node in p must be preceded and followed by an adjacent edge. Furthermore, all nodes and edges in the graph must appear in p at least once.

Our motivation for defining graph linearization is representing the topology of a multigraph through a walk. Specifically, the linearization p of G is a walk that represents all its adjacency relation, which we use to solve the graph isomorphism problem by comparing walks instead of multigraphs. For this purpose, we define parameterized matching on walks as follows:

Definition 3 (Parameterized Match on Graph Walks). *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two connected undirected multigraphs. Also, let $V'_1 \subseteq V_1$ and $E'_1 \subseteq E_1$ be subsets of nodes and edges in G_1 ; similarly, $V'_2 \subseteq V_2$ and $E'_2 \subseteq E_2$ are subsets of nodes and edges in G_2 . The walks $p = p_{1\dots k}$, in G_1 , and $q = q_{1\dots k}$, in G_2 , are said to parameterized-match if and only if there exists a bijective function $f : (V'_1 \cup E'_1) \rightarrow (V'_2 \cup E'_2)$ such that $q_i = f(p_i)$ for $1 \leq i \leq k$.*

The core idea of using parameterized matching to solve the graph isomorphism problem is as follows. Let p be a linearization of G_1 ; hence, p represents the topology of G_1 . Thus, if a walk q in G_2 parameterized-matches p , then p and q have the same topology. Consequently, considering that q represents G_2 , we conclude that G_1 and G_2 are isomorphic. More formally, we prove the following theorem that solves the graph isomorphism problem through parameterized walks.

Theorem 1. *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two connected undirected multigraphs such that $n = |V_1| = |V_2|$ and $m = |E_1| = |E_2|$; also, let $p = p_{1\dots\ell}$ be a linearization of G_1 . Then, G_1 and G_2 are isomorphic if and only if there exists a walk $q = q_{1\dots\ell}$ in G_2 such that $p = p_{1\dots\ell}$ parameterized-matches $q = q_{1\dots\ell}$.*

Proof. In order to prove the theorem, we need to show that (i) if G_1 and G_2 are isomorphic, then there exists a walk $q = q_{1\dots\ell}$ in G_2 that parameterized-matches $p = p_{1\dots\ell}$; and (ii) if there exists a walk $q = q_{1\dots\ell}$ in G_2 that parameterized-matches $p_{1\dots\ell}$, then G_1 and G_2 are isomorphic.

First we prove (i). According to Problem 1, if G_1 and G_2 are isomorphic, there exists a bijective function $f : \mathcal{E}_{G_1} \rightarrow \mathcal{E}_{G_2}$ for which Equation 1-1 is evaluated as *true*. Notice that $p = p_{1\dots\ell}$ represents all the adjacency relation of G_1 , which is defined on the left side of the biconditional. Considering the format of $p_{1\dots\ell}$ (see Definition 2) and the existence of a bijective function f that satisfies the right side of the biconditional, we can conclude that $q = f(p_1)f(p_2) \cdots f(p_\ell)$ is a walk in G_2 . Furthermore, as $f(p_i) = q_i$, walks $p = p_{1\dots\ell}$ and $q = q_{1\dots\ell}$ parameterized-match.

Now we prove (ii). Let $q = q_{1\dots\ell}$ be a walk in G_2 that parameterized-matches $p = p_{1\dots\ell}$. Then, there exists a bijective function $f : \mathcal{E}_{G_1} \rightarrow \mathcal{E}_{G_2}$ such that $q_i = f(p_i)$ for all $1 \leq i \leq \ell$. Recall that all the nodes in V_1 and all the edges in E_1 appear at least once in p (see Definition 2) and that all the adjacency relation of G_1 is represented in $p = p_{1\dots\ell}$. Therefore, the existence of a bijective function f such that $q_i = f(p_i)$, for all $1 \leq i \leq \ell$, implies that Equation 1-1 is evaluated as *true*; then, G_1 and G_2 are isomorphic. \square

Corollary 1. *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two connected undirected multigraphs where $|V_1| \leq |V_2|$. Also, let $p = p_{1\dots\ell}$ be a linearization of G_1 . Then, G_1 is isomorphic to a subgraph in G_2 if and only if there exists a walk $q = q_{1\dots\ell}$ in G_2 such that $p = p_{1\dots\ell}$ parameterized-matches $q = q_{1\dots\ell}$.*

Proof. A walk $q = q_{1\dots\ell}$ on G_2 is a linearization of a subgraph $G = (V, E)$ of G_2 where V and E are the sets of nodes and edges, respectively, included in $q = q_{1\dots\ell}$ (see Definition 2). Then, because of Theorem 1, G , a subgraph of G_2 , is isomorphic to G_1 . \square

3.2. Characteristics and Algorithms for Graph Linearization

There may be many linearizations that represent the same graph. Many factors such as different starting nodes and different visiting orders can result in different linearizations. However, a compact representation is preferable. For solving graph isomorphism, the length of the linearization is an important measure on the matching time. This is because a shorter linearization often leads to a smaller cost at the matching stage. Next, we define *length-optimal linearization*.

Definition 4 (Length-Optimal Linearization). *The linearization $p = p_{1\dots\ell}$ of a connected undirected multigraph is length-optimal if the length of p (i.e., ℓ) is minimum.*

Finding the linearization of a graph is very similar to the *Chinese Postman Problem* (CPP). CPP finds a walk that visits all the edges (and all the nodes) in the multigraph at least once; the only difference is that graph linearization does not require the starting node to be the same final node. A $O(n^3 + m^2)$ algorithm for the CPP was proposed [54]. We can adapt this algorithm to calculate a length-optimal linearization. However, for large multigraphs, it is desirable to have algorithms with lower time complexity even if they do not produce length-optimal linearizations. Then, we consider BFS-L and DFS-L which produce linearizations using the graph traversal algorithms, as follows:

- BFS-L traverses the multigraph in the breadth-first search manner with some differences. When it explores a node, it visits all of its unexplored adjacent edges regardless if the node they lead to has been visited before. After discovering such nodes, it goes back to each one of them to (i) check if it has unexplored adjacent edges; and (ii) explore such unexplored edges (if there is any). But it is necessary to add in p all the walk that connects the current node and the next node to be explored; this significantly increases the length of the linearization walk. The algorithm terminates when there are no discovered nodes to be explored.
- DFS-L traverses the multigraph in the depth-first search manner. It starts from any node and makes recursive calls to visit the adjacent nodes that are connected to it through unexplored edges. The base case of the recursive call is when it reaches a node whose edges are all explored; then, it returns directly. Every time a recursive execution is finished, the execution instance that called it must add the corresponding connecting edge and source node in p again; this is because they need to be visited again so that other neighbours can be explored. The procedure finishes when it goes back to the starting node and it does not have any unexplored edges.

Notwithstanding, the linearizations produced by these algorithms can be long. As an attractive trade-off between length-optimality and efficiency, we propose a greedy approximation algorithm with an approximation guarantee.

3.3. Graph Linearization Algorithm - GLA

This section presents the GLA or *Graph Linearization Algorithm*. First, we describe the key ideas of the algorithm in Section 3.3.1; then we go through the details in Section 3.3.2. Its correctness is proven in Section 3.3.3. In Section 3.3.4 we present an upper bound for the length of GLA linearizations; furthermore, we show with empirical examples that, in practice, the produced linearizations are close to optimal (see Section 3.3.5). Finally, in Section 3.3.6, we present the complexity analysis.

3.3.1. Key Ideas

One of the challenges of linearization algorithms is visiting all the edges with short linearization length. Given that, in order to visit an edge, it is necessary to visit first one of its end nodes, we use the number of unexplored adjacent edges that the nodes have to conduct the traversal in a convenient manner. Particularly, GLA does the traversal in a similar way as DFS-L. The base case of the traversal is when we reach a node that has no unexplored adjacent edges; in this case, we say that such node is *covered*. In GLA, the number of uncovered nodes is stored. Then, if all the nodes are covered when a base case takes place, we do not add the way back up to the root of the DFS tree in the linearization.

Furthermore, GLA takes into account the number of unexplored edges of the nodes, at all stages of the process, to produce shorter linearizations. Specifically, we develop three heuristics: (i) the traversal starts from the node with the lowest degree; (ii) the unexplored edges that lead to already explored nodes are visited before than the ones that lead to unexplored nodes; and (iii) the edges that lead to unexplored nodes are considered sorted, in ascending order, on the number of unexplored edges they have. Heuristics (i) and (iii) aim to put the nodes that are close to be covered in the top levels of the DFS tree. On the other hand, heuristic (ii) aims to cover the nodes in the highest levels of the DFS tree at an early stage.

The first insight is that we want to cover the nodes at the top of the DFS tree early. This is because when the base case of the recursion is reached, going back to a node in a higher level of the DFS tree makes the linearization longer than going back to lower levels; thus, we want to reduce the probability of needing to go back to a high level. The nodes with low degree are more likely to be covered early; thus our first heuristic chooses the node with the lowest degree as the starting node of the traversal (the root of the DFS tree). The next level of the DFS tree are the root's adjacent nodes that we discover first. And we want to cover them rapidly as well. Therefore, heuristic (iii) prefers the edges $e = (u, v)$ that lead to unexplored nodes v , sorted on the number of unexplored edges of v . Then, the nodes at the top levels of the DFS tree will be the ones with the lowest degrees, as the ones with the highest degrees may be covered in a lower level of the tree. Furthermore, to increase the probability of early covering the nodes at the top of the tree, we use heuristic (ii).

More specifically, heuristic (ii) works as follows. When a node u is processed, all its unexplored edges e that lead to already explored nodes w are visited first. Notice that we do not want to continue the traversal on w as it is already being processed in a higher level of the DFS tree. All we need is exploring e ; however this implies visiting w again, so that the properties of the linearization are satisfied. Then, we visit e and w and, in order to return to u , we visit e and u again. Notice that this decreases the number of unexplored edges of w . Therefore, it is more likely that w is covered before starting all the way back up on the DFS tree when the base case of the recursion takes place. Furthermore, we do not need to add such way back into p if there are no other unexplored graph elements. In practice, this condition is satisfied quite often due to the combination of the three heuristics. Notice that these heuristics make the traversal explore one region of the multigraph before visiting another one; then, the produced linearization is shorter.

3.3.2. Algorithm

The pseudocode of the Graph Linearization Algorithm (GLA) is listed in Figure 3-1. Each graph element has a boolean attribute that indicates whether it has been explored. Furthermore, the number of unexplored graph elements is stored by variable *unexplGE*. This variable is used to avoid reinserting graph elements in the backtracking of the DFS search tree when there are no unexplored elements left. The number of unexplored adjacent edges that each node v has is stored in $v.NumUnexplEdges$. The produced linearization is implemented as the list p . These variables are used during the DFS-like traversal to apply the heuristics presented in the last section. When a graph element is inserted into p for the first time, it has to be set as *explored* and the number of unexplored graph elements in the graph, *unexplGE*, must be decreased in one unit. Furthermore, if such graph element is an edge, the number of unexplored edges of its adjacent nodes must also be decreased.

The algorithm starts by setting every graph element as *unexplored*. Then, for each node, the number of unexplored edges is calculated as the total number of adjacent edges it has. The walk p is set as empty and the number of unexplored graph elements is calculated as the sum of the number of nodes and the number of edges in the graph. After these initializations, a DFS-like traversal is performed using the recursive function in TRVERSEGRAPH() (see Figure 3-2) starting from the node u with the lowest degree, i.e. the first call of the procedure TRVERSEGRAPH() is performed over u . This procedure is composed of the following steps:

1. Add u into p .
2. Go to already explored nodes v through unexplored edges e . Add e and v into p . In case there are still unexplored graph elements, it goes back to u through e ; this implies adding e and u into p again.

3. Go to unexplored nodes v through unexplored edges e . The nodes v must be considered sorted on their number of unexplored edges. Then e is added into p and the recursive procedure `TRAVERSEGRAPH()` is called over v . If there are still unexplored graph elements after this call, then e and u must be added again into p so that other neighbours of u can be visited. When there are no other neighbours to explore, the method returns so the linearization continues at a higher level of the DFS tree.

Notice that Step (3) is represented in Figure 3-2 through lines 8 – 14. In such lines it seems that, at each iteration, we find the node v with the minimum $NumUnexploredEdges$; however, this is presented in this way just for clarity. Instead, we can sort the couples $e = (u, v)$ on $v.NumUnexploredEdges$ before the loop. Then, in each iteration, we consider each of such couples $e = (u, v)$ in ascending order on $v.NumUnexploredEdges$. The only additional operation we perform in each iteration is checking whether the adjacent node is still uncovered. This is because the value of each $v.NumUnexploredEdges$ does not change, unless such v' is covered. Specifically, if v is explored at a lower level of the DFS tree traversal, we will not come back to this loop until v is covered. Checking if a node is covered takes constant time. Consequently, the complexity of this operation is the initial sorting which is $O(d \lg d)$ where d is the maximum degree of the nodes in V .

The algorithm terminates when the first call to the recursive procedure finishes, i.e., when it goes back to the root of the DFS tree and there are no unexplored adjacent edges. At this point, $unexplGE = 0$ and p contains the linearization. The linearization produced by GLA for the graph presented in Figure 1-3(a) is $Ae_1Be_3Ce_4De_5Ce_5De_2Be_2De_6E$; its length is 17.

Algorithm 1: GLA Algorithm

Input: $G = (V, E)$

Output: p

1. **for every** $e \in E$ **do** $e.Explored \leftarrow false$
 2. **for every** $v \in V$ **do**
 3. $v.Explored \leftarrow false$
 4. $\mathcal{S} \leftarrow \{(u, v) \mid u \in V \wedge (u, v) \in E\}$
 5. $v.NumUnexploredEdges \leftarrow |\mathcal{S}|$
 6. **choose** $u \in V_P$ **with** $min(u.NumUnexploredEdges)$
 7. $p \leftarrow \langle \rangle$, $unexplGE \leftarrow |V| + |E|$
 8. $TraverseGraph(G, u, p, unexplGE)$
 9. **return** p
-

Figure 3-1.: GLA algorithm.

Algorithm 2: TRAVERSEGRAPH() **Procedure****Input:** $G = (V, E), u, p, unexplGE$

-
1. $p.Add(u), u.Explored \leftarrow true, unexplGE--$
 2. **for every** $e \in E$ **such that** $e = (u, v)$ **do**
 3. **if** $!e.Explored \wedge v.Explored$ **then**
 4. $p.Add(e), e.Explored \leftarrow true, unexplGE--, p.Add(v)$
 5. $u.NumUnexplEdges--, v.NumUnexplEdges--$
 6. **if** $unexplGE > 0$ **do**
 7. $p.Add(e), p.Add(u)$
 8. **while** there are unexplored edges $e = (u, v)$
 9. **choose** e **with** $min(v.NumUnexploredEdges)$
 10. $p.Add(e), e.Explored \leftarrow true, unexplGE--$
 11. $u.NumUnexplEdges--, v.NumUnexplEdges--$
 12. $TraverseGraph(G, v, p, unexplGE)$
 13. **if** $unexplGE = 0$ **then break**
 14. $p.Add(e), p.Add(u)$
-

Figure 3-2.: TRAVERSEGRAPH() procedure.**3.3.3. Correctness Proof**

In this section, we first show that the output walk $p = p_{1\dots\ell}$ produced by GLA contains all the nodes and edges in the input multigraph $G = (V, E)$ (condition (3) of Definition 2). Then, we show that $p = p_{1\dots\ell}$ is indeed a linearization of G , i.e., we show that conditions (1) and (2) of Definition 2 are also satisfied.

The analysis for condition (3) uses two properties:

- *Property 1.* If a node is visited, all of its adjacent edges must be visited as well. According to lines 2 – 4 and lines 8-10 of TRAVERSEGRAPH() in Figure 3-2, all the unexplored edges of a node will be visited when such node is visited.
- *Property 2.* If a node is visited, all its adjacent nodes must be visited as well. This property is guaranteed by the code in lines 8 – 12 of TRAVERSEGRAPH() in Figure 3-2.

With the two properties above, we can prove the following lemma:

Lemma 1. *Given a connected undirected multigraph $G = (V, E)$, the output walk p produced by GLA includes all its nodes and edges.*

Proof. We can prove the lemma by contradiction under two cases:

- *Case 1.* Suppose that all the nodes are explored but there exists at least one edge $e = (u, v)$ that is not visited by GLA. This hypothesis contradicts Property 1: when the node u is visited, all its adjacent edges must be visited as well.
- *Case 2.* Suppose that there exists at least one node u that is not visited. Assume that we start the linearization from node v_0 . Since the multigraph is connected, there is a walk from v_0 to u ; let us denote such walk as $\langle v_0, v_1, v_2, \dots, v_k, u \rangle$. We know that, for $0 \leq i < k$, v_i and v_{i+1} are adjacent nodes; v_k and u are adjacent too. Since v_0 is visited, according to Property 2, we know that v_1 is visited as well. Then, v_1, v_2, \dots, v_k and u are visited as well. Thus, we have that u is also visited, which contradicts the hypothesis.

Therefore, all the nodes and edges in G must have been visited by GLA and included in its output linearization p . □

Then, the correctness of is proven by the following theorem:

Theorem 2. *The Graph Linearization Algorithm (GLA) outputs a linearization $p_{1\dots\ell}$ of the input multigraph $G = (V, E)$.*

Proof. We must prove that conditions (1), (2) and (3) of Definition 2 are satisfied for the output walk $p = p_{1\dots\ell}$ generated by GLA. In particular, condition (3) is satisfied due to Lemma 1. Then, in the remainder of this proof we show that the walk $p_{1\dots\ell}$ satisfies conditions (1) and (2).

Notice that when `TRAVERSEGRAPH()` is called for the first time (line 8, Figure 3-1), a node u is inserted (line 1, Figure 3-2). Then, if the *if* statement of line 3 (Figure 3-2) is evaluated as *true*, edges e and nodes v are alternatively added for $e = (u, v)$ where e is unexplored and v is explored (line 4). After this, the *if* statement of line 6 is evaluated. If such evaluation yields *true*, the algorithm finishes; in such case, $p_{1\dots\ell}$ is an alternating sequence of nodes and edges where the odd indices correspond to nodes and the even indices correspond to edges. Furthermore each p_i , for even values in $1 \leq i \leq \ell$, is an edge that connects p_{i-1} and p_{i+2} ; this is because the graph elements inserted in p are adjacent (lines 2, 8, 9). Thus, the conditions (1) and (2) of Definition 2 are satisfied. Now, let us consider the case where the *if* statement of line 6 is evaluated as *false*. In such case, e and u are added into p again before visiting the next neighbours; thus conditions (1) and (2) are still satisfied for the current fragment of the walk p .

Then, in the loop of line 8, unexplored edges $e = (u, v)$ that lead to unexplored nodes v are considered. Edge e is inserted in line 10. When procedure `TRAVERSEDGRAPH()` is called in line 12, v is inserted as well. Then, if the *if* statement of line 13 is evaluated as *true*, conditions (1) and

(2) of Definition 2 are satisfied for the returned walk. If such statement is evaluated as *false*, e and v are added into p again before exploring the next neighbour (line 14); thus the conditions (1) and (2) of Definition 2 are still satisfied for the sequence of graph elements currently added into p . \square

3.3.4. Length of GLA Linearization

Theorem 3 shows that given the multigraph $G = (V, E)$, the length of the walk generated by GLA is at most 2 times the length of an optimal linearization. Therefore, the length produced by GLA is asymptotically optimal.

Theorem 3. *GLA is 2-approximate with respect to the length of the length-optimal linearization.*

Proof. Any linearization algorithm, including length-optimal algorithms, must traverse each edge of the multigraph at least once. Thus, for a multigraph $G = (V, E)$, where $n = |V|$ and $m = |E|$, the number of edges in its linearization is at least m . Since a linearization has the format of alternating between nodes and edges, a linearization with k edges has $k + 1$ nodes. Hence, the optimal linearization p^* has at least m edges and $m + 1$ nodes. Therefore, $|p^*| \geq 2m + 1$.

When GLA linearizes a multigraph, it visits any edge at most twice. This is because, when procedure `TRAVERSEGRAPH()` is executed over node u , an unexplored edge e that leads to any explored or unexplored node v is added once into p (lines 4 and 10, respectively, Figure 3-2). If after executing the next instructions there are still unvisited graph elements, it is necessary to go back to u through e ; this means that e and u are added into p again (lines 7 and 14, Figure 3-2). After this, e is not visited ever again given that only unexplored edges are considered (lines 3 and 8, Figure 3-2). Therefore, the number of edges in the linearization is at most $2m$. Again, since a linearization has the format of alternating between nodes and edges, the linearization p^{GLA} has at most $2m$ edges and $2m + 1$ nodes. Therefore, we have $|p^{GLA}| \leq 4m + 1$. It leads to the approximation ratio of GLA,

$$\frac{|p^{GLA}|}{|p^*|} \leq \frac{4m + 1}{2m + 1} \leq 2$$

\square

Notice that this theorem is based on the fact that each edge in the multigraph G appears at most twice in the linearization $p = p_{1\dots\ell}$ generated by GLA. Then, ℓ is compared to a lower bound that visits each edge only once to show worst-case approximation ratio. However, even an optimal linearization may not achieve the lower bound for many graph structures. Thus, for average cases in practice, GLA linearization is much closer to the optimal, as elaborated with empirical examples in Section 3.3.5.

3.3.5. Empirical Comparison on the Length of Different Linearization Algorithms

This section compares GLA with the linearization methods presented in Section 3.2: BFS-L and DFS-L. Furthermore, we consider a length-optimal algorithm, which we denote as OPT-L. Specifically, we compare the algorithms on the length of the linearizations produced for multigraphs of different topologies.

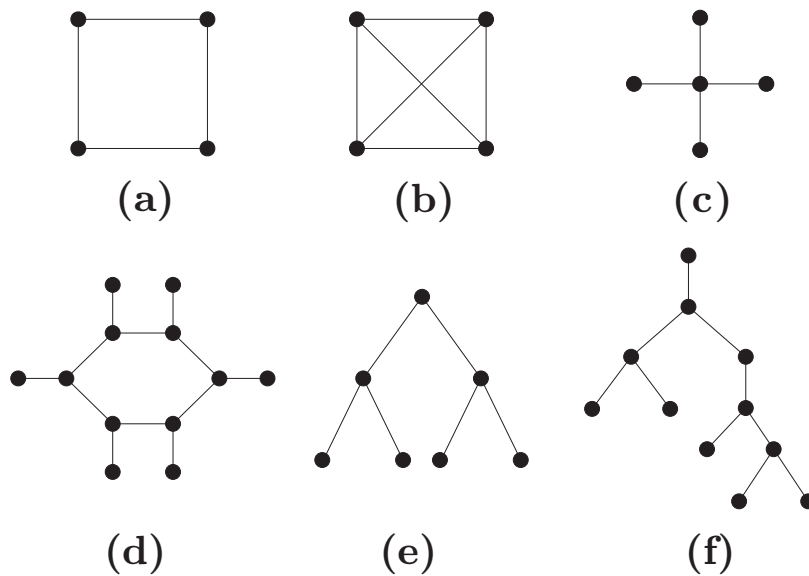


Figure 3-3.: Examples of graphs with different topology: (a) cyclic graph, (b) complete graph, (c) star graph, (d) neuron graph, (e) balanced tree and (f) unbalanced tree.

Table 3-1 shows the length of the linearizations produced by each algorithm for each graph in Figure 3-3. We can see that GLA performs better than both DFS-L and BFS-L for all cases. In some of them, the difference is quite significant. Moreover, DFS-L is much better than BFS-L because the length of the linearizations produced by BFS-L is considerably increased by the long walks that connect the node being examined and the next node to be examined; in DFS-L those walks are much shorter because of the traversal order. Even though GLA is similar to DFS-L, GLA's heuristics significantly reduce the length of the produced linearizations; they improve the locality of the walk so that GLA is likely to finish exploring one region of the graph before moving to another one, instead of shuffling among different regions that prolong linearization walks. In Table 3-1, we can see that these heuristics are effective: GLA performs close to OPT-L: GLA produces the optimal optimization for all the graphs in the table except for the unbalanced tree for which its result is close to the one of OPT-L.

Table 3-1.: Comparison of the output length of different linearization algorithms.

| Graph | Figure | Nodes | Edges | BFS-L | DFS-L | GLA | OPT-L |
|-----------------|---------------|-------|-------|-------|-------|-----|-------|
| Square graph | Figure 3-3(a) | 4 | 4 | 29 | 17 | 9 | 9 |
| Complete graph | Figure 3-3(b) | 4 | 6 | 35 | 25 | 17 | 17 |
| Star graph | Figure 3-3(c) | 5 | 4 | 31 | 24 | 13 | 13 |
| Neuron graph | Figure 3-3(d) | 12 | 12 | 124 | 49 | 35 | 35 |
| Balanced tree | Figure 3-3(e) | 7 | 6 | 53 | 25 | 17 | 17 |
| Unbalanced tree | Figure 3-3(f) | 11 | 10 | 79 | 41 | 35 | 31 |

3.3.6. Complexity Analysis

In this section, we derive the complexity of GLA for linearizing the graph $G = (V, E)$, where $n = |V|$ and $m = |E|$.

Time Complexity. In Figure 3-1, line 1 takes $O(m)$ time. The time complexity of lines 2 – 5 is $O(2m)$ because every edge in the graph is visited twice (as the graph is undirected). Line 6 takes $O(n)$. However, the complexity of GLA is dominated by the traversed walk (line 8, Figure 3-1) which corresponds to the linearization p . Notice that p has at most $2m$ edges and $2m + 1$ nodes (as presented in Section 3.3.4). Each insertion takes constant time as it is always done at the end of p . But when a node is inserted for the first time, it is necessary to consider the unexplored adjacent edges e that lead to unexplored nodes v sorted on $v.NumUnexplEdges$ (lines 8–9, Figure 3-2). This sorting operation takes $O(d \lg d)$, where d is the maximum degree of the nodes in G ; specifically $d = \max_{v \in V} v.degree$. Thus, the time complexity of GLA is $O(2m + n(d \lg d)) = O(m + dn \lg d)$.

Space Complexity. The space requirement of GLA is given by a list that stores the linearization $p = p_{1...l}$. Thus, the space complexity of GLA corresponds to the length of the linearization, i.e. $\Theta(l)$. Because a GLA linearization can have at most $2m$ edges and $2m + 1$ nodes, the space complexity of the algorithm is $\Theta(m)$.

4. Algorithm for Graph Isomorphism

In this chapter we present a linearization-based algorithm for solving graph isomorphism. In particular, the *Parameterized Matching on multi-Graphs* (PMG) algorithm uses a linearization of $G_1 = (V_1, E_1)$, denoted as $p = p_{1\dots\ell}$, and matches it against $G_2 = (V_2, E_2)$ to determine whether G_1 and G_2 are isomorphic by using Theorem 1. In Section 4.1 we present the high-level idea of the algorithm. Then, in Sections 4.2 and 4.3, we respectively present the pseudocode and prove its correctness. We present the complexity analysis of the algorithm in Section 4.4 while we show experimental results in Section 4.5. Finally, we discuss how to adapt this algorithm for subgraph isomorphism in Section 4.6.

4.1. Key Ideas

PMG considers all the possible injective functions $f : \mathcal{E}_{G_1} \rightarrow \mathcal{E}_{G_2}$ to determine whether there is mapping with two properties: (i) f is bijective; and (ii) there exists a walk $q = q_{1\dots\ell}$ in G_2 such that $q = q_{1\dots\ell}$ parameterized-matches $p_{1\dots\ell}$. These possible injective functions are explored by traversing p and G_2 simultaneously. Specifically, a graph element p_i is compared to a graph element ge in G_2 to determine whether an injective mapping is possible. We progressively extend a successful mapping by considering p_{i+1} and an adjacent graph element of ge . Thus, when the mapping is successful for p_ℓ , the traversed walk in G_2 parameterized-matches p ; hence G_1 and G_2 are isomorphic.

The graph elements of G_2 are traversed in the depth-first manner while p is traversed from left to right. Let us consider the DFS tree that represents the traversal of G_2 . Each level i of the DFS tree is comprised of graph elements from G_2 . If i is odd, it is a level of nodes; otherwise, it is a level of edges. Then, the idea of this traversal of G_2 is considering the possible injective mappings by attempting to set $f(p_i) = ge$ where $ge \in \mathcal{E}_{G_2}$ is a graph element at level i of the DFS tree. In order to guarantee that the mapping is injective, two conditions must be verified: (i) if $p_i = p_j$, for $i < j$, and $f(p_i) = ge$, then the only valid mapping for p_j is ge ; and (ii) if $p_i \neq p_j$, for $i < j$, and $f(p_i) = ge$, then $f(p_j) \neq ge$. Notice that if we consider all the assignments in the walk from the root of the DFS tree to a leaf at the level ℓ , we obtain a bijective function $f : \mathcal{E}_{G_1} \rightarrow \mathcal{E}_{G_2}$ such that $f(p_1) \cdots f(p_\ell)$ is a walk in G_2 that parameterized-matches $p_{1\dots\ell}$.

Next, we show our heuristics to prune the search space. At each step of the process, a node $u \in V_2$ and a node in p_i are compared. Let us say that we set $f(p_i) = u$. In order to extend the match, we use node degrees and previous assignments in f to prune the search space. Specifically, we

consider two cases:

Case 1: Node p_{i+2} is unassigned: We consider all the possible assignments $f(p_{i+1}) = e$ and $f(p_{i+2}) = v$ for edges $e = (u, v) \in E_2$ such that: (i) both e and v are unassigned; and (ii) $v.degree = p_{i+2}.degree$. Condition (i) is to guarantee that f is injective; condition (ii) is a pruning criterion based on that fact that, if G_1 and G_2 are isomorphic, then analogous nodes must have the same degree. Notice that if p_{i+2} is unassigned, p_{i+1} is unassigned as well; this is because the assignment of an edge in p is done at the same time (or after) the assignment of its end nodes. The process continues by considering p_{i+2} and each v .

Case 2: Node p_{i+2} is assigned to $v \in V_2$: There are two sub-cases. (a) Edge p_{i+1} is already assigned: it is not necessary to check adjacency as this was done when the mapping was set. We continue by considering p_{i+2} and v . (b) Edge p_{i+1} is unassigned: the algorithm considers all the possible assignments $f(p_{i+1}) = e$ for the unassigned edges $e = (u, v)$. The process continues at p_{i+2} and v .

Notice that the procedures for each of these cases guarantees that the mapping of both nodes and edges is injective. Furthermore, the DFS tree is expected to be sparse due to the pruning criteria; however, all the possible mapping functions are considered. If the algorithm reaches a successful assignment for p_ℓ , then the algorithm reports that the multigraphs are isomorphic.

4.2. Algorithm

The algorithm PMG, that determines whether $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic, is listed in Figure 4-1. The mapping function is represented as the array f ; namely, in such array, there is a position for each graph element ge in G_1 that stores its associated mapping $f(ge)$ to G_2 . On the other hand, boolean array g indicates if each graph element in G_2 is already assigned to a graph element in G_1 (through function f). The process starts by obtaining a linearization $p = p_{1\dots\ell}$ of G_1 by means of GLA. Then, we initialize the mappings of all the graph elements in G_1 as *undefined* (which we abbreviate as *undef* in the pseudocode). Likewise, we set that none of the graph elements in G_2 has been assigned to graph elements in G_1 .

After these initializations, we start the exploration of the search space. In particular, the DFS search trees are explored by calling the recursive procedure EXTENDMATCH() (see Figure 4-2). Each executing instance of this procedure considers a node p_i , a node $u \in V_2$ and a copy of arrays f and g . Furthermore, it is assumed that u has already been assigned to $f[p_i]$. Then, what the procedure does is attempting to set the adjacent graph elements of u , as mappings for p_{i+1} and p_{i+2} , under the two cases presented in Section 4.1. The corresponding partial mappings are extended by recursive calls to EXTENDMATCH() according to the rules of these cases.

The roots of the DFS search trees, which correspond to the initial calls to the recursive procedure (line 8, Figure 4-1), are the nodes in V_2 that have the same degree as p_1 . When we run PMG for the running example (see Figure 1-3), and $p = Ae_1Be_3Ce_4De_5Ce_5De_2Be_2De_6E$ is the linearization of G_1 , the match is returned when either walk $q_1 = Xe'_1Ye'_3Ze'_4We'_5Ze'_5We'_2Ye'_2We'_6S$ or walk $q_2 = Xe'_1Ye'_3Ze'_5We'_4Ze'_4We'_2Ye'_2We'_6S$ is traversed. Notice that both q_1 and q_2 parameterized-match p . The mapping functions of these matches correspond to the functions f_1 and f_2 presented in Figure 1-3(c).

Algorithm 3: PMG Algorithm

Input: $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$

Output: *true/false*

1. $p = GLA(G_1)$
 2. **for every** $ge \in (V_1 \cup E_1)$ **do** $f[ge] \leftarrow undef$
 3. **for every** $ge \in (V_2 \cup E_2)$ **do** $g[ge] \leftarrow false$
 4. **for every** $u \in V_2$ **do**
 5. **if** $u.degree = p_1.degree$
 6. $f' \leftarrow copyOf(f), f'[p_1] \leftarrow u$
 7. $g' \leftarrow copyOf(g), g'[u] \leftarrow true$
 8. **if** $ExtendMatch(u, p, 1, f', g', G_2) = true$
 9. **return true**
 10. **return false**
-

Figure 4-1.: PMG algorithm.

4.3. Correctness Proof

The correctness of the Parameterized Matching on multi-Graphs algorithm (PMG) is proven by the following theorem:

Theorem 4. *The Parameterized Matching on multi-Graphs algorithm (PMG) determines whether two input multigraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic.*

Proof. We prove the correctness of PMG by showing the following invariant: when $u \in V_2$ and p_i are processed in the recursive procedure $EXTENDMATCH()$, the walk $p_{1\dots i}$ parameterized-matches a walk $q_{1\dots i}$ in G_2 where $q_i = u$. Specifically, $q_{1\dots i} = f(p_1) \cdots f(p_i)$. Next, we show that this condition holds throughout the execution of the algorithm.

Algorithm 4: EXTENDMATCH() **Function****Input:** $u, p = p_{1..l}, i, f, g, G_2 = (V_2, E_2)$ **Output:** *true/false*

```

1.  if  $i = l$  then return true
2.  if  $f[p_{i+2}] = \text{undef}$ 
3.    for every  $e = (u, v) \in E_2$  do
4.      if  $g[v] = \text{false}$  and  $g[e] = \text{false}$  and  $v.\text{degree} = p_{i+2}.\text{degree}$ 
5.         $f' \leftarrow \text{copyOf}(f), f'[p_{i+1}] \leftarrow e, f'[p_{i+2}] \leftarrow v$ 
6.         $g' \leftarrow \text{copyOf}(g), g'[e] \leftarrow \text{true}, g'[v] \leftarrow \text{true}$ 
7.        if  $\text{ExtendMatch}(v, p, i + 2, f', g', G_2) = \text{true}$ 
8.          return true
9.    else
10.      $v = f[p_{i+2}]$ 
11.     if  $f[p_{i+1}] = \text{undef}$ 
12.       for every  $e = (u, v) \in E_2$  such that  $g[e] = \text{false}$ 
13.          $f' \leftarrow \text{copyOf}(f), f'[p_{i+1}] \leftarrow e$ 
14.          $g' \leftarrow \text{copyOf}(g), g'[e] \leftarrow \text{true}$ 
15.         if  $\text{ExtendMatch}(v, p, i + 2, f', g', G_2) = \text{true}$ 
16.           return true
17.     else
18.       if  $\text{ExtendMatch}(v, p, i + 2, f, g, G_2) = \text{true}$ 
19.         return true
20.   return false

```

Figure 4-2.: EXTENDMATCH() function.

- **Initialization:** When $i = 1$, p_1 parameterized-matches u given that $f(p_1) = u$ was set in line 6 - Figure 4-1 before calling $\text{ExtendMatch}(u, p_1)$ (line 8, Figure 4-1).
- **Maintenance:** Let us assume that the invariant holds for p_i and u . Then, we have to consider two cases: (i) $f(p_{i+2})$ has not been defined; and (ii) $f(p_{i+2}) = v'$. Let us consider first case (i). Notice that, because of lines 3 – 4 in Figure 4-2, $\text{ExtendMatch}(v, p_{i+2})$ is only called for unassigned nodes v such that $e = (u, v) \in E_2$ is also unassigned; thus, adding $f(p_{i+1}) = e$ and $f(p_{i+2}) = v$ (line 5, Figure 4-2) maintains the injective property of f . Hence, when $\text{ExtendMatch}(v, p_{i+2})$ is executed, $p_{1..i+2}$ parameterized-matches the walk $f(p_1) \cdots f(p_{i+2})$ in G_2 and $f(p_{i+2}) = v$. For the case (ii), we have two sub-cases: that p_{i+1} is already assigned or that it is not. In the former, we need not change the function and, thus, the invariant will hold when $\text{ExtendMatch}(v, p_{i+2})$ is called (line 18, Figure 4-2). In the latter, the algorithm only considers unassigned edges $e = (u, v')$ for the mapping of $f(p_{i+1})$ (lines 12 – 13, Figure 4-2); hence, when $\text{ExtendMatch}(v', p_{i+2})$ is called in line 15, the mapping f will still be injective and

$$f(p_{i+2}) = v'.$$

- **Termination:** When no matches can be extended until p_ℓ , all the calls of *ExtendMatch()* return *false* (see line 20); hence, PMG returns *false* as well (line 10, Figure 4-1). This is correct as all the possible mapping functions were considered. Namely, (i) the search was started from all the valid nodes (lines 5 – 10, Figure 4-1); and (ii) all the possible mappings were considered at all stages of the search (lines 3 – 5 and 12, Figure 4-2). On the contrary, if a match is extended in any branch of the search and reaches position ℓ (line 1, Figure 4-2), we obtain that $p_{1\dots\ell}$ parameterized-matches the walk $q_{1\dots\ell} = f(p_1) \cdots f(p_\ell)$ in G_2 ; then, PMG returns *true* (line 9, Figure 4-1). In such case, considering Theorem 1, we conclude that G_1 and G_2 are isomorphic.

□

4.4. Complexity Analysis

In this section, we analyse the complexity of determining isomorphism of multigraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, where $n = |V_1| = |V_2|$ and $m = |E_1| = |E_2|$, using PMG.

Time Complexity. The PMG algorithm performs two basic steps: (i) linearizing multigraph G_1 into a walk $p = p_{1\dots\ell}$; and (ii) matching the linearization $p = p_{1\dots\ell}$ against multigraph G_2 . The former takes $O(m + dn \lg d)$, where d is the maximum degree of the nodes in G_1 , as presented in Section 3.3.6. The complexity of the latter is studied in this section.

The initialization of the arrays f and g takes $O(n+m)$ (lines 2–3, Figure 4-1). However, this cost is negligible with respect to the number of executions of the recursive procedure *EXTENDMATCH()*; each execution requires constant time. This number is equal to the number of nodes and edges in the DFS search trees. The number of edges in a DFS tree is equivalent to the number of nodes: each node, except the root, is associated to an edge that leads to its parent. Then, the asymptotic behavior of PMG depends on the number of nodes in the DFS trees. Next theorem gives an upper bound for this number.

Theorem 5. *Let $p = p_{1\dots\ell}$ be a linearization of multigraph G_1 . Also, let d be the maximum degree of the nodes in multigraph G_2 ; specifically $d = \max_{v \in V_1} v.\text{degree}$. The DFS tree that represents the traversal of G_2 done by PMG has at most $O(d^{\lceil \ell/2 \rceil})$ nodes.*

Proof. Let n and m be the cardinality of the sets of nodes and edges, respectively, of both G_1 and G_2 (i.e., $n = |V_1| = |V_2|$, $m = |E_1| = |E_2|$). In the worst case, all the nodes in V_1 and V_2 have the same degree; in such case, the pruning criterion based on only matching the nodes $v \in V_2$ with the same degree of p_i is not useful. The DFS tree rooted at one of the nodes of multigraph G_2 has $\lceil \ell/2 \rceil$ levels of nodes and $\lfloor \ell/2 \rfloor$ levels of edges; we just consider the levels of nodes. The root has

one node and the second level has d nodes. Each of these d nodes is associated to $d - 1$ nodes in the third level (as the edges that lead to nodes in upper levels of the tree are not considered); thus, the third level has $d(d - 1)$ nodes. Similarly, the fourth level has $d(d - 1)(d - 2)$ nodes. In general, level i of the tree has $\prod_{j=0}^{i-2} (d - j)$ nodes. Thus, the total number of nodes of a DFS tree is:

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

Since a linearized walk alternates between nodes and edges while starting and ending at a node, ℓ is odd. Thus, $O(d^{\lceil \ell/2 \rceil - 1}) = O(d^{\lfloor \ell/2 \rfloor})$. □

As we have a DFS tree starting at each node in G_2 , the total number of nodes visited, and hence the time complexity of PMG, is $O(nd^{\lfloor \ell/2 \rfloor})$. Thus, it is important to have a short linearization of G_1 . Note that if G_2 is complete, i.e., $d = n - 1$, the time complexity is $O(n(n - 1)^{\lfloor \ell/2 \rfloor}) = O(n^{\lfloor \ell/2 \rfloor})$.

However, it is important to remark that Theorem 5 gives an upper bound for the worst-case complexity. It assumes that, at every level of nodes, all the possible neighbors are explored. The average-case situations are often not that “bad” because: (i) when a node p_i has already been assigned, only such assigned node is considered; and (ii) when the multigraph has varied node degrees, the pruning criterion highly reduces the number of adjacent nodes to be visited. Thus, in practice, our algorithm has a better performance than the given worst-case bound.

Space Complexity. PMG compares the linearization p of G_1 with its potential parameterized-matching walks in G_2 . Notice that only one of such walks is considered at a time. Therefore, PMG only needs to store the mapping table of the linearization with respect to the current walk being considered. Such table contains mapping for all the nodes and edges. Hence, the space complexity of PMG is $\Theta(n + m)$.

4.5. Experimental Evaluation

We assess the performance of our proposed approach experimentally. We implement our proposed framework in both Python and C# and release them as public resources ¹ to help future comparison studies. We compare our approach to VF2, using an optimized implementation from the networkX library ². Both are implemented in Python and open-sourced.

¹<http://ids.postech.ac.kr/graph>

²<http://networkx.github.io>

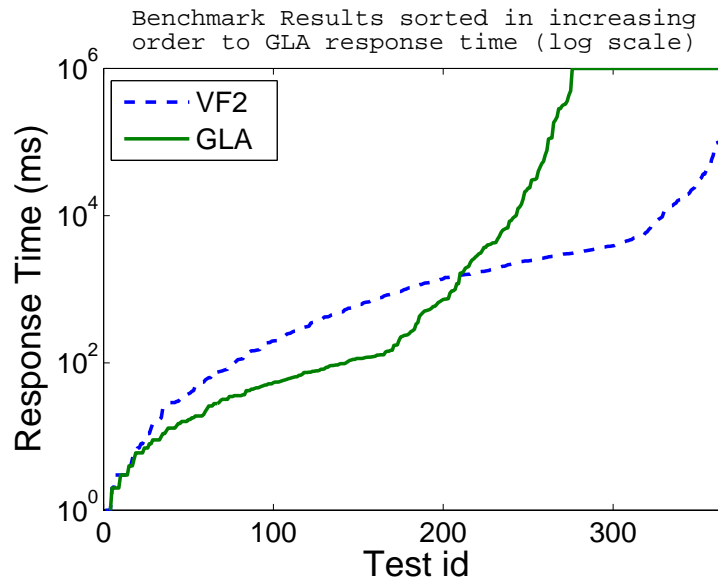


Figure 4-3.: Response time of GLA and VF2 on the benchmark graphs. *Test id* is sorted, in ascending order, on GLA response time.

As datasets, we employ a set of public benchmark graphs (Section 4.5.1) and synthetic graphs (Section 4.5.2). All evaluations are performed on a server running under a Windows platform on a 3,40GHz CPU with 16GB memory.

4.5.1. Benchmark Graphs

In this section, we study performance over a collection of public benchmark graphs used for evaluating isomorphism papers, to validate the generality of GLA. More specifically, we consider the following three widely adopted families of graphs in the public repository³. Each family has graphs of size up to one thousand nodes.

- Strongly regular graphs: Graphs with high regularity. The repository has 87 pairs, covering families of Steiner Triple, Latin Square, Paley, Lattice, and Triangular graphs used in prior literature.
- Component-based graphs: Graphs connecting regular graphs as a component. The repository has 84 pairs, covering a union of regular graphs, cliques, or tripartite graphs used in existing work.

³<https://sites.google.com/site/giconauto/home/benchmarks>

- **Graphs based on Miyazaki’s construction:** Graphs following Miyazaki’s construction to deliberately add complexity to the problem. There are 195 instances available in the repository.

Figure 4-3 first illustrates the response time differences of VF2 and GLA for all 366 instances. *Test id* is assigned, in ascending order, on GLA response time (X-axis); the Y-axis represents the response time of the two algorithms. Observe that in 181 pairs, about half of the datasets, GLA terminates earlier. This is encouraging as the benchmark datasets are intentionally biased into “strongly regular” graphs. For example, nodes in the graphs in the strongly regular graph group are all of the same degree, and node degrees in the component-based graphs also coincide into few values (*e.g.*, degree of a node within the regular component and degree of a connecting node). Our target scenario is supporting graphs that are more structurally heterogeneous. The third group of Miyazaki’s construction adds some such variation, from which GLA clearly excels VF2. In summary, GLA shows fast response in the half of all benchmark graphs.

Table 4-1 and 4-2 show the break down of this result. In Table 4-1, our empirical finding is consistent with the above analysis, as VF2 excels in regular graphs, while GLA is significantly faster in 65 % (126 out of 195) in Miyazaki-based constructed graphs. This is an interesting result since Miyazaki-constructed graphs constitute one of the hardest cases for graph isomorphism algorithms [141]. Table 4-2 shows cases where GLA is not short-running, or takes 2+ minutes. However, in the majority of 272 short-running cases, namely 66 % of such cases, GLA runs faster. This opens up a possibility of a hybrid algorithm that selects between these two algorithms, either statistically-based on the graph topology or dynamically after running for some time, which we leave as future work.

Table 4-1.: Number of winning cases of GLA and VF2 on the benchmark graphs.

| | Strongly Regular graphs | Component- Based Graphs | Miyazaki’s construction | Total |
|------------|------------------------------------|------------------------------------|------------------------------------|--------------|
| VF2 | 63 | 53 | 69 | 185 |
| GLA | 24 | 31 | 126 | 181 |

4.5.2. Synthetic Graphs

In this section, we study performance over synthetically generated graphs to isolate the factors that positively and negatively affect the performance of the two algorithms. For graph generation, we deliberately avoid the “trivial cases”. For example, consider a graph where node v_i is connected to v_1, \dots, v_{i-1} , for $1 < i \leq n$. As the degree of each node is unique, testing isomorphism can be done trivially by using a simple heuristic like sorting nodes by degree. In contrast, we consider cases

Table 4-2.: Ratio of short-running cases of GLA and VF2 on the benchmark graphs.

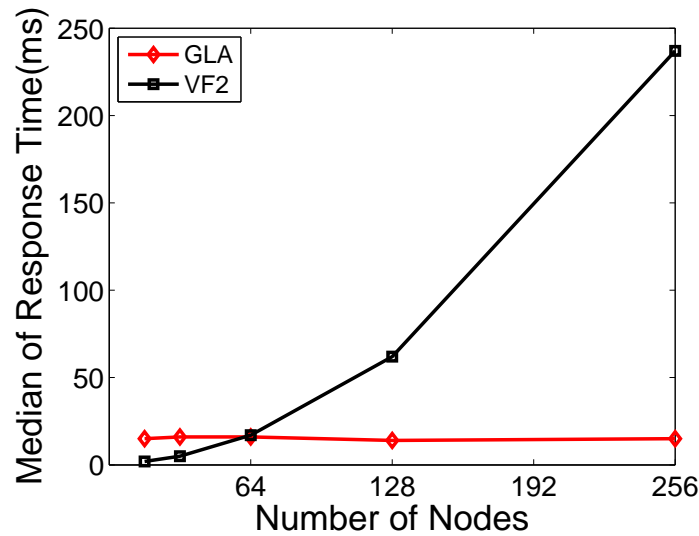
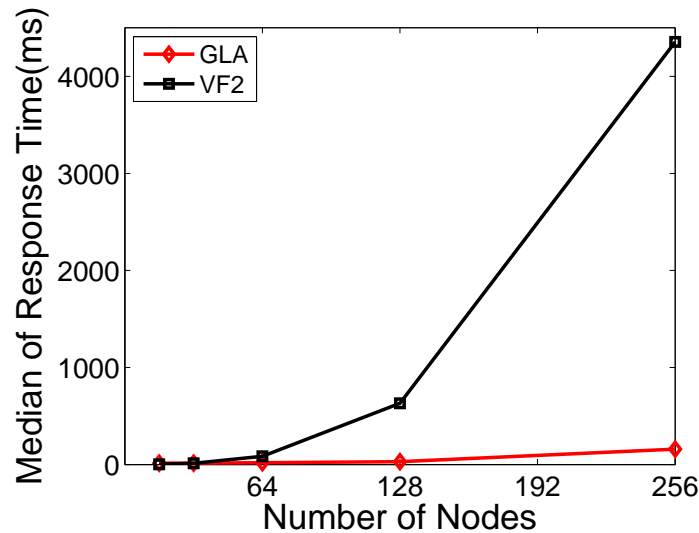
| | Strongly Regular graphs | Component- Based Graphs | Miyazaki's construction | Total |
|------------|------------------------------------|------------------------------------|------------------------------------|---------------------|
| VF2 | 100 % (87/87) | 100 % (84/84) | 100 % (195/195) | 100 % (366/366) |
| GLA | 55.2 % (48/87) | 94.0 % (79/84) | 74.4 % (145/195) | 74.3 % (272/366) |

where no such simple heuristic can be used. Graphs where every node has identical degree would be much more challenging in that sense.

Meanwhile, we also randomize the node degrees to complement benchmark studies focusing on regular topologies. Given that the complexity of VF2 is reported to vary significantly over degree, from $O(n^2)$ to $O(nn!)$ [44], we consider both low- and high-degree cases to evaluate algorithms in a wide spectrum of settings. The lower end of this spectrum is observed when the matching graphs are early found in a sparse graph, while the opposite case of dense graphs often leads to long running times. More specifically, we generate sparse and dense identical-degree graphs as follows: **1-Sparse:** We generate a random graph G , with n nodes and $3n$ edges, where every node has degree three. We first build a random binary tree with $n - 1$ edges. Then, the nodes with the degree less than three are connected to another such node randomly chosen. **2-Dense:** We generate graph G' by subtracting G from a complete graph. Every node of G has the same degree (i.e., $n - 4$).

In each setting, we vary the number of nodes from 16 to 256 to evaluate the response time of GLA and VF2. For each point in the figures, we randomly generate 45 graphs and report the median response time. We choose median response time as our performance metric because the running time on different graphs significantly varies over graph complexity (as discussed above) while the optimization margin is narrow for easy cases and hard extremes. Our target problems are thus neither of these. Namely, using the average or min/max as the main performance metric would bias the results to represent either extreme. In contrast, median would filter out extreme results.

Figure 4-4(a) and (b) show the results for sparse graphs and dense graphs, respectively. The X -axis is the number of nodes (in log scale) and the Y -axis is the median response time in milliseconds. Note the two figures have different scales. Furthermore, the number of edges is linear with the number of nodes for sparse graphs and quadratic for dense graphs. In Figure 4-4(a), the median running time of GLA remains more or less constant to 10 milliseconds, despite the increase in graph size. As a result, when $n = 256$, GLA outperforms VF2 by an order of magnitude. In Figure 4-4(b), we observe a consistent trend, except that the performance gap is larger. In particular, for $n = 256$, GLA is faster by two orders of magnitude. These figures show that GLA has low res-

(a) *Sparse* median response time.(b) *Dense* median response time.**Figure 4-4.:** Response time of GLA and VF2: (a) on sparse graphs; and (b) on dense graphs.

ponse time, less than VF2. This can be explained by the effective pruning of the notoriously large search space, which is guided by the heuristics employed during the linearization and matching phases.

4.6. PMG-SI: Solution for Subgraph Isomorphism

In this section we present an adaptation of PMG to solve the subgraph isomorphism problem. Specifically, we determine whether graph $G_1 = (V_1, E_1)$, where $n_1 = |V_1|$ and $m_1 = |E_1|$, is

isomorphic to a subgraph in $G_2 = (V_2, E_2)$, where $n = |V_2|$ and $m = |E_2|$. We solve this problem by means of the linearization approach, which is correct due to the Corollary 1 of Theorem 1. In Section 4.6.1 we present the algorithm and in Section 4.6.2 we present some experimental results.

4.6.1. Algorithm

The algorithm is essentially the same as the one presented in Figure 4-1, considering that we linearize the smallest graph (i.e., G_1). However, some aspects must be taken into account. For instance, the length of array f is n_1 while the length of array g is n . Furthermore, the heuristic to prune on node degrees must be modified: instead of an equality constraints, we must use inequalities. In particular, the condition of line 5 in Figure 4-1 is replaced by $u.degree \geq p_1.degree$; similarly, the corresponding constraint of line 4 in Figure 4-2 is replaced by $v.degree \geq p_{i+2}.degree$. This might decrease the pruning power of the node degree heuristic, especially when the node degrees in G_2 are significantly higher than the ones of G_1 .

Algorithm 5: PMG-SI Algorithm

Input: $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$

Output: \mathcal{R}

1. $p = GLA(G_1), \mathcal{R} = \emptyset$
 2. **for every** $ge \in (V_1 \cup E_1)$ **do** $f[ge] \leftarrow undef$
 3. **for every** $ge \in (V_2 \cup E_2)$ **do** $g[ge] \leftarrow false$
 4. **for every** $u \in V_2$ **do**
 5. **if** $u.degree \geq p_1.degree$
 6. $f' \leftarrow copyOf(f), f'[p_1] \leftarrow u$
 7. $g' \leftarrow copyOf(g), g'[u] \leftarrow true$
 8. $ExtendMatchSI(u, p, 1, f', g', G_2, \mathcal{R})$
 9. **return** \mathcal{R}
-

Figure 4-5.: PMG-SI algorithm.

Figure 4-5 presents a variation of the algorithm, called PMG-SI, that reports all the subgraphs of G_2 that are isomorphic to G_1 in the set \mathcal{R} . The DFS traversal is performed by the recursive procedure `EXTENDMATCHSI()` (see Figure 4-6). The correctness proof and the complexity analysis presented in Sections 4.3 and 4.4, respectively, also apply for PMG-SI. Thus, its time complexity is $O(nd^{\lfloor \ell/2 \rfloor})$, where d is the maximum node degree in G_2 and ℓ is the length of G_1 's linearization.

4.6.2. Experimental Evaluation

We evaluate the performance of our approach for solving subgraph isomorphism under a variety of graph sizes. We first show the experimental setup and then we show the results.

Algorithm 6: EXTENDMATCHSI() Procedure**Input:** $u, p = p_{1..l}, i, f, g, G_2 = (V_2, E_2), \mathcal{R}$

-
1. **if** $i = l$ **then return** $\mathcal{R}.Add(f)$
 2. **if** $f[p_{i+2}] = undef$
 3. **for every** $e = (u, v) \in E_2$ **do**
 4. **if** $g[v] = false$ **and** $g[e] = false$ **and** $v.degree \geq p_{i+2}.degree$
 5. $f' \leftarrow copyOf(f), f'[p_{i+1}] \leftarrow e, f'[p_{i+2}] \leftarrow v$
 6. $g' \leftarrow copyOf(g), g'[e] \leftarrow true, g'[v] \leftarrow true$
 7. $ExtendMatchSI(v, p, i + 2, f', g', G_2, \mathcal{R})$
 8. **else**
 9. $v = f[p_{i+2}]$
 10. **if** $f[p_{i+1}] = undef$
 11. **for every** $e = (u, v) \in E_2$ **such that** $g[e] = false$
 12. $f' \leftarrow copyOf(f), f'[p_{i+1}] \leftarrow e$
 13. $g' \leftarrow copyOf(g), g'[e] \leftarrow true$
 14. $ExtendMatchSI(v, p, i + 2, f', g', G_2, \mathcal{R})$
 15. **else**
 16. $ExtendMatchSI(v, p, i + 2, f, g, G_2, \mathcal{R})$
-

Figure 4-6.: EXTENDMATCHSI() procedure.**Experimental Setup****Implementation.** We implement the GLA algorithm and the PMG algorithm in C#.**Small graphs.** We employ graphs $G_1 = (V_1, E_1)$ with different sizes and topologies. In order to vary the topology, we consider complete graphs, path graphs, cyclic graphs and star graphs. A cyclic graph is a path in which the first node is the same as the last node. A star graph is a graph in which one node is connected to every other node.**Large graphs.** We generate graphs $G_2 = (V_2, E_2)$ using the *Recursive Matrix* (RMAT) model [31] that generates scale-free graphs similar to the types of graphs used in many applications. We use graphs with different sizes: $|V_2| = 1024, 16384, 131072, 524288, 1048576$. The number of edges of each graph is $|E_2| = 5 \times |V_2|$ representing sparse graphs.**Hardware.** We perform the experiments on a commodity server with 3,30GHz Intel Xeon X5680 CPU with 24GB 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 server, 400000 comparisons are performed per second. We also report the length of the linearization generated for G_1 .

Experimental Results

The small graphs $G_1 = (V_1, E_1)$ were linearized into walks $p = p_{1\dots\ell}$ by means of the GLA algorithm. Then, they were queried on each one of the large graphs $G_2 = (V_2, E_2)$, where $n = |V_2|$ and $m = |E_2|$, via the PMG algorithm. The behaviour of the matching process was very similar on all the large graphs; thus, we analyse in detail only the experiment on the large graph with $|V_2| = 1024$.

- **For complete graphs:** The length of the linearization grows linearly with number of edges in the graph G_1 as expected from Theorem 3 (see Figure 4-7(a)). However, it grows at a quadratic rate with V_1 (see Figure 4-7(b)); this can be explained by the fact that in complete graphs $|E_1| = O(|V_1|^2)$. The time taken by the algorithm grows faster for low values of ℓ than for greater values (see Figure 4-7(c)). This is because for low values there are more matches and, hence, more graph elements need to be explored; for greater values, mismatches are early detected.
- **For path graphs:** The length of the linearization is linear respect to both the number of nodes and edges in G_1 (see Figure 4-8(a,b)). This is because in paths $|E_P| = O(|V_P|)$. The time taken by PMG grows exponentially on the length of the linearization of G_1 (see Figure 4-8(c)). This verifies the time complexity analysis of PMG in Section 4.4, which concludes that its time complexity is $O(nd^{\lfloor \ell/2 \rfloor})$ where d is the maximum degree of the nodes in G_1 . Given that, as the experiment is done on a fixed G_2 , both n and d are constant. Consequently, we obtain an exponential function of ℓ . Given that for cyclic and star graphs, $|E_P| = O(|V_P|)$, the performance for such graphs is similar.

In Figure 4-9, we show a comparison of the performance among all the different small graphs G_1 against two large graphs G_2 . In Figure 4-9(a) we show the linearization length of the different types of graphs G_1 . Because of the number of edges, the linearization of the complete graphs is significantly much longer in all cases. Figure 4-9(b,c) show the time taken by PMG for all the types of graphs G_1 in the graphs G_2 of $|V| = 1024$ and $1,048,576$, respectively.

Notice that the time taken by the cyclic graphs is considerably greater than the other types of graphs G_1 on both large graphs G_2 . This is because, even though paths, star and cyclic graphs of a given $|V_1|$ have a similar topology, their number of edges are $|V_1| - 1$, $|V_1| - 1$ and $|V_1|$, respectively. So let us compare the matching of a cyclic graph with both the path and the star graphs. A cyclic graph is the same as a path but with an additional edge that connects the first node with the last one.

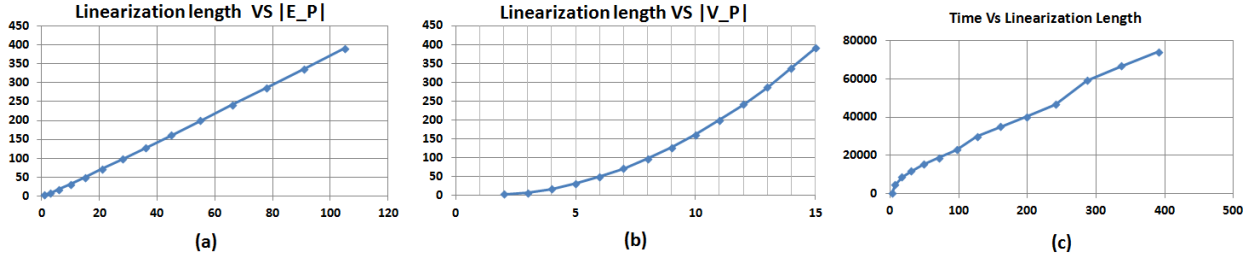


Figure 4-7.: Subgraph isomorphism for complete graphs $G_1 = (V_1, E_1)$ on a graph $G_2 = (V_2, E_2)$ where $|V_2| = 1024$. (a) Linearization obtained by GLA for different values of E_1 . (b) Linearization obtained by GLA for different values of $|V_1|$. (c) Time taken by PMG to solve the subgraph isomorphism problem for different values of the linearization length.

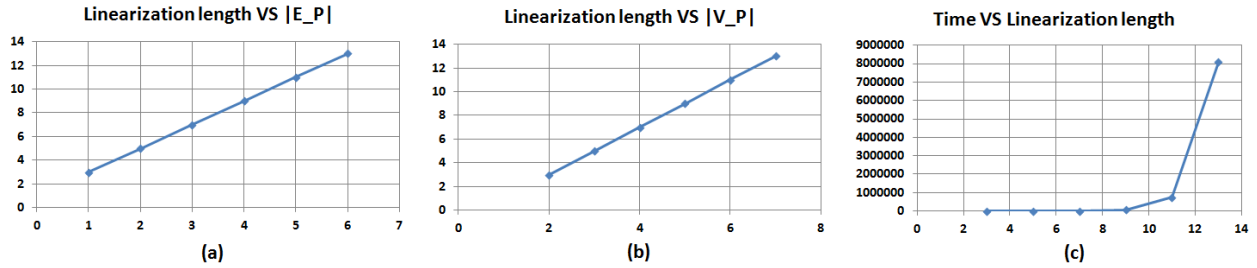


Figure 4-8.: Subgraph isomorphism for path graphs $G_1 = (V_1, E_1)$ on a graph $G_2 = (V_2, E_2)$ where $|V_2| = 1024$. (a) Linearization obtained by GLA for different values of E_1 . (b) Linearization obtained by GLA for different values of $|V_1|$. (c) Time taken by PMG to solve the subgraph isomorphism problem for different values of the linearization length.

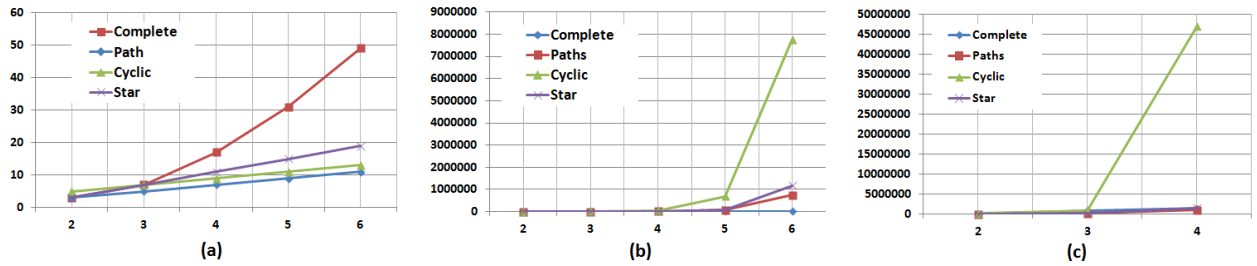


Figure 4-9.: Experimental results of different types of graphs $G_1 = (V_1, E_1)$ (complete, path, cyclic and star graphs) on different graphs $G_2 = (V_2, E_2)$. X-axis is $|V_1|$. (a) Linearization length obtained by GLA for all the graphs $G_1 = (V_1, E_1)$. Number of graph element comparisons used in PMG for graphs $G_2 = (V_2, E_2)$ with (b) $|V_2| = 1024$, (c) $|V_2| = 1048576$.

Thus, the linearization of a cyclic graph of a given V_1 exceeds in one the length of the linearization of a path with the same number of nodes. Consequently, the time required to solve the problem for a cyclic graph with $|V_1|$ nodes is equivalent to the time required for a path with $|V_1| + 1$ nodes (as

it can be verified in Figure 4-9(b,c)).

Even though the number of edges of a graph with $|V_1|$ nodes is $|V_1| - 1$ for a star graph and $|V_1|$ for a cyclic graph, the linearization of the star graph can be longer. This is because most edges of the star graphs are visited back and forth in order to return to the central node. However, because the mapping of a given edge $e \in E_1$ is established only once, the second occurrence of the edge in the linearization is skipped. Therefore, the time taken for a star graph of a given $|V_1|$ is similar to the time of a path of the same size but much less than the time of a cyclic graph of the same size. On the other hand, even though complete pattern graphs have long linearizations, the time required for the matching phase is low due to the early detection of mismatches.

Part II.

Queries on Attributed Graphs Solved through Parameterized Matching

5. Generalized Pattern Queries

In the last decades, the use of multigraphs to represent information of different types has widely spread. In particular, attributed multigraphs have been used to represent social networks [114], communication networks [20] and bioinformatics structures [92], to name some. In this chapter, we define the attributed multigraph model considered in this thesis (Section 5.1). Furthermore, we introduce a new type of queries on attributed multigraphs: *generalized pattern queries*. We show that our queries support both reachability and pattern match queries and even queries that cannot be represented under these models (Section 5.2).

5.1. Graph Model

We consider an *attributed multigraph*, a graph in which nodes and edges have attributes; furthermore, more than one edge may connect the same pair of nodes representing different relationships. Next, a more formal definition is presented:

Definition 5 (Attributed Multigraph). *An attributed multigraph $G = (V, E, f_V, f_E)$ is a multigraph where:*

- V is a set of nodes.
- E is a multiset of edges on $V \times V$.
- f_V is a node-attribute function defined on the set of nodes V such that, for each $v \in V$, $f_V(v) = (A_1 = a_1, \dots, A_d = a_d)$ where A_k is an attribute and a_k is the value assigned to the corresponding attribute.
- f_E is an edge-attribute function defined on E in a similar way as f_V is defined on the set of nodes V .

Furthermore, we assume that each node v has at least two attributes: $v.id$ which is a unique identifier, and $v.type$ which is a label (or categorical attribute) called *type*. Figure 5-1 shows an example of an attributed multigraph modelling a social network. It contains nodes of two types: *person* and *photo*, and edges of three types: *friend*, *colleague* and *tag*. A person node may have the *gender* attribute; its value is *male* for the nodes with *ids* “Mike” and “Nick” and *female* for the nodes with *ids* “Alice”, “Mary”, “Rose” and “Kate”.

Unlike previous literature that does not consider attributes for edges, we support edges with attributes. This can be useful, for instance, to consider the date when the friendship or the tag was established. In Figure 5-1, we can see that Rose and Nick became friends on December 2, 2013.

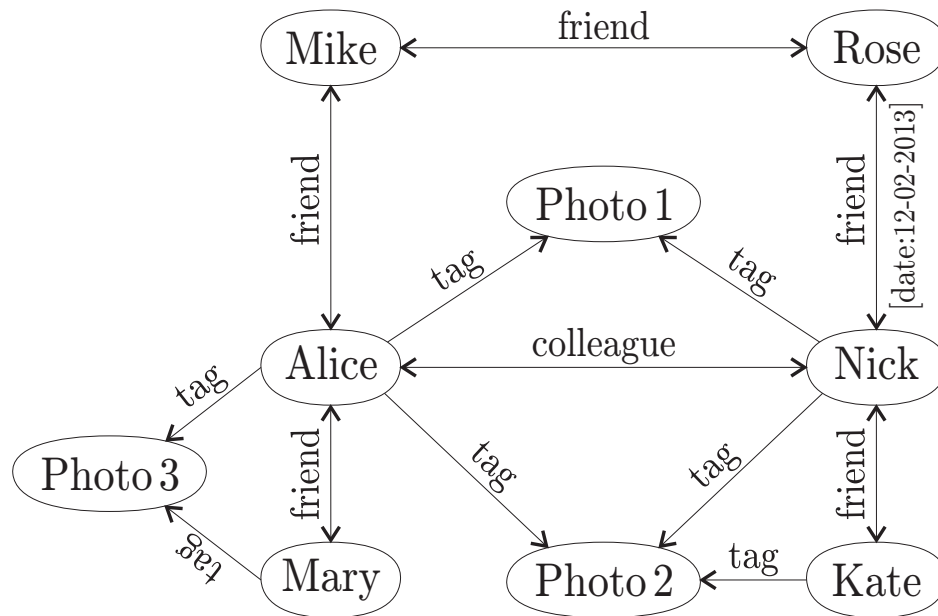


Figure 5-1.: Example of an attributed multigraph that represents a social network. Node type is *person* or *photo*, and edge type is *friend*, *colleague* or *tag*.

As graph processing generally requires random data access (no locality), we assume that the graph is managed in main memory to allow fast random access using pointer-base representation. Furthermore, if the graph is directed, each node has references to both inbound and outbound edges to allow queries to traverse both directions.

5.2. Query Model

We propose a novel query model, which allows to find paths, walks, and subgraphs that satisfy reachability requirements. We call our queries *generalized pattern queries*. In this section, we first introduce a few new concepts, and we use them to define our query model. We also present examples and discuss the expressive power of our queries. We show that our queries can be used to express both reachability queries and pattern match queries, and beyond.

5.2.1. Prerequisites

Our model supports not only structural requirements, but also attribute predicates and reachability. In particular, our queries contain nodes and edges that are associated to *attribute predicates* and

edges that are associated to *reachability expressions*. In this section we formalize these concepts.

An *attribute predicate* is a set of conditions on the attributes established for a node or an edge in a graph query. Specifically, an attribute predicate is either called a *node predicate* or an *edge predicate* depending on whether it is *associated to* a node or an edge, respectively. We formalize this definition as follows:

Definition 6 (Attribute Predicate). *An attribute predicate on a node (or edge) of an attributed multigraph is an expression, with conjunctions and disjunctions, that evaluates some conditions on its attributes. Let us denote as $u.pred$ the predicate associated to the node (or edge) u . An attribute predicate $u.pred$ associated to the node (or edge) u is drawn from the following grammar:*

$$P \rightarrow \text{Attribute Op Constant} \mid P \wedge P \mid P \vee P$$

$$\text{Op} \rightarrow < \mid \leq \mid > \mid \geq \mid = \mid \neq$$

A node/edge v in the attributed multigraph satisfies the predicate $u.pred$, denoted as $v \sim u$, if $u.pred$ is true when evaluated with v 's attributes.

Furthermore, our model generalizes the traditional definition of reachability in graphs — we introduce the concept of (u, v, ρ) -reachability. Here u and v represent nodes associated to node predicates and ρ represents an edge that is associated to a *reachability expression*, which we define next.

Definition 7 (Reachability Expression). *Let e and n represent edges and nodes that are associated to predicates, respectively. We define a reachability expression as an expression defined over Q :*

$$Q \rightarrow e \mid Q(nQ)^* \mid (Qn)^*Q \mid Q \cup Q \quad (5-1)$$

A reachability expression is associated to an edge in our query. We denote it as $\rho.re$ when it is associated to edge ρ .

Note that a reachability expression represents a reachability relation. Hence, the corresponding edge can be mapped to a path with arbitrary length of the attributed multigraph.

Definition 8 ((u, v, ρ) -Reachability). *Let us consider the nodes u and v associated to predicates, and an edge ρ associated to a reachability expression. Given an attributed multigraph $G = (V, E, f_V, f_E)$ and two nodes $u', v' \in V$, v' is said to be (u, v, ρ) -reachable from u' iff: (1) $u' \sim u$; (2) $v' \sim v$; and (3) there exists a path drawn from $\rho.re$ that connects u' and v' .*

Note that a reachability expression $\rho.re$ must be an alternating sequence of edges and nodes associated to predicates where the Kleene star and union operators are supported. We use $\mathcal{W}_G(u, v, \rho)$

to represent the set of all paths in $G = (V, E, f_V, f_E)$ that connect any $u' \in V$ to any $v' \in V$ such that v' is (u, v, ρ) -reachable from u' .

In order to determine whether a particular $v' \in V$ is (u, v, ρ) -reachable from a given $u' \in V$, we can either match *forward* from u to v through $\rho.re$ (*forward approach*), or match from v to u *backward* through $\rho.re^R$, which is the reversal of the reachability expression $\rho.re$ (*backward approach*). In the forward approach, outbound edges of the nodes in the attributed multigraph are considered; in the backward approach, inbound edges of the nodes are traversed. Table 5-1 shows how to obtain the reversal of different reachability expressions drawn from the context-free grammar Q under different rules of production (see Definition 7).

Table 5-1.: Reversal of reachability expressions drawn from Q under different rules of production (see Definition 7).

| | |
|------------|----------------|
| $\rho.re$ | $\rho.re^R$ |
| e | e |
| $Q(nQ)^*$ | $(Q^R n)Q^R$ |
| $(Qn)^*Q$ | $Q^R(nQ^R)^*$ |
| $Q \cup Q$ | $Q^R \cup Q^R$ |

5.2.2. Definition of Generalized Pattern Queries

Now, we are ready to define generalized pattern queries and their matches:

Definition 9 (Generalized Pattern Query). *A generalized pattern query $G_P = (V_P, E_P)$ is a weakly connected directed multigraph¹ where:*

- V_P is a set of nodes associated to node predicates.
- E_P is a multiset of edges, on $V_P \times V_P$, that are associated to reachability expressions.
- Each edge $e' = (u, v) \in E_P$ establishes a (u, v, e') -reachability requirement.

A match of a generalized pattern query $G_P = (V_P, E_P)$ is a set of nodes in the attributed multigraph that bijectively satisfy the predicates associated to the nodes in V_P . Furthermore, each pair of nodes in the match must satisfy the reachability requirements established for the edges that connect their corresponding nodes in G_P . Hence, our model considers structural requirements, predicates and reachability. We present a more formal definition next.

¹A directed multigraph is *weakly connected* if replacing all its directed edges with undirected edges produces a connected (undirected) multigraph.

Definition 10 (Match of a Generalized Pattern Query). *Let the generalized pattern query $G_P = (V_P, E_P)$, where $V_P = \{u_1, u_2, \dots, u_n\}$ and $n = |V_P|$. Also let $G = (V, E, f_V, f_E)$ be an attributed multigraph and $V' \subseteq V$. The tuple (v_1, \dots, v_n) , for $v_i \in V'$ and $1 \leq i \leq n$, is a match of G_P in G iff there exists a bijective function $f : V_P \rightarrow V'$ such that:*

1. $v_i = f(u_i)$ and $v_i \sim u_i$ for every $1 \leq i \leq n$.
2. $f(u_j)$ is (u_i, u_j, e) -reachable from $f(u_i)$ for every $e = (u_i, u_j) \in E_P$, $1 \leq i, j \leq n$.

We say that (v_1, \dots, v_n) is a match of G_P in G under f .

In this thesis, we study the next problem:

Problem 4 (Generalized Pattern Query Problem (GPQP)). *Given a generalized pattern query $G_P = (V_P, E_P)$ and an attributed multigraph $G = (V, E, f_V, f_E)$, the Generalized Pattern Query Problem (GPQP) consists of finding the set $G_P(G)$ of all the matches of G_P in G , i.e.,*

$$G_P(G) = \{(f_k(u_1), \dots, f_k(u_n)) \mid f_k \in \mathcal{M}_{G_P \rightarrow G}\},$$

where $\mathcal{M}_{G_P \rightarrow G}$ denotes the set of mapping functions $f_k : V_P \rightarrow V'$, for a set $V' \subseteq V$, that yield matches of G_P in G .

5.2.3. Example

An example of a generalized pattern query is presented in Figure 5-2(a). In particular, the query searches for photos in which Alice is tagged with two people: one of her colleagues and another female; furthermore, the colleague must be within Mary's network (they must be connected through *friend*-edges). Figure 5-2(b) shows the output of this query on the attributed multigraph of Figure 5-1. In this case, all the edges except e_1 represent single edge predicates; thus their matches are edges in the attributed multigraph. As for e_1 , we can see in Figure 5-1 that there is a path between Mary and Nick that satisfies the regular expression $([type = friend][type = person])^*[type = friend]$. The support of predicate attributes on both nodes and edges, as well as the expressive power of the reachability expressions, allows us to construct many useful and complex queries.

5.2.4. Discussion

Generalized pattern queries possess rich expressive power. It is a strict superset of reachability queries [40, 151, 84, 127, 128] and pattern match queries [159, 161]. We can use it to express reachability queries, pattern match queries, and their combination. In particular, a reachability query (RQ) is a special case of generalized pattern queries that only has two nodes (associated to predicates) and one edge (associated to a reachability expression). While a RQ establishes a single reachability requirement, generalized pattern queries allow multiple (u, v, ρ) -reachability

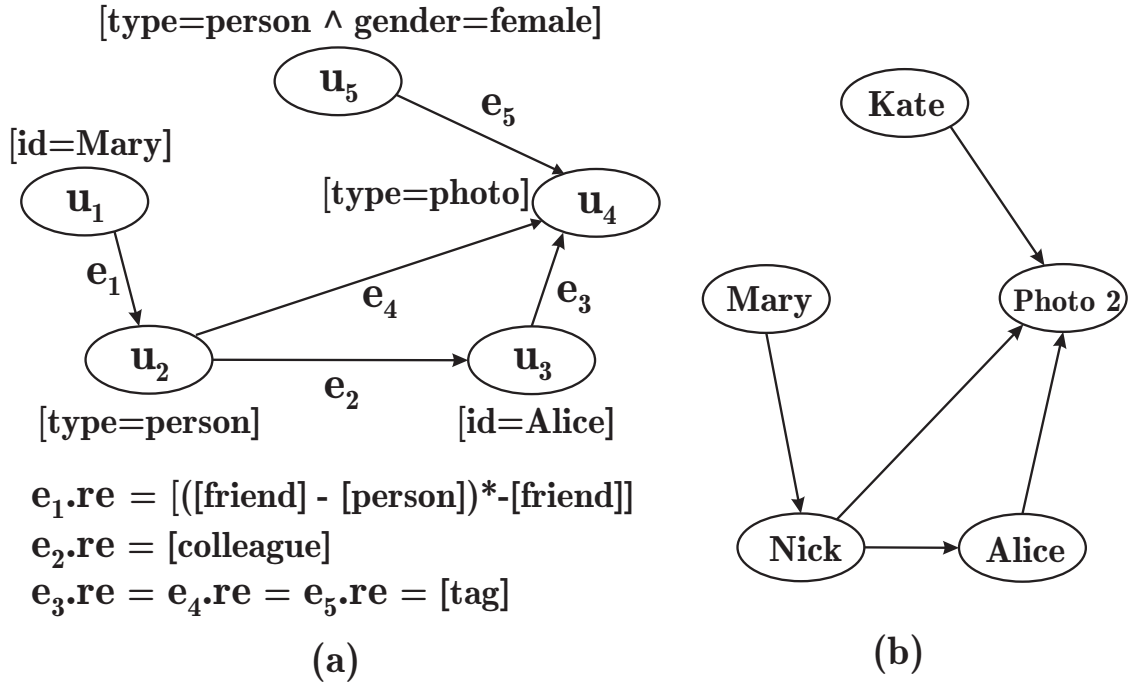


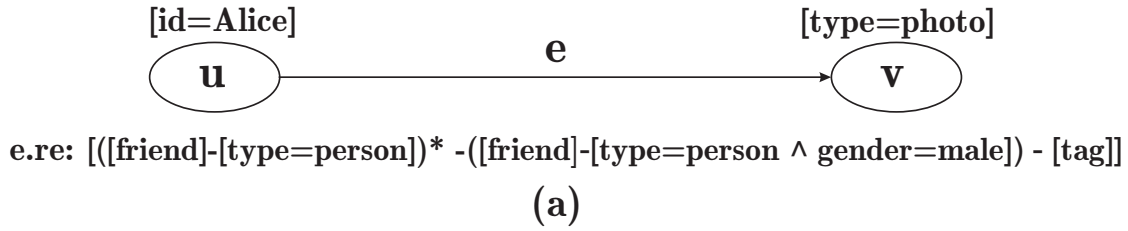
Figure 5-2.: Example of generalized pattern queries: (a) a generalized pattern query; (b) output obtained for the attributed multigraph of Figure 5-1.

requirements that are co-related. We support reachability queries with attribute predicates on both intermediate edges and nodes, which had not been supported by previous models.

For example, if we want to know in which photos there are men within Alice’s network, we can use a query like the one in Figure 5-3(a). The output of the query on the multigraph of Figure 5-1 includes $(Alice, Photo1)$ and $(Alice, Photo2)$. The set of the connecting paths associated to these results is presented in Figure 5-3(b). Furthermore, if we also want to retrieve the names of the men (tagged in the photos), we can use the query in Figure 5-4(a). The output of such query on the graph of Figure 5-1 is composed by $(Alice, Nick, Photo1)$ and $(Alice, Nick, Photo2)$.

A pattern match query, associated to subgraph isomorphism, is also a special case of generalized pattern queries where $e.re$, for all $e \in E_P$, corresponds to a single edge predicate. For instance, let us consider the query in Figure 5-5: it finds the friends of Alice that are tagged with her in a photo and also retrieve the photos. The output of this query on the graph of Figure 5-1 is presented in Figure 5-5(b). This result is a subgraph of the attributed multigraph that is isomorphic to the query and satisfies all the predicates.

Generalized pattern queries can also represent queries that cannot be modelled by either reachability or subgraph isomorphism queries. Specifically, generalized pattern queries are graphs that establish the predicates on a set of nodes of interest (through node predicates) and the reachability requirements among them (through highly-expressive regular expressions associated to the edges).



$$W_G(u, v, \rho) = \{ \langle \text{Alice, friend, Mike, friend, Rose, friend, Nick, tag, Photo 1} \rangle, \\ \langle \text{Alice, friend, Mike, friend, Rose, friend, Nick, tag, Photo 2} \rangle \}$$

(b)

Figure 5-3.: Example of a reachability query expressed as a generalized pattern query. (a) Query. (b) Set of the connecting paths associated to the output of this query on the graph of Figure 5-1.

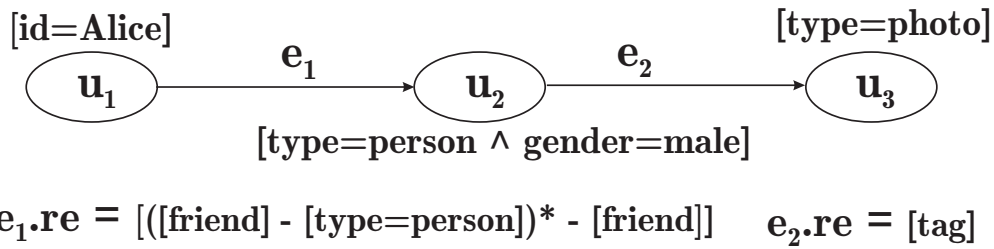


Figure 5-4.: Example of a reachability query with intermediate nodes of interest expressed as a generalized pattern query.

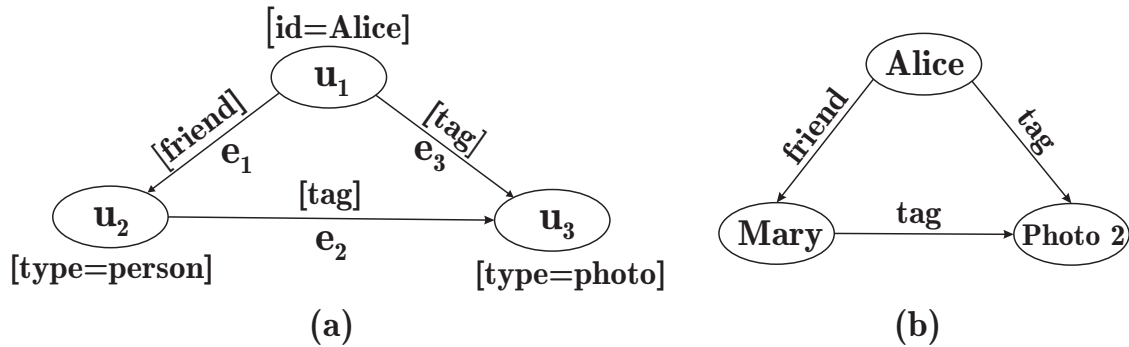


Figure 5-5.: Example of a pattern match query expressed as a generalized pattern query: (a) query graph; (b) output obtained for the attributed multigraph of Figure 5-1.

We show an example in Figure 5-2(a); this query cannot be expressed by either of the previous query models. In the next sections, we present an algorithm that finds all the matches of a generalized pattern query; hence, this algorithm also solves reachability queries, pattern match queries, and more.

6. Linearization on Generalized Pattern Queries

In this chapter, we extend the concept of *linearization* presented in Section 3.1 to represent generalized pattern queries in a linear manner. We then define *match of a query linearization* against the attributed multigraph, and show that each match of the query linearization corresponds to a match of our query. Therefore, processing our query requires two steps: (1) linearize the query, and (2) find the matches of the query linearization. In Section 6.1, we present the main definitions and prove the correctness of our approach. Then, in Section 6.2, we propose an algorithm that takes into account the attributed multigraph statistics to produce a query linearization that entails low time requirements during the matching phase.

6.1. Query Linearization

In this section, we present the definition of *query linearization* and how it can be used to solve the generalized pattern queries.

Definition 11 (Query Linearization). *Let $G_P = (V_P, E_P)$ be a generalized pattern query. An undirected walk $p = p_{1\dots\ell}$ on G_P is a query linearization of G_P iff:*

1. p_i is a node $v \in V_P$ if i is odd, $1 \leq i \leq \ell$.
2. p_i is an edge $e \in E_P$ if i is even, $1 \leq i \leq \ell$, such that either $e = (p_{i-1}, p_{i+1})$ or $e = (p_{i+1}, p_{i-1})$.
3. Each node $v \in V_P$ and each edge $e \in E_P$ appears at least once in p .

Notice that this definition is similar to Definition 2. The difference is that, as the graph is a generalized pattern query, the nodes are associated to attributes and each edge defines a (u, v, ρ) -reachability requirement. Then, the objective of a query linearization p of G_P is representing the structure, the reachability requirements and the attribute predicates of G_P in a linear manner. We use an alternating sequence of adjacent nodes (associated to predicates) and edges (associated to reachability expressions) that starts and ends at a node. All the nodes and edges in G_P must appear in p at least once so that the complete set of adjacency relations in E_P is represented.

We use undirected walks, as there may not exist a directed walk including all the nodes and edges. However, the direction of each edge $e \in E_P$ is denoted in the query linearization as $e.direction$ and is considered during the matching phase. Namely, for the edges p_i in p , $p_i.direction = forward$ if $p_i = (p_{i-1}, p_{i+1})$ and $p_i.direction = backward$ if $p_i = (p_{i+1}, p_{i-1})$, for even values of i in $1 \leq i \leq \ell$. We refer to them as *forward* and *backward edges*, respectively. A forward edge p_i establishes a (p_{i-1}, p_{i+1}, p_i) -reachability requirement; similarly, a backward edge p_i establishes a (p_{i+1}, p_{i-1}, p_i) -reachability requirement, for even values in $1 \leq i \leq \ell$. Then, the forward and backward edges are matched in the attributed multigraph using the forward and backward approach, respectively.

There may be more than one query linearization that represents the same generalized pattern query. Figure 6-1 depicts an example of two possible linearizations for the query in Figure 5-2(a).

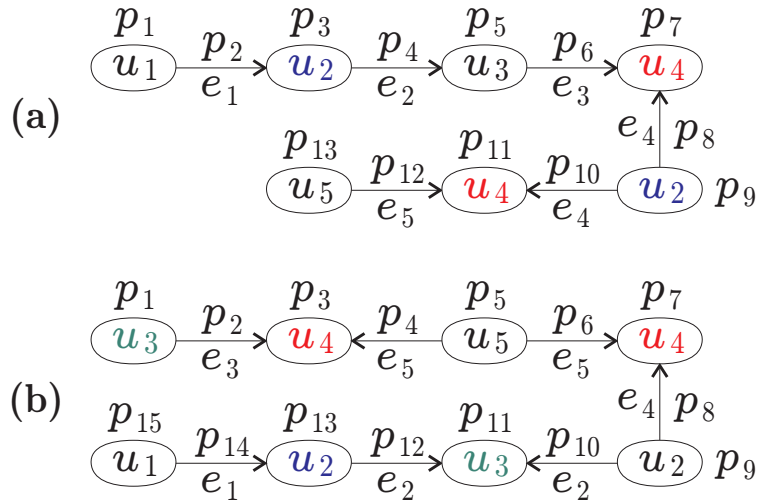


Figure 6-1.: Two linearizations for the graph presented in Figure 5-2(a).

Because processing a walk is simpler than processing a graph, we use query linearization to solve the generalized pattern queries problem by comparing undirected walks of G with a query linearization p . Then, we define *match of a query linearization*:

Definition 12 (Match of a Query Linearization). *Consider the generalized pattern query $G_P = (V_P, E_P)$ and an attributed multigraph $G = (V, E, f_V, f_E)$. Also, let $p = p_{1..l}$ be a query linearization of G_P and $q = q_{1..k}$ be an undirected walk on G where $k \geq l$. Furthermore, let q' be a subsequence of nodes in q such that $q' = (q'_1, q'_3, \dots, q'_l) = (q_{s_1}, q_{s_3}, \dots, q_{s_l})$ where $s_1 = 1$, $s_l = k$ and $s_i < s_{i+2}$ for odd values of i in $1 \leq i < l$. Then, the subsequence q' of nodes is a match of the query linearization p iff:*

1. There exists a bijective mapping function $f : V_P \rightarrow V'$, where V' is the set of nodes in the subsequence q' , such that $q_{s_i} = f(p_i)$ and $q_{s_i} \sim p_i$, for odd values of i , $1 \leq i \leq l$.

2. $q_{s_{i+2}}$ is (p_i, p_{i+2}, p_{i+1}) -reachable from q_{s_i} for odd values of i , $1 \leq i < \ell$, for which $p_{i+1}.direction = forward$. We denote this as $q_{s_i} \xrightarrow{p_i, p_{i+2}} q_{s_{i+2}}$.
3. q_{s_i} is (p_{i+2}, p_i, p_{i+1}) -reachable from $q_{s_{i+2}}$ for odd values of i , $1 \leq i < \ell$, for which $p_{i+1}.direction = backward$. We also denote this as $q_{s_i} \xleftarrow{p_i, p_{i+2}} q_{s_{i+2}}$.

We say that q' is a match of p in G under function f .

Notice that a match $q' = (q_{s_1}, q_{s_3}, \dots, q_{s_\ell})$ is a subsequence of nodes contained in an undirected walk q on G ; such nodes are connected in a manner that satisfies the structural constraints, the predicates and the reachability requirements associated to the query linearization $p = p_{1 \dots \ell}$. Specifically, Condition (1) validates the predicates established by the nodes in the query linearization and their bijective association with the nodes in q' . Furthermore, Conditions (2) and (3) validate the reachability requirements established by the edges in the linearization. It is important to remark that for edges whose direction is *backward*, the reversal of its associated reachability expression must be considered. An example of a match of the query linearization $p = p_{1 \dots 13}$ of Figure 6-1(a) on the attributed multigraph of Figure 5-1 is presented in Figure 6-2. Specifically, the match $q' = (Mary, Nick, Alice, Photo2, Nick, Photo2, Kate)$ is contained in the undirected walk $q = q_{1 \dots 19}$.

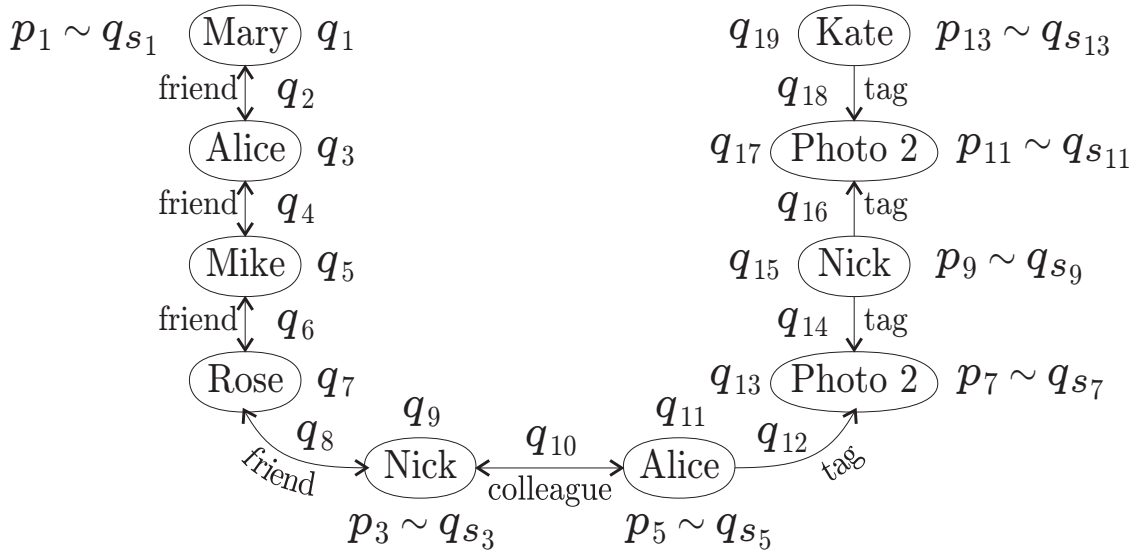


Figure 6-2.: $q' = (Mary, Nick, Alice, Photo2, Nick, Photo2, Kate)$ is a match of the the linearization $p = p_{1 \dots 13}$ presented in Figure 6-1(a). It is equivalent to finding a match $(Mary, Nick, Alice, Photo2, Kate)$ of G_P in Figure 5-2(a).

The core idea of our approach to find all the matches of the query G_P in G is as follows. Let p be a query linearization of G_P . Recall that, p represents the structural requirements, the reachability requirements and the predicates of G_P . If a subsequence q' of nodes in G is a match of p , then q' satisfies the requirements and predicates established in p (and, hence, in G_P); thus, q' is an

occurrence of G_P in G . More formally, the following theorem allows us to find the matches of a generalized pattern query in an attributed multigraph by means of query linearization.

Theorem 6. *Consider the generalized pattern query $G_P = (V_P, E_P)$, where $V_P = \{u_1, u_2, \dots, u_n\}$, and an attributed multigraph $G = (V, E, f_V, f_E)$. Also, let $p = p_{1\dots\ell}$ be a query linearization of G_P and $V' \subseteq V$. Then, the subsequence q' of nodes is a match of p under function $f : V_P \rightarrow V'$, where V' is the set of nodes contained in q' , if and only if $(f(u_1), \dots, f(u_n))$ is a match of G_P under bijective mapping function f in G .*

Proof. Let us consider the attributed multigraph $G = (V, E, f_V, f_E)$, the generalized pattern query $G_P = (V_P, E_P)$ and the query linearization $p = p_{1\dots\ell}$ of G_P . Furthermore, $V_P = \{u_1, u_2, \dots, u_n\}$ and $n = |V_P|$. Also, let V' be a subset of n nodes in G ($V' \subseteq V$). In order to prove the theorem, we need to show that (i) if q' is a match of p in G under the bijective function $f : V_P \rightarrow V'$, then $(f(u_1), \dots, f(u_n))$ is a match of G_P in G under f ; and (ii) if $(f(u_1), \dots, f(u_n))$ is a match of G_P in G under function $f : V_P \rightarrow V'$, then q' is a match of p in G under f .

First we prove (i). According to Definition 12, if q' is a match of p in G , there exists a bijective mapping function $f : V_P \rightarrow V'$ such that $q_{s_i} = f(p_i)$ and $q_{s_i} \sim p_i$, for odd values of i , $1 \leq i \leq \ell$ (condition (1) of Definition 12). Given that p contains all the nodes in V_P (see Definition 11), condition (1) of Definition 10 is satisfied. Furthermore, because p also contains all the adjacency relations of G_P , conditions (2) and (3) of Definition 12 imply that for every pair $e = (u_1, u_2) \in E_P$, $f(u_2)$ is (u_1, u_2, e) -reachable from $f(u_1)$ (condition (2) of Definition 10).

Now we prove (ii). Let us consider $V' = \{v_1, v_2, \dots, v_n\}$. According to Definition 10, if (v_1, \dots, v_n) is a match of G_P in G , then there exists a bijective mapping function $f : V_P \rightarrow V'$ such that $v_i = f(u_i)$ and $v_i \sim u_i$, for all $1 \leq i \leq n$ (condition (1) of Definition 10). Then, condition (1) of Definition 12 is satisfied. Moreover, for every pair $e = (u_1, u_2) \in E_P$, $f(u_j)$ is (u_i, u_j, e) -reachable from $f(u_i)$ (condition (2) of Definition 10). Given that all the edges in E_P are contained at least once in p (see Definition 11), for each $e = (u_i, u_j) \in E_P$, there exists at least a walk $\langle f(u_i), \dots, f(u_j) \rangle \in \mathcal{W}_G(u_i, u_j, e)$. Therefore, conditions (2) and (3) of Definition 12 are satisfied as well. □

Notice that this theorem is based on the following facts: (i) a query linearization represents all the adjacency relations of the corresponding generalized pattern query; and (ii) the matches of both the generalized pattern query and the query linearization involve the existence of a bijective mapping of the nodes that satisfies the node predicates and the reachability requirements.

In other words, this theorem states that the matches of the generalized pattern query $G_P = (V_P, E_P)$ in the attributed multigraph $G = (V, E, f_V, f_E)$ are defined with the same bijective mapping functions of the matches of p in G , where p is a query linearization of G_P . Each match of

p in G is associated to a match of G_P in G under a common function f . For example, for the attributed multigraph in Figure 5-1, the mapping function of both p and G_P (presented in Figures 6-2 and 5-2(b), respectively) are associated to the same mapping function:

$$f : (u_1, u_2, u_3, u_4, u_5) \rightarrow (Mary, Nick, Alice, Photo2, Kate)$$

Then, finding the matches of G_P is equivalent to finding the matches of p in G . Based on this, our approach to find the matches of a generalized pattern query consists of two steps: computing a linearization p of G_P and finding the matches of such linearization in G . We describe each of these steps in the next sections.

6.2. Enhanced Graph Linearization Algorithm — E-GLA

In this section, we present an algorithm for identifying a query linearization with low matching cost, out of many possible linearizations, as illustrated in Figure 6-1. In particular, our algorithm is called *Enhanced-Graph Linearization Algorithm* (E-GLA).

6.2.1. Baseline: GLA for Length-Optimality

As a baseline, we consider GLA (Graph Linearization Algorithm), which was presented in Section 3.3. This algorithm aims at selecting a *length-optimal* linearization. For example, Figure 6-1(a) is a length-optimal linearization: a linearization with minimum length among all possible linearizations (see Definition 4). As exact computation incurs prohibitive cost, GLA implements simple heuristics on a depth-first search (DFS) of the query graph. Specifically, the search can be represented as a DFS tree. The main heuristics of GLA is to cover the nodes in the highest levels of the DFS tree to keep the depth minimal; this is done by prioritizing visits to the nodes with minimum number of unexplored edges when there are choices. Its heuristics achieve 2-approximate optimality (see Theorem 3).

Though length-optimality is a reasonable cost model for the isomorphism problem, this is not as effective in a more complex query model like ours where selectivity varies. Specifically, it is important to take into account that we match the query linearization against the attributed multigraph using a depth-first search (DFS), where the top levels of the DFS tree correspond to the first elements of the query linearization. Then, we want to put the predicates with fewer matches at the beginning of the query linearization in order to prune the search space by detecting mismatches at an early stage of the process.

To illustrate this, let us consider the generalized pattern query of Figure 5-5 and the attributed multigraph of Figure 5-1. A query linearization can be started from any of the nodes in the generalized pattern query; however, u_1 , u_2 and u_3 have 1, 6 and 3 matches during the matching phase,

respectively. Therefore, we start the query linearization from u_1 in order to consider only one DFS search tree in the matching phase. For the next steps, let us consider the following statistics: (i) the average number of photos a person is tagged in is 1,16; and (ii) the average number of friends a person has is 1,66. These statistics are obtained by simple counting on the attributed multigraph of Figure 5-1. Hence, it is better to continue the query linearization with u_3 as it is likely to lead to fewer matches in the second level of the DFS search trees of the matching phase. Thus, the query linearization that most likely reduces the search space is $u_1e_3u_3e_2u_2e_1u_1$. We already observe the cost benefit of using statistics in this example even though the graph is tiny. It becomes much more important when the graph size increases, as different query linearizations may incur orders of significantly different costs.

6.2.2. Key Ideas

In this section, we develop the key ideas of the Enhanced-Graph Linearization algorithm (E-GLA). The cost of the matching phase is represented as DFS search trees where the roots are the possible matches of p_1 . The goal of E-GLA is pruning such trees by placing the nodes in V_P with fewer matches in G (i.e., nodes with low selectivity) at the beginning of the query linearization p . As it was mentioned in last section, this causes that such few matches are placed at the top levels of the DFS search trees during the matching phase. Hence, the search space is pruned at an early stage by avoiding partial matches of p that will not lead to complete matches.

More formally, we define the *selectivity* for each node $u \in V_P$ in the generalized pattern query. The selectivity of the node u with respect to the attributed multigraph $G = (V, E, f_V, f_E)$ is denoted as $select_G(u)$. It is calculated as the probability of selecting a node from G that satisfies the predicate $u.pred$. In this sense, E-GLA gives preference to the nodes in the query with the lowest selectivity for starting (or continuing) the linearization.

The traversal performed by E-GLA on the generalized pattern query is a DFS search with the following heuristics: (1) the traversal starts from the node with the lowest selectivity; (2) the unexplored edges that lead to already explored nodes are visited before than the ones that lead to unexplored nodes; and (3) the edges that lead to unexplored nodes are considered sorted, in ascending order, on the selectivity of such nodes. Notice that heuristics (1) and (3) aim to optimize for selectivity, by placing the nodes of the generalized pattern query that will have many mismatches at the beginning of the query linearization; this helps to prune the search space at an early stage of the matching phase. Moreover, heuristic (2) optimizes for the length of the linearization as in GLA. Thus, E-GLA produces a linearization that not only considers the length but also takes into account the statistics of the attributed multigraph to reduce the matching-phase time.

6.2.3. Algorithm

The pseudocode in Figure 6-3 describes E-GLA. Each node and edge has a boolean flag that indicates whether it has been explored. All these flags are initialized with *false*. Furthermore, the number of unexplored nodes/edges is stored in variable *unexplVE*. This variable is used to avoid reinserting graph elements in the backtracking of the DFS search tree when there are no unexplored elements left. The query linearization is represented as a list *p*.

Notice that the selectivity of each node *v*, denoted as *v.selectivity*, is populated by means of the function COMPUTESELECTIVITY(). The parameters of this function are the corresponding node predicate *v.pred* and the attributed multigraph $G = (V, E, f_V, f_E)$. This function can be implemented in different ways: (i) counting the nodes in the attributed multigraph that satisfy *v.pred*; (ii) using previous information on the attributed multigraph statistics; or (iii) designing formulas based on the distribution of the node attributes. Thus, the complexity of this function can vary from constant to linear on the number of nodes in the attributed multigraph.

With such selectivity information, we find the node *u* with lowest selectivity (line 5, Figure 6-3). Then, the DFS traversal is performed by calling the recursive procedure STATSTRAVERSE() over node *u* (see Figure 6-4). This procedure is essentially the same as TRAVERSEGRAPH() (the recursive procedure that performs the traversal for GLA); the difference is that edges that lead to unexplored nodes are sorted, in ascending order, on their selectivity (lines 7 – 8, Figure 3-2). The algorithm terminates when the first call to STATSTRAVERSE() finishes.

Algorithm 7: E-GLA Algorithm

Input: $G_P = (V_P, E_P)$, $G = (V, E, f_V, f_E)$ **Output:** *p*

1. **for every** $e \in E_P$ **do** $e.Explored \leftarrow false$
 2. **for every** $v \in V_P$ **do**
 3. $v.Explored \leftarrow false$
 4. $v.selectivity \leftarrow ComputeSelectivity(v.pred, G)$
 5. **choose** $u \in V_P$ **with** $min(u.selectivity)$
 6. $p \leftarrow \langle \rangle$, $unexplVE \leftarrow |V| + |E|$
 7. $StatsTraverse(G_P, u, p, unexplVE)$
 8. **return** *p*
-

Figure 6-3.: E-GLA algorithm.

Example. To illustrate how E-GLA works, Table 6-1 shows the selectivity of the nodes in the query of Figure 5-2(a). This can be obtained from the statistics on the attributed multigraph of Figure 5-1, e.g., the number of nodes that satisfy each predicate divided by the total number of

Algorithm 8: STATSTRAVERSE() Procedure**Input:** $G_P = (V_P, E_P), u, p, unexplVE$

-
1. $p.Add(u), u.Explored \leftarrow true, unexplVE--$
 2. **for every** $e \in E$ such that $e = (u, v)$ or $e = (v, u)$ **do**
 3. **if** $\neg e.Explored \wedge v.Explored$ **then**
 4. $p.Add(e), e.Explored \leftarrow true, unexplVE--, p.Add(v)$
 5. **if** $unexplVE > 0$ **do**
 6. $p.Add(e), p.Add(u)$
 7. **while** there are unexplored edges $e = (u, v)$ or $e = (v, u)$
 8. **choose** e **with** $min(v.selectivity)$
 9. $p.Add(e), e.Explored \leftarrow true, unexplVE--$
 10. $StatsTraverse(G_P, v, p, unexplVE)$
 11. **if** $unexplVE = 0$ **then break**
 12. $p.Add(e), p.Add(u)$
-

Figure 6-4.: STATSTRAVERSE() procedure.

nodes. E-GLA may start the linearization from the nodes with lowest selectivity, namely u_1 and u_3 : if the starting node is u_1 , the linearization obtained is the one presented in Figure 6-1(a); if the starting node is u_3 , the linearization is slightly longer as presented in Figure 6-1(b).

Table 6-1.: Selectivity of the nodes in the generalized pattern query of Figure 5-2(a) with respect to the attributed multigraph of Figure 5-1.

| Node | u_1 | u_2 | u_3 | u_4 | u_5 |
|-------------|-------|-------|-------|-------|-------|
| Selectivity | 1/9 | 2/3 | 1/9 | 1/3 | 4/9 |

6.2.4. Correctness Proof

The correctness of this algorithm is proven in the following theorem:

Theorem 7. *The Enhanced-Graph Linearization Algorithm (E-GLA) outputs a query linearization of the input generalized pattern query $G = (V_P, E_P)$.*

Proof. Let $G = (V_P, E_P)$ be a generalized pattern query and $p = p_{1\dots\ell}$ the output walk produced by E-GLA. Notice that, disregarding the predicates associated to the nodes in V_P and the reachability expressions associated to the edges in E_P , G_P is a regular graph. Similarly, a query linearization of G_P , without the node predicates and the reachability expressions of the edges, is a simple linearization (as defined in Definition 2). Then, we prove Theorem 7 by showing that the

output walk produced by E-GLA is a linearization.

Let us recall that the recursive procedure STATSTRAVERSE() is essentially the same as the recursive procedure TRAVERSEGRAPH(). These procedures are in charge of the DFS traversal performed by GLA and E-GLA, respectively. Their only difference is the order in which adjacent edges that lead to unexplored nodes are considered. Hence, even though the traversal performed by E-GLA can be different from the one of GLA, such traversal maintains the same properties. Namely, conditions (1), (2) and (3) of Definition 2 are satisfied due to Theorem 2. That is, the output walk p is a linearization of G_P . Thus, if the corresponding predicates and reachability expressions are respectively included in the nodes and edges of the walk p , then such walk is a *query linearization* of G_P . \square

6.2.5. Length of E-GLA Linearization

In this section, we formally discuss the strength of E-GLA. Theorem 8 shows that given the generalized pattern query $G_P = (V_P, E_P)$, the length of the linearization generated by E-GLA is at most 2 times the length of a length-optimal linearization. That is, E-GLA optimizes for selectivity, yet gives the same asymptotic length guarantee as GLA.

Theorem 8. *E-GLA is 2-approximate with respect to the length of a length-optimal linearization.*

Proof. Let $G_P = (V_P, E_P)$ be a generalized pattern query and $p = p_{1\dots\ell}$ the output walk produced by E-GLA. Any query linearization algorithm, including length-optimal algorithms, must traverse each edge of the multigraph at least once. Thus, the number of edges in a query linearization of G_P is at least $|E_P|$. Since a linearization has the format of alternating between nodes and edges, a query linearization with k edges has $k + 1$ nodes. Hence, the optimal query linearization p^* has at least $|E_P|$ edges and $|E_P| + 1$ nodes. Therefore, $|p^*| \geq 2|E_P| + 1$.

Like GLA, E-GLA also visits any edge at most twice during the DFS traversal. This is because, when procedure STATSTRAVERSE() is executed over node u , an unexplored edge e that leads to any explored or unexplored node v is added once into p (lines 4 and 9, respectively, Figure 6-4). If after executing the next instructions there are still unvisited graph elements, it is necessary to go back to u through e ; this means that e and u are added into p again (lines 6 and 12, Figure 6-4). After this, e is not visited ever again given that only unexplored edges are considered (lines 3 and 7, Figure 6-4). Therefore, the number of edges in the query linearization is at most $2|E_P|$. Again, since a linearization has the format of alternating between nodes and edges, the query linearization p^{E-GLA} has at most $2|E_P|$ edges and $2|E_P| + 1$ nodes. Therefore, we have $|p^{E-GLA}| \leq 4|E_P| + 1$. It leads to the approximation ratio of E-GLA,

$$\frac{|p^{E-GLA}|}{|p^*|} \leq \frac{4|E_P| + 1}{2|E_P| + 1} \leq 2.$$

□

This worst-case approximation ratio is obtained by comparing the length of E-GLA linearizations with a lower bound where it is assumed that each edge in the query appears only once in the query linearization. However, even a length-optimal linearization may not achieve this lower bound for many graph structures, because it may require including an element more than once. Thus, for many cases, in practice, E-GLA linearizations are much closer to length-optimal linearizations than the ratio given by this theorem.

6.2.6. Complexity Analysis

This section studies the computational complexity of E-GLA to generate a query linearization $p = p_{1\dots\ell}$ of the generalized pattern query $G_P = (V_P, E_P)$ for the attributed multigraph $G = (V, E, f_V, f_E)$, where $n = |V|$ and $m = |E|$.

Time Complexity. Initializing the edges and nodes as *unexplored* takes $O(|E_P|)$ and $O(V_P)$, respectively (lines 1–3, Figure 6-3). As discussed in Section 6.2.3, the time complexity for computing the selectivity of each node can vary from constant to linear on n . In this analysis, we assume that we have information on the distribution of the node attributes; thus, the cost of executing this procedure is $O(1)$ per node and, hence, $O(|V_P|)$ for all nodes in G_P (line 4, Figure 6-3). Finding the node with lowest selectivity takes $O(|V_P|)$ (line 5, Figure 6-3). Initializing p and *unexplVE* takes constant time (line 6, Figure 6-3).

Notwithstanding, the time complexity of E-GLA is dominated by the undirected walk traversed (line 7, Figure 6-3) which corresponds to the length of the query linearization. In Section 6.2.5, we showed that p has at most $2|E_P|$ edges and $2|E_P| + 1$ nodes. Each insertion takes constant time as it is always done at the end of p . But when a node is inserted for the first time, it is necessary to consider the unexplored adjacent edges e that lead to unexplored nodes v sorted on their selectivity (lines 7–8, Figure 6-4). This sorting operation takes $O(d \lg d)$, where d is the maximum degree of the nodes in G_P (including both incoming and outgoing edges). Thus, the time complexity of E-GLA is $O(2|E_P| + |V_P|(d \lg d)) = O(|E_P| + d|V_P| \lg d)$.

Space Complexity. The space complexity is given by the list that stores the linearization, i.e., by the length of the linearization. Because the linearization can have at most $2m$ edges and $2m + 1$ nodes, the total space complexity is $\Theta(m)$.

7. Solution of Generalized Pattern Queries

We develop a matching algorithm for generalized pattern queries, which we call GPQM. This algorithm uses a query linearization $p = p_{1\dots\ell}$ of the generalized pattern query $G_P = (V_P, E_P)$ to find the matches of G_P in the attributed multigraph $G = (V, E, f_V, f_E)$. In Section 7.1, the high-level ideas of the algorithm are presented. Then, we go through the details in Section 7.2. Its correctness is proven in Section 7.3 while the complexity analysis is derived in Section 7.4. Finally, some experimental results are presented in Section 7.5.

7.1. Key Ideas

The GPQM algorithm searches for matches of the query linearization $p = p_{1\dots\ell}$ of the generalized pattern query $G_P = (V_P, E_P)$ in the attributed multigraph $G = (V, E, f_V, f_E)$. According to Theorem 6, each match represents a match of G_P in G . This section presents the key ideas of the algorithm.

We search all the possible undirected walks in G that may contain matches of a linearization $p = p_{1\dots\ell}$ in a DFS manner. Our search starts from p_1 : the roots of the candidate DFS search trees are the nodes that satisfy p_1 . Then, we try to associate each node p_i to every node u in these DFS search trees that satisfies both the node predicates and the reachability requirements. Each undirected walk from the root of the search tree to u is associated to an injective function f that establishes a mapping between the nodes in the linearization and certain nodes on the walk. The injective property ensures that two different nodes in p are not assigned to the same node $u \in V$; likewise, two different nodes in V are not associated to the same node in p .

More specifically, the query linearization $p = p_{1\dots\ell}$ is traversed from p_1 to p_ℓ while we consider the possible assignments for each p_i , $1 \leq i \leq \ell$ in the graph G . Let us assume that $f(p_i) = u$ was set. We recursively extend the current partial match under f where there are two cases:

Case 1: Node p_{i+2} is unassigned: We consider all the possible assignments $f(p_{i+2}) = v$ for all the nodes $v \in V$ such that $u \xrightarrow{p_i, p_{i+2}} v$ (or $u \xleftarrow{p_i, p_{i+2}} v$, if p_{i+1} is a backward edge). Let us establish the predicate $p'_i.pred : [ID = u.id]$. Because u was assigned to p_i , we can find the nodes v by

finding the nodes v such that $u \xrightarrow{p'_i p_{i+2}} v$ (or $u \xleftarrow{p'_i p_{i+2}} v$). Notice that, if p_{i+1} is a backward edge, solving $u \xrightarrow{p'_i p_{i+2}} v$ by taking the backward approach prunes the search space as there is only a single source, i.e., u . This is an advantage of using the query linearization approach instead of separately solving each reachability requirement and then computing their intersections. Furthermore, among the possible nodes v , we only consider the ones that are unassigned; this is to guarantee that f is injective. In case that there exists no node v that satisfies the requirements, the match cannot be extended. Otherwise, the process continues at p_{i+2} and each v .

Case 2: Node p_{i+2} is assigned to $v \in V$: There are two sub-cases. (a) The reachability requirement established by p_{i+1} has already been evaluated: it is not necessary to evaluate reachability again; we continue by considering p_{i+2} and v . (b) The reachability requirement established by p_{i+1} has not been evaluated: we establish the node predicates $p'_i.pred : [ID = u.id]$ and $p'_{i+2}.pred : [ID = v.id]$ and evaluate whether $u \xrightarrow{p'_i p'_{i+2}} v$ (or $u \xleftarrow{p'_i p'_{i+2}} v$, if p_{i+1} is a backward edge, which is evaluated using the backward approach). If the requirement is satisfied, the process continues at p_{i+2} and v . Otherwise, the match cannot be extended.

The above procedures guarantee that: (i) all the reachability requirements established by the edges in G_P are satisfied; (ii) all the possible mapping functions (or all the possible matches) are considered; and (iii) because of Case 2, the reachability requirement defined by each edge is evaluated only once. If the algorithm reaches a successful assignment for p_ℓ , then the algorithm reports that the corresponding mapping function f is associated to a match of G_P in G .

7.2. Algorithm

Figure 7-1 lists the pseudocode of GPQM. Three arrays are used to store the information of a partial (or full) match. (i) The mapping function is represented as the array f where each position is associated to a node $u \in V_P$, which will eventually contain the mapping of u . In the beginning, all the mappings are *undefined* (which we denote as *undef*). (ii) We have a boolean array h , where each position is associated to an edge $e \in E_P$ to establish whether the reachability requirement established by e has already been checked. (iii) We have a boolean array g where each node in V is associated to a position in the array. Specifically, $g[v] = true$, for a node $v \in V$, if a mapping for v has already been established (through f). Intuitively, all the positions of arrays g and h are initialized with *false*.

The algorithm starts by computing a query linearization $p = p_{1\dots\ell}$ of G_P and initializing the set \mathcal{R} of matches as empty. Then, all the nodes v in V such that $v \sim p_1$ are considered as roots of the DFS trees. In particular, the DFS traversal is performed by the recursive procedure `PROCESSNODE()` (see Figure 7-2). Each execution instance of this procedure considers a node p_i , a node $u \in V$ and a copy of f , g and h . It is assumed that u was assigned to $f(p_i)$ before the execution instance

was called. Thus, there is a partial match of $p_{1\dots i}$ in G under f . Then, what the procedure does is attempting to extend this partial match according to the cases presented in last section.

Algorithm 9: GPQM Algorithm

Input: $G_P = (V_P, E_P)$, $G = (V, E, f_V, f_E)$ **Output:** \mathcal{R}

1. $p = E - GLA(G_P, G)$, $\mathcal{R} = \emptyset$
 2. **for every** $v \in V_P$ **do** $f[v] \leftarrow undef$
 3. **for every** $e \in E_P$ **do** $h[e] \leftarrow false$
 4. **for every** $v \in V$ **do** $g[v] \leftarrow false$
 5. **for every** $u \in V$ **do**
 6. **if** $u \sim p_1$ **then**
 7. $f' \leftarrow copy(f)$, $g' \leftarrow copy(g)$, $h' \leftarrow copy(h)$
 8. $f'[p_1] = u$, $g'[u] = true$
 9. $ProcessNode(u, p, 1, f', g', h', G, \mathcal{R})$
 10. **return** \mathcal{R}
-

Figure 7-1.: GPQM algorithm.

We efficiently evaluate the reachability requirements in the query linearization by means of the function `FINDREACHABLENODES()` (see Figure 7-3). Let us consider an edge p_{i+1} in the query linearization. We first tackle the case where $p_{i+1}.direction = forward$. Then, in order to evaluate the corresponding (p_i, p_{i+2}, p_{i+1}) -reachability requirement, this function constructs a deterministic finite automaton (DFA) that accepts all the paths $\langle u, \dots, v \rangle$ such that $v \in V$ is (p_i, p_{i+2}, p_{i+1}) -reachable from $u \in V$. These walks are explored in a DFS manner. The roots of the DFS search trees are the nodes that satisfy $p_{i+1}.pred$. Starting from these nodes, we continue the DFS traversal constrained by the DFA. The process terminates when all the paths that lead to the final state, associated to $p_{i+2}.pred$, are considered.

If, on the contrary $p_{i+1}.direction = backward$, the (p_{i+2}, p_i, p_{i+1}) -reachability requirement is validated in a similar manner. However, we use the backward approach to enhance efficiency. Specifically, notice that if we have already processed a node p_i in the query linearization, then a particular node $u \in V$ such that $u \sim p_i$ has already been found. Hence, following the backward approach makes the search start from a single node (i.e., u) rather than all the possible nodes v such that $v \sim p_{i+2}$. Note that many paths starting from valid nodes v may not lead to u , so they should not be considered at this point. Then, the search is represented by a single DFS tree rooted at u . The DFA must accept the paths in G that satisfy the reversal of the corresponding reachability expression (i.e., $p_{i+1}.re^R$). Thus, such paths are traversed inversely: for each node on a path, the incoming edges are considered. For a given reachability requirement, disregarding the direction of its corresponding edge, the function `FINDREACHABLENODES()` returns the set of nodes $v \in V$

Algorithm 10: PROCESSNODE() Procedure**Input:** $u, p = p_{1\dots\ell}, i, f, g, h, G = (V, E, f_V, f_E), \mathcal{R}$

-
1. **if** $i = \ell$ **then** $\mathcal{R}.Add(f)$
 2. **else**
 3. $p'_i.pred : [ID = u.id]$
 4. **if** $f[p_{i+2}] = undef$
 5. $\mathcal{Q} \leftarrow FindReachableNodes(p'_i, p_{i+2}, p_{i+1})$
 6. **for every** $v \in \mathcal{Q}$ **do**
 7. **if** $g[v] = false$ **then**
 8. $f' \leftarrow copy(f), g' \leftarrow copy(g), h' \leftarrow copy(h)$
 9. $f'[p_{i+2}] \leftarrow v, g'[v] \leftarrow true, h'[p_{i+1}] \leftarrow true$
 10. $ProcessNode(v, p, i + 2, f', g', h', G, \mathcal{R})$
 11. **else**
 12. $v = f[p_{i+2}]$
 13. **if** $h[p_{i+1}] = true$ **then**
 14. $ProcessNode(v, p, i + 2, f, g, h, G, \mathcal{R})$
 15. **else**
 16. $p'_{i+2}.pred : [ID = v.id]$
 17. $\mathcal{Q} \leftarrow FindReachableNodes(p'_i, p'_{i+2}, p_{i+1})$
 18. **if** $\mathcal{Q} \neq \emptyset$
 19. $h \leftarrow copy(h), h'[p_{i+1}] \leftarrow true$
 20. $ProcessNode(v, p, i + 2, f, g, h', G, \mathcal{R})$
-

Figure 7-2.: PROCESSNODE() procedure.**Algorithm 11: FINDREACHABLENODES() Function****Input:** $u, v, e, G = (V, E, f_V, f_E)$ **Output:** \mathcal{Q}

-
1. **if** $e.direction = forward$ **then** $reachReq = (u, v, e)$
 2. **else then** $reachReq = (v, u, e)$
 3. $\mathcal{Q} \leftarrow QueryConstrainedDFA(reachReq, G, e.direction)$
 4. **return** \mathcal{Q}
-

Figure 7-3.: FINDREACHABLENODES() function.

that satisfy the corresponding reachability requirements with respect to a given $u \in V$.

Example. Figure 7-4 shows the DFS search tree traversed by GPQM to find the matches of the

query of Figure 5-2(a) on the attributed multigraph of Figure 5-1 by using the linearization of Figure 6-1(a). Notice that we associate each node in the linearization to nodes in the attributed multigraph that satisfy the reachability requirements. For example, the people associated to p_3 ($[type = person]$) are the ones that can be reached from *Mary* ($[id = Mary]$) through a path satisfying the expression $[([type = friend][type = person])^*[type = friend]]$. Such paths are explored using the DFS constrained by the automaton presented in Figure 7-5.

Moreover, the example demonstrates that both reachability requirement evaluation and injective assignments prune the search space. For instance, there is just one possible mapping for $p_{11} = u_4$ in each branch of the search; because $p_7 = u_4$, then the only possible mapping for p_{11} is the same mapping established for p_7 , i.e., *Photo 1* in the left branch and *Photo 2* in the right branch. The highlighted walk (notes with double lines in Figure 7-4), which has mappings for all the elements in the query linearization, corresponds to the undirected walk presented in Figure 6-2. The mapping function associated to this walk constitutes a match of p (and hence of G_P) in G (the one of Figure 5-2(b)).

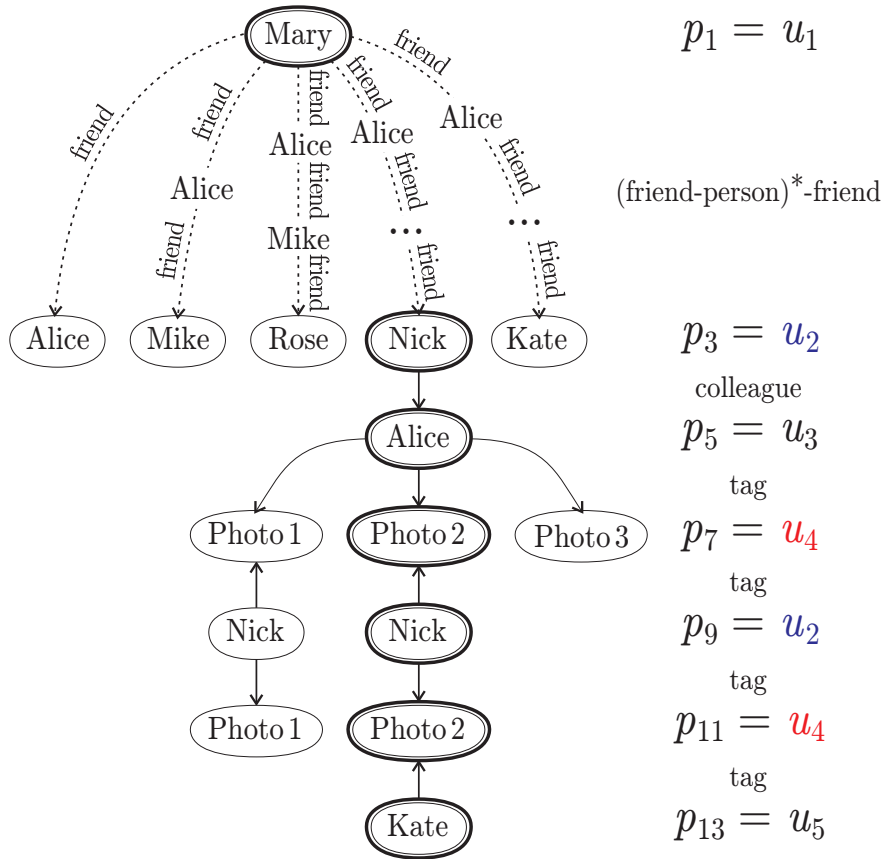


Figure 7-4.: DFS search tree traversed by GPQM to find the matches of the query of Figure 5-2(a) on the attributed multigraph of Figure 5-1 by using the linearization of Figure 6-1(a).

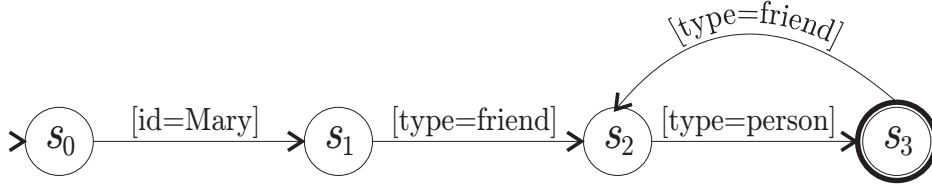


Figure 7-5.: Deterministic Finite Automaton (DFA) corresponding to the reachability expression $[[id = Mary]([type = friend][type = person])^*[type = friend][type = person]]$ in Figure 5-2(a).

7.3. Correctness Proof

We formally prove the correctness of the GPQM algorithm by means of the following theorem:

Theorem 9. *Given the attributed multigraph $G = (V, E, f_V, f_E)$ and the generalized pattern query $G_P = (V_P, E_P)$, the GPQM algorithm reports all the matches of G_P in G .*

Proof. We prove this theorem by showing the following invariant: when $u \in V$ and p_i are processed in the recursive procedure $\text{PROCESSNODE}()$, there exists a match of $p_{1..i}$ in G under some bijective function f . Let us denote such match as $(v_1, v_3, \dots, v_{i-2}, u)$, where $v_j \in V$. Then, $f(p_j) = v_j$, for odd values of j in $1 \leq j < i$, and $f(p_i) = u$. Next, we show that this condition holds throughout the execution of the algorithm.

- **Initialization:** Let us consider the pseudocode presented in Figure 7-1. Because of the execution of lines 6 and 8 before calling $\text{PROCESSNODE}(u, p_1)$ (in line 9), there exists a match (u) of (p_1) under f (i.e., $f(p_1) = u$).
- **Maintenance:** Let us assume that the invariant holds when $\text{PROCESSNODE}(u, p_i)$ is executed (see Figure 7-2). Thus, there exists a match $(v_1, v_3, \dots, v_{i-1}, u)$ of $p = p_{1..i}$ under a given function f . If $i < \ell$, we have to consider two cases: (i) $f(p_{i+2})$ has not been defined; and (ii) $f(p_{i+2}) = v$, where $v \in V$.

Let us consider first case (i). Notice that, because of lines 5 – 7, $\text{PROCESSNODE}(v', p_{i+2})$ is only called for unassigned nodes $v' \in V$ such that v' is (p_i, p_{i+2}, p_{i+1}) -reachable from u (or u is (p_{i+2}, p_i, p_{i+1}) -reachable from v' , if p_{i+1} is a backward edge). Thus, adding $f(p_{i+2}) = v'$ (line 9, Figure 7-2) maintains the injective property of f and satisfies the corresponding reachability requirements. Hence, when $\text{PROCESSNODE}(v', p_{i+2})$ is executed (line 10, Figure 7-2), the invariant is satisfied given that $(v_1, v_3, \dots, v_{i-1}, u, v')$ is a match of $p_{1..i+2}$.

For the case (ii), we have two sub-cases: the case where the reachability requirement established by p_{i+1} has already been evaluated and the case where it has not. In the former, we need not change the function and the invariant will hold when $\text{PROCESSNODE}(v, p_{i+2})$ is called (line 14, Figure 7-2). In the latter, $\text{PROCESSNODE}(v, p_{i+2})$ is only called if the node v is (p_i, p_{i+2}, p_{i+1}) -reachable from u (or if u is (p_{i+2}, p_i, p_{i+1}) -reachable from v , if p_{i+1} is

a backward edge). Furthermore, the mapping function f is not changed. Therefore, when $\text{PROCESSNODE}(v, p_{i+2})$ is called (line 20, Figure 7-2), $(v_1, v_3, \dots, v_{i-1}, u, v)$ is a match of $p_{1\dots i+2}$; thus, the invariant holds.

- **Termination:** When a partial match is extended in any branch of the search and reaches position ℓ (line 1, Figure 7-2), we know that the function f contains a match of $p_{1\dots\ell}$ because of the invariant. Furthermore, all the matches of p in G are inserted in \mathcal{R} as all the possible mapping functions are considered. Namely, (i) the search is started from all the valid nodes (lines 5 – 9, Figure 7-1); and (ii) in all the stages of the search, all the possible injective mappings that satisfy the reachability requirements are considered (lines 5 – 10, 13 – 14 and 17 – 20, Figure 7-2). This set of matches also corresponds to the set of matches of G_P in G (see Theorem 6).

□

7.4. Complexity Analysis

This section establishes the worst-case time complexity of GPQM. As we know that subgraph isomorphism, a simpler problem than Problem 4, is NP-complete [70], matching generalized pattern queries is expensive in the worst case. However, we also show that GPQM often performs better in practice than its worst-case bound and elaborate the reason.

Let us consider the generalized pattern query $G_P = (V_P, E_P)$ and the attributed multigraph $G = (V, E, f_V, f_E)$, where $n = |V|$ and $m = |E|$. As it was shown in Section 6.2.6, finding a convenient linearization $p = p_{1\dots\ell}$ of G_P takes $O(d|E_P| \lg d)$ where d is the maximum degree of the nodes in G_P (line 1, Figure 7-1). The initialization of f , g and h takes $O(|V_P|)$, $O(|E_P|)$ and $O(n)$, respectively (lines 2 – 4, Figure 7-1). Notwithstanding, these costs are insignificant with respect to the cost incurred by repeatedly executing the procedure $\text{PROCESSNODE}()$.

In order to calculate the time complexity of GPQM, we first find an upper bound for the number of executions of the recursive procedure $\text{PROCESSNODE}()$. This number is equal to the number of nodes that are associated to a given p_i in the DFS search trees. The next theorem establishes an upper bound.

Theorem 10. *Let $p_{1\dots\ell}$ represent a linearization of a generalized pattern query, and let $G = (V, E, f_V, f_E)$ represent an attributed multigraph, where $n = |V|$. A DFS search tree that represents the traversal of G done by GPQM has $O(n^{\lceil \ell/2 \rceil - 1})$ nodes associated to the different nodes p_i in the linearization, for odd values of i in $1 \leq i < \ell$.*

Proof. Let us consider the undirected walks from the root of a DFS search tree to the lowest leaves, i.e. the undirected walks that have $\lceil \ell/2 \rceil$ nodes associated to a p_i . These walks are the

ones that determine the height of the tree with the greatest number of executions of the procedure `PROCESSNODE()`.

Next we calculate the number of possible assignments for each node p_i that requires the execution of the procedure `PROCESSNODE()`. There is one possible assignment for p_1 in a given DFS search tree. According to the reachability requirement established by p_2 , there can be up to $n - 1$ nodes that could be assigned to p_3 . This is because the node assigned to p_1 cannot be assigned to p_3 because of the injective requirement. Each of these $n - 1$ nodes that can be associated to p_3 may yield to $n - 2$ possible assignments for p_5 (again, the paths that lead to the nodes assigned to p_1 and p_3 are not considered); thus, the total number of possible assignments for p_5 in the DFS search tree is $O((n - 1)(n - 2))$ nodes. Similarly, the total number of assignments in the tree for p_7 is $O((n - 1)(n - 2)(n - 3))$ nodes. In general, p_i has at most $\prod_{j=1}^{\lfloor i/2 \rfloor} (n - j)$ possible assignments. Therefore, the total number of assignments in the tree for odd values of i in $1 \leq i < \ell$ is:

$$1 + \sum_{\substack{i=3 \\ i+=2}}^{\ell-2} \prod_{\substack{j=1 \\ j+=1}}^{\lfloor i/2 \rfloor} (n - j) = O(n^{\lfloor (\ell-2)/2 \rfloor}) = O(n^{\lfloor \ell/2 \rfloor - 1})$$

□

In the worst case, we can have at most n DFS search trees with these characteristics, by assigning each distinct node in G to p_1 . In other words, we obtain a different DFS search tree by starting the search from a different node of G . Thus, the total number of times that the procedure `PROCESSNODE()` can be executed is: $O(n \times n^{\lfloor \ell/2 \rfloor - 1}) = O(n^{\lfloor \ell/2 \rfloor})$.

The complexity of each execution of the procedure `PROCESSNODE()` is dominated by the execution of the function `FINDREACHABLENODES()` (lines 5 and 17, Figure 7-2). The complexity of this function is determined by the construction of the DFA and the DFS search constrained by it (line 3, Figure 7-3). This can be done, using the traditional technique [142], as follows. We convert the regular expression, whose length we denote as r , into an NFA with $O(r)$ nodes. Then, this NFA is converted into a DFA in $O(2^r)$ time. Let $O(s)$ be a tight upper bound on the length of the paths read by this DFA. The DFS search constrained by such DFA takes $O(n^s)$; this complexity is calculated using a similar analysis as the one of Theorem 5. Thus, the total complexity of an execution of `FINDREACHABLENODES` is $O(2^r + n^s)$. Given that this function is executed at most once in the procedure `PROCESSNODE()`, the complexity of this procedure is also $O(2^r + n^s)$.

Hence, the total complexity of GPQM is the number of times that the procedure `PROCESSNODE()` is executed multiplied by the cost of each single execution. Specifically, the total complexity is $O(n^{\lfloor \ell/2 \rfloor} \times \max\{2^r, n^s\})$ where r is the length of the longest regular expression in the query and s is the length of the longest matching path of any regular expression in such query.

It is, however, important to remark that this analysis gives an upper bound for the worst-case complexity. It assumes that, for every node p_i in the linearization, all the possible assignments will be evaluated. The average-case situations are often not that “bad” because (i) when a node p_i has already been assigned, only the assigned node is considered; and (ii) the node and edge predicates as well as reachability requirements often effectively prune the search space: for each possible assignment for a given p_i (where i is odd) only few assignments for p_{i+2} are possible. Furthermore, this analysis was made assuming that the attributed graph is complete, while many graphs in the real world are sparse. Moreover, the values of ℓ , r and s are often constant which makes our algorithm polynomial for many cases. Therefore, in practice, our algorithm shows much better performance than the given worst-case bound, as demonstrated in the experiments.

7.5. Experimental Evaluation

This section presents the implementation and experimental results of the proposed techniques. The main objective is to show the feasibility and efficiency of executing generalized pattern queries. Specifically, this section consists of four parts. We first describe the experimental setup that includes the information of the data set, the test set and the environment used (Section 7.5.1). Second, we introduce some examples of queries and the results of their evaluation on the complete DBLP graph by means of GPQM (Section 7.5.2). Third, we evaluate the performance of GPQM when the sizes of the query and the graph are varied (Section 7.5.3). Lastly, we illustrate the efficiency of using graph statistics (Section 7.5.4).

7.5.1. Experimental Setup

Datasets. We use the DBLP graph, which is a well-known computer science bibliography that captures information on authors, papers and where they are published (e.g., journals and conferences), as well as academic citations. We obtained the data from the DBLP website [52] on December 20, 2013 to produce a graph with more than 10 million elements (1.684.750 nodes and 9.955.181 edges) containing three node types and three edge types.

To study how the query response time varies with the graph size, we also generate synthetic graphs of varying sizes as subsets of the DBLP graph. To preserve the network structure among graph nodes, we perform the following steps to generate a graph with a specific number of nodes. We randomly select an arbitrary article, that has not been selected yet, and add the article node into the synthetic graph. Furthermore, we add all its edges and immediate neighbours (the 1-hop neighbourhood) if they do not belong to the generated graph. We repeat this step until the number of nodes reaches the target. We produce three synthetic DBLP graphs with $\{250K, 500K, 1M\}$ nodes and $\{923,762, 2,049,242, 4,510,296\}$ edges, respectively.

Query workload. There is no standard benchmark for generalized pattern queries. Therefore, we form queries that possess different features (such as reachability, pattern match queries and their combinations). As performance measures, we report the query execution time and the number of graph elements accessed during the processing.

Implementation. We implement a query processing system in C#. This system includes the query linearization and matching algorithms, as well as the query language parser and compiler. We use a server with 2,79GHz Xeon CPU and 24GB main memory (RAM) running the Windows Server operating system.

7.5.2. Queries on the Complete DBLP Graph

We introduce examples of generalized pattern queries with different characteristics. Then, we discuss experimental results of performing such queries on the DBLP graph.

Examples of Generalized Pattern Queries

We present some examples of generalized pattern queries in Figure 7-1. For instance, the generalized pattern query **Q1** evaluates reachability. This query makes use of the Kleene star to find paths of arbitrary length that represent the relationship between a specific author and his/her academic descendants¹. In particular, **Q1** allows to find all the academic descendants of Jiawei Han. On the other hand, the generalized pattern query **Q2** is a combination of reachability and pattern match queries. Specifically, the query allows to find the articles co-authored by two academic descendants of Jiawei Han.

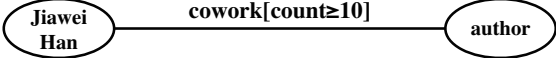
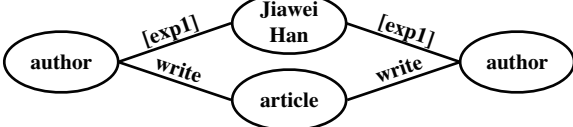
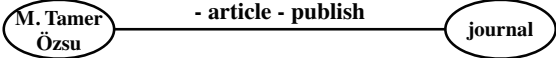
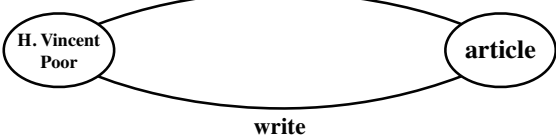
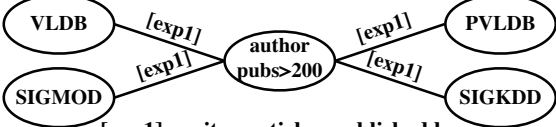

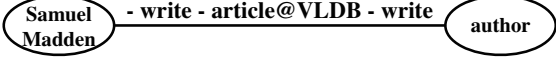
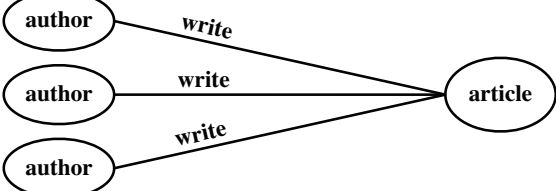
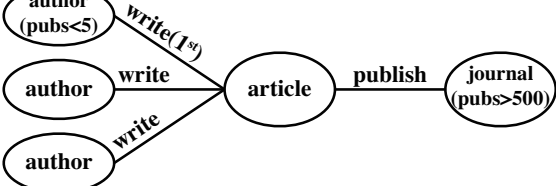
The generalized pattern queries **Q3** and **Q4** are examples of queries with the union operator. The former searches for journals that published papers written by M.Tamer Özsu or any of his co-authors; the latter looks for articles written by H. Vincent Poor and one of his colleagues, such that the colleague is either a prolific author² or has worked with the him for at least 10 times.

Notice that queries **Q5** and **Q6** look similar. However, they have different meaning and execution costs. In particular, **Q5** searches for prolific authors who have published articles at four conferences (i.e., VLDB, PVLDB, SIGMOD and SIGKDD). In contrast, **Q6** is a pattern match query where the articles at each conference also constitute nodes of the query. This makes a big difference since the output of a generalized pattern query is established by its nodes. Specifically, **Q5** reports only the authors that satisfy the condition, while **Q6** looks for all the articles published by the author in each conference and reports all the possible combinations. As a result, **Q6** can potentially have many

¹Academic descendant: An author A is an academic descendant of another author B , if A publishes at least 10 articles with B or A is an academic descendant of an academic descendant of B .

²Prolific author: An author is a prolific author when he/she has written more than 200 articles.

Table 7-1.: Examples of generalized pattern queries on the DBLP graph.

| Query | Query Representation |
|-------|---|
| Q1 | <p style="text-align: center;">(cowork[count≥10] - author -)* cowork[count≥10]</p>  |
| Q2 |  <p style="text-align: center;">[exp1]: (cowork[count≥10] - author -)* cowork[count≥10]</p> |
| Q3 | <p style="text-align: center;">((write) or (cowork - author - write)) - article - publish</p>  |
| Q4 | <p style="text-align: center;">(cowork[count≥10] - author - write) or (cowork - author[publication>200] - write)</p>  |
| Q5 |  <p style="text-align: center;">[exp1]: write - article - published by</p> |
| Q6 |  |
| Q7 | <p style="text-align: center;">(write - article@VLDB - write - author)^{≤k} - write - article@VLDB - write</p>  |
| Q8 |  |
| Q9 |  |

more matches than **Q5**, if the authors who satisfy the condition also have many combinations of papers from the four conferences. We discuss **Q7** and **Q8** in Section 7.5.3, and **Q9** in Section 7.5.4.

Performance of the Query Evaluation

Figure 7-6 contains three subfigures that show the experimental results of evaluating the generalized pattern queries of Table 7-1 on the complete DBLP graph. In particular, Figure 7-6a presents the execution time and Figure 7-6b shows the number of visited graph elements of each query. Similarly, Figure 7-6c shows the number of matches for each query. For example, we found 18 matches of **Q1** in 0,1 second visiting 6, 575 graph elements.

These results show that the number of accessed graph elements is strongly correlated to the execution time, which validates the assumption of using the number of visited elements as our complexity metric. In contrast, the number of matches does not necessarily reflect the processing time.

We show the difference on the performance between **Q5** and **Q6**. The execution time and the number of visited graph elements for **Q6** are almost 100 times greater than the ones for **Q5**. Also, **Q6** has 980 matches, while **Q5** has only one match. The reason is that, in the case of **Q5**, once we establish the reachability from an author to one of the conferences, we can stop exploring other paths that lead to the same conference via a different publication. In other words, we only need to find if the author has published in the conference. In contrast, **Q6** needs to enumerate all the publications from the author in the conference. That is why **Q6** is more expensive and returns more results. This verifies the hypothesis posed at the end of Section 7.5.2. Also, these results demonstrate that the expressive power of generalized pattern queries allows us to write the queries in a flexible way, where the trade-off between the execution time and the amount of information returned is established according to our needs.

In summary, we show that our query linearization and matching algorithms can evaluate generalized pattern queries.

7.5.3. Varying Graph and Query Sizes

We show that the processing time depends on the size of both the attributed multigraph and the generalized pattern query. We use two generalized pattern queries: **Q7** and **Q8** of Table 7-1.

Let us consider first query **Q7**. The expression $\rho^{\leq k}$ indicates that the expression ρ can be concatenated with itself from 0 to $k - 1$ times, i.e., $\rho^{\leq k} = \rho \cup \rho^2 \cup \dots \cup \rho^k$. Thus, **Q7** searches for the VLDB co-authors of Madden within a $(k + 1)$ -hop. Note that k sets an upper bound on the length of the paths that match the corresponding reachability requirement. We use this query on the experiments with varying k . Figure 7-7a shows the performance of our algorithm for this query **Q7** on various sizes of the DBLP graph. When the graph size is smaller than $1M$, or k is equal or smaller than

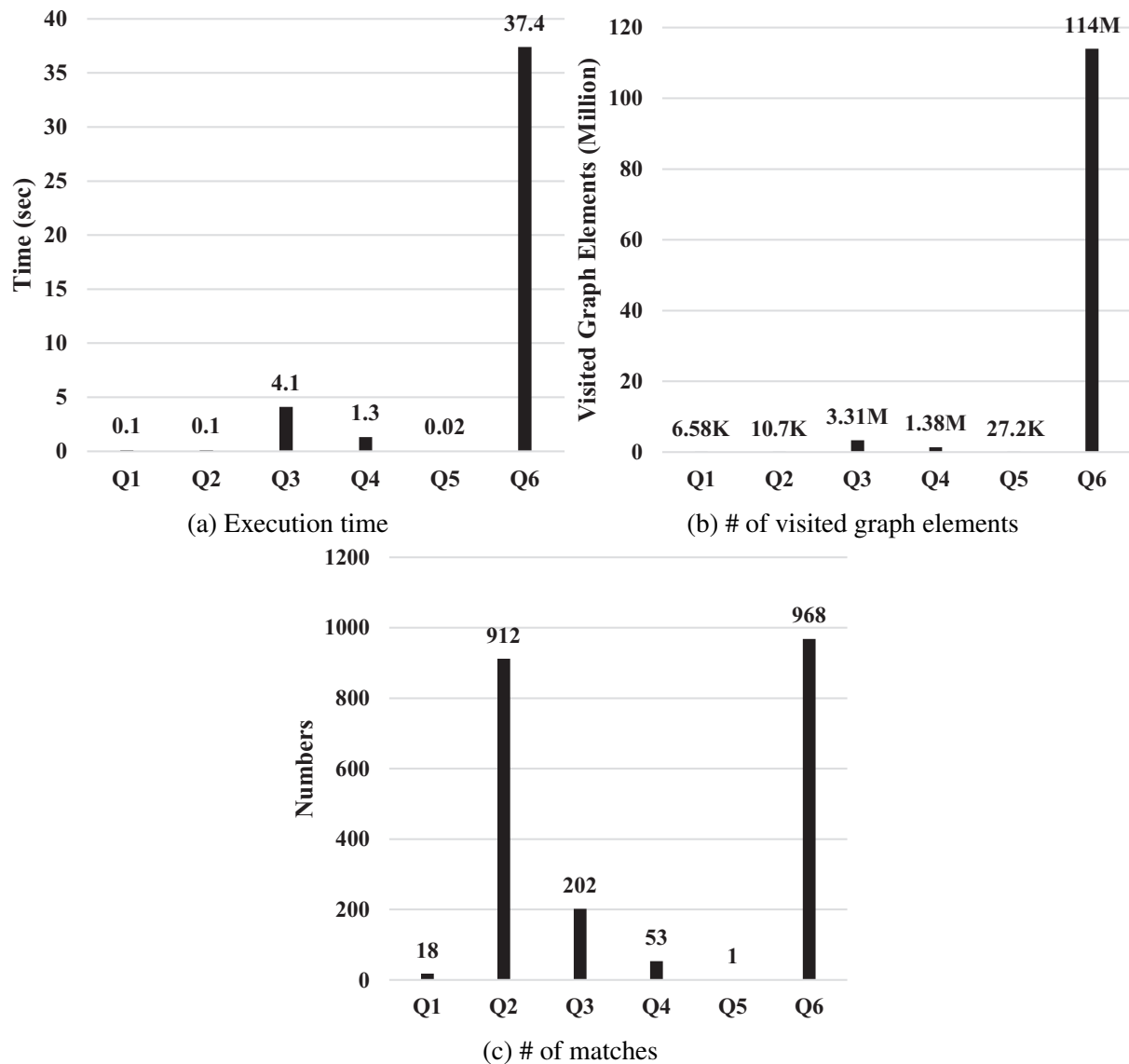


Figure 7-6.: Experimental results for queries **Q1–Q6** (see Table 7-1) on the complete DBLP graph.

3, the execution time of the query is less than 2 seconds. When we use the complete DBLP graph, however, the execution time grows with respect to the query length. As a result, while we find the 6-hop co-author neighbourhood for a specific author, via VLDB paper authorship, the query requires 50 seconds for the full DBLP graph. This shows that the execution time depends on the size of graph and the query complexity.

Next, we use a star pattern match query and change the number of nodes. Query **Q8** is a template for such type of query. This is a complex query with few predicates that prune the search space. It does not have various candidates of query linearization orders. We vary the number of nodes in the query by changing the number of author nodes. Figure 7-7b shows that the execution time is

strongly related to the graph size and the query size.

In conclusion, the time required for processing a generalized pattern query depends on both the size of the attributed multigraph as well as the query.

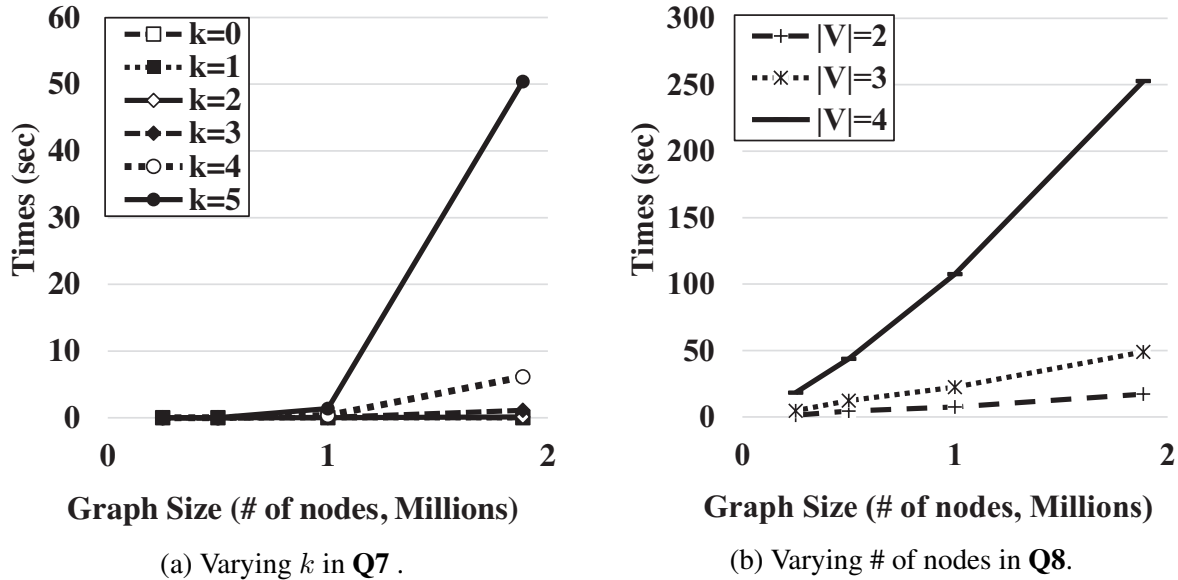


Figure 7-7.: Response time of GPQM for queries Q7–Q8 (see Table 7-1) on subgraphs of the DBLP graph of varying sizes.

7.5.4. Efficiency of E-GLA

The query linearization algorithm E-GLA uses graph statistics and selectivity estimation to generate a better linearization order. We compare E-GLA with GLA, an algorithm that optimizes the length of the linearizations but does not use the attributed multigraph statistics (see Section 3.3).

Notice that both GLA and E-GLA produce the same linearization for Q8. Thus, we use the query Q9 of Table 7-1 to evaluate the difference. This query searches for articles written by three authors and published at a journal with more than 500 publications. Furthermore, the first author must have at most 4 publications.

GLA starts from either an author or a journal node and then goes to the article node. Next, it selects an arbitrary non-visited node and repeatedly goes back to the center (i.e., the article node) until all the nodes are visited. In contrast, E-GLA starts from a journal node because it has the most restrictive predicate (journal with 500+ publications). Next, it visits the article node. Then, it selects the node with the lowest selectivity, i.e., the node that establishes that the author must have at most 4

publications. Finally, it goes back to the center and visits the non-visited nodes in an arbitrary order.

The performance of the matching algorithm GPQM is different depending on the used linearization algorithm. Figures 7-8a and 7-8b respectively show the execution time and the number of visited nodes, when each linearization algorithm is used, for different graph sizes. The E-GLA algorithm results in lower execution time and the difference between E-GLA and GLA increase with the the graph size.

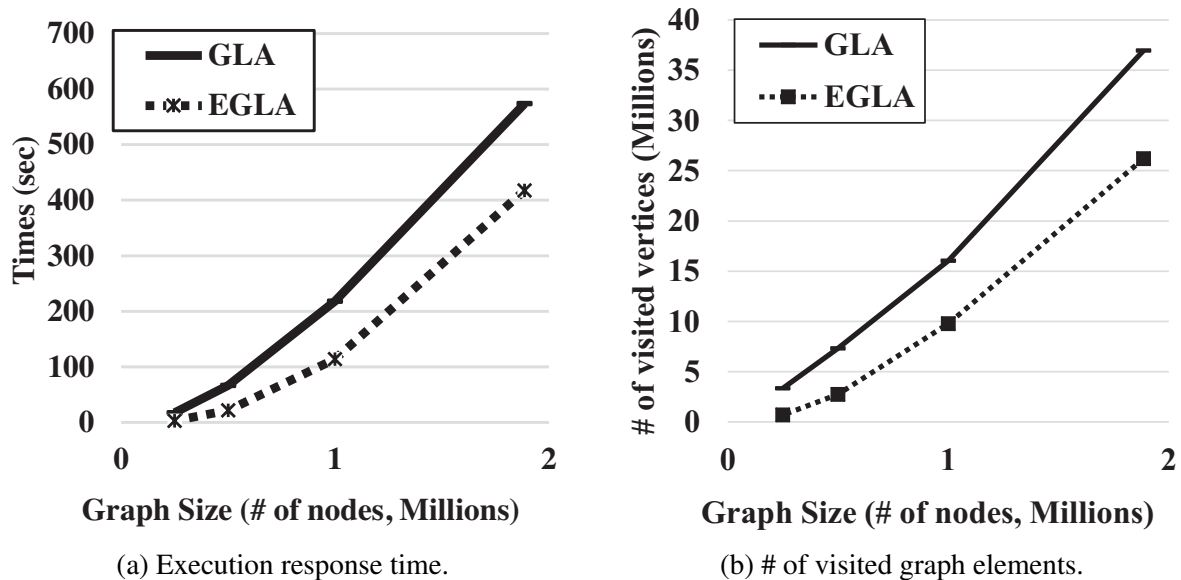


Figure 7-8.: Experimental results for query Q9 (see Table 7-1), using GLA and E-GLA linearizations, on subgraphs of the DBLP graph of varying sizes.

8. Conclusions

This thesis presents a novel approach to determine whether multigraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, where $n = |V_1| = |V_2|$ and $m = |E_1| = |E_2|$, are isomorphic. In particular, this approach is based on a string matching technique called parameterized matching. Parameterized matching is used to find strings that have the same structure, i.e., the relative distances among the occurrences of each symbol is preserved. Our solution starts by representing G_1 in a linear manner, which we call graph linearization $p = p_{1\dots\ell}$. Then, we search for the walks in G_2 that parameterized-match the linearization. If there exists at least one of such walks, we conclude that the graphs are isomorphic. The correctness of our approach is formally proven.

We develop a Graph Linearization Algorithm called GLA. This algorithm does a DFS-like traversal on G_1 guided by heuristics that consider the number of unexplored adjacent edges that nodes have. The GLA algorithm produces short linearizations as illustrated through empirical examples. In fact, it is proven that the produced linearizations are 2-approximate length-optimal. We show that the time complexity of GLA is $O(m + nd \lg d)$ where d is the maximum degree of the nodes in the graph.

New optimizations can be included in the GLA algorithm to incur in lower time during the matching phase. In particular, the matching time does not only depend on the length of the linearization, but also on the order of comparisons. For instance, the topological graph statistics of the multigraphs can be used to produce a linearization that prunes the search space during the matching phase. For example, if the frequency of some nodes of a certain degree is low, it would be appropriate to start the linearization from such nodes. However, for clarity, in this thesis, we focus on the fundamental approach only.

Furthermore, we devise a matching algorithm called PMG that searches for walks in G_2 that parameterized-matches the linearization $p = p_{1\dots\ell}$. Specifically, this algorithm does a DFS traversal on G_2 where all the feasible mappings from the graph elements in the linearization to the graph elements in G_2 are explored. One of the key ideas of the algorithm is to prune the search space by considering node degrees and previous assignments. The time complexity of PMG is $O(nd^{\lceil \ell/2 \rceil})$.

We experimentally evaluate the efficiency of our solution by comparing with a prominent graph isomorphism algorithm called VF2. Experiments on synthetic graphs show that our algorithm performs better for both sparse graphs and complete graphs, but the difference is more significant for

complete graphs. We also perform experiments on benchmark graphs. For those, our algorithm reported better time results than VF2 in about half of the datasets. More precisely, VF2 excels in regular graphs, while our algorithm is significantly faster in 65 % of Miyazaki-based constructed graphs. This is an interesting result since Miyazaki-constructed graphs constitute one of the hardest cases for graph isomorphism algorithms [141]. It is important to remark that in some cases where our solution is not short running, VF2 is fast. However, in the majority of the short-running cases, namely 66 % of such cases, our algorithm runs faster. This opens up a possibility of a hybrid algorithm that selects between these two algorithms, either statistically-based on the graph topology or dynamically after running for some time, which we leave as future work.

We present a straightforward adaptation of our approach to determine whether $G_1 = (V_1, E_1)$ is isomorphic to a subgraph in $G_2 = (V_2, E_2)$. The resulting algorithm, called PMG-SI, preserves the time complexity of PMG, i.e., $O(nd^{\lceil \ell/2 \rceil})$ where $n = |V_2|$ and d is the maximum node degree in G_2 . We experimentally evaluate the algorithm on synthetic graphs G_2 of varying sizes. For the graphs G_1 we used path, star, cyclic and complete graphs. Experimental results verify that the matching time depends on the linearization length and the promptness in which mismatches are detected.

Moreover, we extend our approach to query attributed multigraphs. In fact, we define a new type of queries called generalized pattern queries that establish predicates, reachability and topological requirements. These queries are multigraphs that establish predicates on a set of nodes of interest (through node predicates) and the reachability requirements among them through highly-expressive regular expressions associated to the edges. Such expressions are composed by nodes and edges (associated to predicates) and the regular expression operators (i.e., concatenation, union and Kleene star). Thus, each edge $e = (u, v)$ in the query defines a complex reachability requirement from node u to node v . It is important to remark that previous reachability query models do not support evaluation of predicates on intermediate nodes and edges nor the Kleene star operator. This new type of queries can represent pattern match queries, reachability queries and beyond.

Then, we use the linearization approach to solve the problem of finding the matches of a generalized pattern query $G_P = (V_P, E_P)$ in an attributed multigraph $G = (V, E, f_V, f_E)$. Each vertex and each edge in G is associated to a set of attributes, which are defined by the functions f_V and f_E , respectively. In order to produce a convenient linearization, we propose an algorithm, called E-GLA, that takes into account the statistics of the attributed multigraph to linearize the generalized pattern query. In particular, for each node $u \in V_P$, the selectivity of u is calculated as the probability of selecting a node from G that satisfies the predicates of u . Then, the main heuristic of E-GLA is starting (or continuing) the query linearization from the node with lowest selectivity. In this sense, the mismatches are early detected during the matching phase and, hence, the search space is pruned. Furthermore, E-GLA query linearizations are also 2-approximate length-optimal. The time complexity of the algorithm is the same as the one of GLA.

The matching algorithm, called GPQM, does a DFS traversal on the attributed multigraph. The different feasible mappings between the nodes in the query linearization and the nodes in the DFS search tree are considered. In order to evaluate reachability, we construct a deterministic finite automaton (DFA) that accepts all the paths that satisfy the corresponding reachability expression. Then, we perform another DFS constrained by such automaton. The time complexity of GPQM is $O(nd^{\lceil \ell/2 \rceil})$, where $n = |V|$, d is the maximum node degree in G and ℓ is the length of the query linearization.

We experimentally validate the algorithm on the DBLP graph. We formed generalized pattern queries that possess different features, including reachability, pattern match queries and their combinations. Our experiments verify that the number of visited nodes is strongly correlated with the execution time. Furthermore, the results illustrate that the expressive power of generalized pattern queries allows us to write the queries in a flexible way, where the trade-off between the execution time and the amount of information returned is established according to our needs. Other set of experiments demonstrate that the processing time of a generalized pattern query depends on both the size of the query and the attributed multigraph. In order to evaluate the effectiveness of E-GLA heuristics, we compare the performance of GPQM when either E-GLA or GLA linearization algorithm is used. The results indicate that, when E-GLA is used, the execution time is lower and the difference increase with the graph size. As a conclusion, generalized graph queries are processed efficiently by considering graph statistics and selectivity estimation for query linearization.

Bibliography

- [1] Rakesh Agrawal, Alexander Borgida, and HV Jagadish. Efficient management of transitive relationships. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*. Citeseer, 1989.
- [2] Alfred V Aho and John E Hopcroft. *Design & Analysis of Computer Algorithms*. Pearson Education India, 1974.
- [3] Réka Albert, Hawoong Jeong, and Albert-László Barabási. Internet: Diameter of the worldwide web. *Nature*, 401(6749):130–131, 1999.
- [4] Amihod Amir, Yonatan Aumann, Richard Cole, Moshe Lewenstein, and Ely Porat. Function matching: Algorithms, applications, and a lower bound. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming*, 2003.
- [5] Amihod Amir, Martin Farach, and S Muthukrishnan. Alphabet dependence in parameterized matching. *Information Processing Letters*, 49(3):111–115, 1994.
- [6] Amihod Amir and Igor Nor. Generalized function matching. *Journal of Discrete Algorithms*, 5(3):514–523, 2007.
- [7] Alberto Apostolico, Péter L Erdős, and Moshe Lewenstein. Parameterized matching with mismatches. *Journal of Discrete Algorithms*, 5(1):135–140, 2007.
- [8] Alberto Apostolico and Zvi Galil. *Pattern matching algorithms*. Oxford University Press, USA, 1997.
- [9] Alberto Apostolico and Raffaele Giancarlo. Periodicity and repetitions in parameterized strings. *Discrete Applied Mathematics*, 156(9):1389–1398, 2008.
- [10] Gustavo O Arocena and Alberto O Mendelzon. Weboql: Restructuring documents, databases and webs. In *Data Engineering (ICDE), 1998 IEEE 14th International Conference on*, pages 24–33. IEEE, 1998.
- [11] G Phanendra Babu, Babu M Mehtre, and Mohan S Kankanhalli. Color indexing for efficient image retrieval. *Multimedia Tools and Applications*, 1(4):327–348, 1995.
- [12] Ricardo Baeza-Yates and Gaston H Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):82, 1992.

-
- [13] Brenda S Baker. A program for identifying duplicated code. In *Computing Science and Statistics: Proceedings of the 24th Symposium on the Interface*, 1992.
- [14] Brenda S Baker. On finding duplication in strings and software. Technical report, AT&T Laboratories, 1993.
- [15] Brenda S Baker. A theory of parameterized pattern matching: Algorithms and applications. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, 1993.
- [16] Brenda S Baker. Parameterized pattern matching by boyer-moore-type algorithms. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, page 550. Society for Industrial and Applied Mathematics, 1995.
- [17] Brenda S Baker. Parameterized pattern matching: Algorithms and applications. *Journal of Computer and System Sciences*, 52(1):28–42, 1996.
- [18] Brenda S Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal on Computing*, 26(5):1343–1362, 1997.
- [19] Brenda S Baker. Parameterized diff. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 854–855. Society for Industrial and Applied Mathematics Philadelphia, PA, USA, 1999.
- [20] Amotz Bar-Noy and Ilan Kessler. Tracking mobile users in wireless communications networks. *Information Theory, IEEE Transactions on*, 39(6):1877–1886, 1993.
- [21] David Becerra, Juan Mendivelso, and Yoan Pinzón. An algorithm for the weighted longest common subsequence problem. In *Proceedings of the 5th Colombian Computing Conference (5CCC)*, 2010.
- [22] David Becerra, Juan Mendivelso, and Yoan Pinzón. A multiobjective optimization algorithm for the weighted lcs. *Accepted in Discrete Applied Mathematics*, 2015.
- [23] David Becerra, Juan Mendivelso, and Yoan J Pinzón. A multiobjective approach to the weighted longest common subsequence problem. In *Proceedings of the Prague Stringology Conference 2012 (PSC 2012)*, pages 64–74, 2012.
- [24] Richard E Blake. Partitioning graph matching with constraints. *Pattern Recognition*, 27(3):439–446, 1994.
- [25] Anselm Blumer, Janet Blumer, David Haussler, Andrzej Ehrenfeucht, Mu-Tian Chen, and Joel Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40(1):31–55, 1985.
- [26] Robert S Boyer and J Strother Moore. A fast string searching algorithm. *Communications ACM*, 20(10):762–772, 1977.

- [27] Carl Branden and John Tooze. *Introduction to protein structure*, volume 2. Garland New York, 1991.
- [28] Donald E Brown, Christopher L Huntley, and Andrew R Spillane. A parallel genetic heuristic for the quadratic assignment problem. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 406–415. Morgan Kaufmann Publishers Inc., 1989.
- [29] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: Optimal xml pattern matching. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 310–321. ACM, 2002.
- [30] Emilios Cambouropoulos, Maxime Crochemore, Costas Iliopoulos, Laurent Mouchard, and Yoan Pinzon. Algorithms for computing approximate repetitions in musical sequences. *International Journal of Computer Mathematics*, 79(11):1135–1148, 2002.
- [31] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, 2004.
- [32] Edward PF Chan and Heechul Lim. Optimization and evaluation of shortest path queries. *The VLDB Journal*, 16(3):343–369, 2007.
- [33] William I Chang and Eugene L Lawler. Approximate string matching in sublinear expected time. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, 1990.
- [34] William I. Chang and Eugene L. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4):327–344, 1994.
- [35] Li Chen, Amarnath Gupta, and M Erdem Kurul. Stack-based algorithms for pattern matching on dags. In *Proceedings of the VLDB Endowment*, volume 4, pages 493–504, 2005.
- [36] Yangjun Chen and Yibin Chen. An efficient algorithm for answering graph reachability queries. In *Data Engineering (ICDE), 2008 IEEE 24th International Conference on*, pages 893–902. IEEE, 2008.
- [37] Jiefeng Cheng and Jeffrey Xu Yu. On-line exact shortest distance query processing. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 481–492. ACM, 2009.
- [38] Jiefeng Cheng, Jeffrey Xu Yu, Bolin Ding, Philip S Yu, and Haixun Wang. Fast graph pattern matching. In *Data Engineering (ICDE), 2008 IEEE 24th International Conference on*, pages 913–922. IEEE, 2008.

- [39] William J Christmas, Josef Kittler, and Maria Petrou. Structural matching in computer vision using probabilistic relaxation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(8):749–764, 1995.
- [40] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.
- [41] Richard Cole and Ramesh Hariharan. Faster suffix tree construction with missing suffix links. *SIAM Journal on Computing*, 33(1):26–42, 2004.
- [42] Mariano P Consens and Alberto O Mendelzon. Graphlog: A visual formalism for real life recursion. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1990.
- [43] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty years of graph matching in pattern recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 18(3):265–298, 2004.
- [44] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.
- [45] Luigi P Cordella and Mario Vento. Symbol recognition in documents: a collection of techniques? *International Journal on Document Analysis and Recognition*, 3(2):73–88, 2000.
- [46] Juana Córdoba, Juan C Mendivelso, and Luis F Niño. Búsqueda de secuencias microsatelitales en fríjol común (*phaseolus vulgaris* l.). In *Proceedings of the 3rd Colombian Computing Conference (3CCC)*, 2008.
- [47] Derek Gordon Corneil and Calvin C Gotlieb. An efficient algorithm for graph isomorphism. *Journal of the ACM (JACM)*, 17(1):51–64, 1970.
- [48] M. Crochemore, C.S. Iliopoulos, G. Navarro, Y.J. Pinzon, and A. Salinger. Bit-parallel (δ , γ)-Matching and Suffix Automata. *Journal of Discrete Algorithms*, 3(2-4):198–214, 2005.
- [49] Maxime Crochemore, Artur Czumaj, Leszek Gasieniec, Stefan Jarominek, Thierry Lecroq, Wojciech Plandowski, and Wojciech Rytter. Speeding up two string-matching algorithms. *Algorithmica*, 12(4):247–267, 1994.
- [50] Maxime Crochemore, Costas S Iliopoulos, Thierry Lecroq, Yoan J Pinzon, Wojciech Plandowski, and Wojciech Rytter. Occurrence and substring heuristics for δ -matching. *Fundamenta Informaticae*, 56(1):1–21, 2003.
- [51] Isabel F Cruz, Alberto O Mendelzon, and Peter T Wood. A graphical query language supporting recursion. In *ACM SIGMOD Record*, 1987.

- [52] DBLP. The dblp computer science bibliography. <http://dblp.uni-trier.de/xml/>.
- [53] Cédric Du Mouza, Philippe Rigaux, and Michels Scholl. Parameterized pattern queries. *Data & Knowledge Engineering*, 63(2):433–456, 2007.
- [54] Jack Edmonds and Ellis L Johnson. Matching, euler tours and the chinese postman. *Mathematical Programming*, 5(1):88–124, 1973.
- [55] Frank Eichinger, Klemens Böhm, and Matthias Huber. Mining edge-weighted call graphs to localise software bugs. In *Proceedings of the 2008 European Conference on Machine Learning and Knowledge Discovery in Databases-Part I*, pages 333–348. Springer-Verlag, 2008.
- [56] Brian Falkenhainer, Kenneth D Forbus, and Dedre Gentner. The structure-mapping engine: Algorithm and examples. *Artificial Intelligence*, 41(1):1–63, 1989.
- [57] Kuo-Chin Fan, Cheng-Wen Liu, and Yuan-Kai Wang. A fuzzy bipartite weighted graph matching approach to fingerprint verification. In *Systems, Man, and Cybernetics, 1998. 1998 IEEE International Conference on*, volume 5, pages 4363–4368. IEEE, 1998.
- [58] Wenfei Fan and Philip Bohannon. Information preserving xml schema embedding. *ACM Transactions on Database Systems (TODS)*, 33(1):4, 2008.
- [59] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Yinghui Wu. Adding regular expressions to graph reachability and pattern queries. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 39–50. IEEE, 2011.
- [60] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. Graph pattern matching: From intractable to polynomial time. *Proceedings of the VLDB Endowment*, 3(1-2):264–275, 2010.
- [61] Wenfei Fan, Jianzhong Li, Shuai Ma, Hongzhi Wang, and Yinghui Wu. Graph homomorphism revisited for graph matching. *Proceedings of the VLDB Endowment*, 3(1-2):1161–1172, 2010.
- [62] Alexander Filatov, Alexander Gitis, and Igor Kil. Graph-based handwritten digit string recognition. In *Proceedings of the 3rd International Conference on Document Analysis and Recognition*, volume 2, pages 845–848. IEEE, 1995.
- [63] Nathan J Fine and Herbert S Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16:109–114, 1965.
- [64] Stefan Fischer, Kaspar Gilomen, and Horst Bunke. Identification of diatoms by grid graph matching. *Structural, Syntactic, and Statistical Pattern Recognition*, 2396:335–370, 2002.

- [65] Patrick J Flynn and Anil K Jain. Cad-based computer vision: from cad models to relational graphs. In *Systems, Man and Cybernetics, IEEE International Conference on*, pages 162–167. IEEE, 1989.
- [66] Pasquale Foggia, Roberto Genna, and Mario Vento. Symbolic vs. connectionist learning: an experimental comparison in a structured domain. *Knowledge and Data Engineering, IEEE Transactions on*, 13(2):176–195, 2001.
- [67] Denis Fourches, Eugene Muratov, and Alexander Tropsha. Trust, but verify: on the importance of chemical structure curation in cheminformatics and qsar modeling research. *Journal of Chemical Information and Modeling*, 50(7):1189–1204, 2010.
- [68] Kimmo Fredriksson and Maxim Mozgovoy. Efficient parameterized string matching. *Information Processing Letters*, 100(3):91–96, 2006.
- [69] Brian Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. In *AAAI FS*, 2006.
- [70] Michael R Garey and David S Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. WH Freeman & Co., 1979.
- [71] Cristina Gomila and Fernand Meyer. Tracking objects by graph matching of image partition sequences. In *Proceedings of the 3rd IAPR-TC15 Workshop on Graph-Based Representations in Pattern Recognition*, pages 1–11, 2001.
- [72] Alla Goralčíková and Václav Koubek. A reduct-and-closure algorithm for graphs. In *Mathematical Foundations of Computer Science 1979*, pages 301–307. Springer, 1979.
- [73] Ralf Hartmut Güting. Graphdb: Modeling and querying graphs in databases. In *Proceedings of 20th International Conference on Very Large Data Bases (VLDB)*, volume 94, pages 12–15, 1994.
- [74] Robert M Haralick and Gordon L Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980.
- [75] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13:338, 1984.
- [76] Carmit Hazay. Parameterized matching. Master’s thesis, Bar-Ilan University, 2004.
- [77] Carmit Hazay, Moshe Lewenstein, and Dina Sokol. Approximate parameterized matching. *ACM Transactions on Algorithms (TALG)*, 3(3):29, 2007.
- [78] Huahai He and Ambuj K Singh. Graphs-at-a-time: Query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 405–418. ACM, 2008.

- [79] Laurent Héroult, Radu Horaud, Françoise Veillon, and Jean-Jacques Niez. Symbolic image matching by simulated annealing. In *Proceedings of the 4th British Machine Vision Conference (BMVC'90)*, 1990.
- [80] Adel Hlaoui and Shengrui Wang. A new algorithm for graph matching with application to content-based image retrieval. *Structural, Syntactic, and Statistical Pattern Recognition*, 2396:291–300, 2002.
- [81] Ramana M Idury and Alejandro A Schäffer. Multiple matching of parameterized patterns. *Theoretical Computer Science*, 154(2):203–224, 1996.
- [82] Costas S Iliopoulos, Marcin Kubica, M Sohel Rahman, and Tomasz Waleń. Algorithms for computing the longest parameterized common subsequence. In *Combinatorial Pattern Matching*, pages 265–273. Springer, 2007.
- [83] Ruoming Jin, Hui Hong, Haixun Wang, Ning Ruan, and Yang Xiang. Computing label-constraint reachability in graph databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 123–134. ACM, 2010.
- [84] Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 3-hop: A high-compression indexing scheme for reachability query. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 813–826. ACM, 2009.
- [85] Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. Efficiently answering reachability queries on very large directed graphs. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 595–608. ACM, 2008.
- [86] Sanjay Joshi and Tien-Chien Chang. Graph-based heuristics for recognition of machined features from a 3d solid model. *Computer-Aided Design*, 20(2):58–66, 1988.
- [87] Orgad Keller, Tsvi Kopelowitz, and Moshe Lewenstein. On the longest common parameterized subsequence. *Theoretical Computer Science*, 410(51):5347–5353, 2009.
- [88] Whoi-Yul Kim and Avinash C Kak. 3-d object recognition using bipartite matching embedded in discrete relaxation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13:224–251, 1991.
- [89] Donald E Knuth, James H Morris Jr, and Vaughan R Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323, 1977.
- [90] S Rao Kosaraju. Faster algorithms for the construction of parameterized suffix trees. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Washington, DC, USA, 1995.

- [91] Lucas Kovar, Michael Gleicher, and Frédéric Pighin. Motion graphs. *ACM Transactions on Graphics (TOG)*, 21(3):473–482, 2002.
- [92] Christopher Lee, Catherine Grasso, and Mark F Sharlow. Multiple sequence alignment using partial order graphs. *Bioinformatics*, 18(3):452–464, 2002.
- [93] Inbok Lee, Juan Mendivelso, and Yoan J Pinzón. $\delta\gamma$ -parameterized matching. *Lecture Notes in Computer Science, String Processing and Information Retrieval*, 5280:236–248, 2008.
- [94] Josep Lladós, Enric Martí, and Juan J Villanueva. Symbol recognition by error-tolerant subgraph matching between region adjacency graphs. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 23(10):1137–1143, 2001.
- [95] Eugene M Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of Computer and System Sciences*, 25(1):42–65, 1982.
- [96] Bin Luo and Edwin R Hancock. Structural graph matching using the em algorithm and singular value decomposition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 23(10):1120–1136, 2001.
- [97] Roger C Lyndon and Marcel-Paul Schützenberger. The equation $a^m = b^n c^p$ in a free group. *The Michigan Mathematical Journal*, 11:289–298, 1962.
- [98] William A Mackaness and Kate M Beard. Use of graph theory to support map generalization. *Cartography and Geographic Information Systems*, 20(4):210–221, 1993.
- [99] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22:935, 1993.
- [100] Christopher J. Matheus, Philip K. Chan, and Gregory Piatetsky-Shapiro. Systems for knowledge discovery in databases. *Knowledge and Data Engineering, IEEE Transactions on*, 5(6):903–913, 1993.
- [101] Edward M McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272, 1976.
- [102] Brendan D McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45, 1981.
- [103] Juan Mendivelso. Definition and solution of a new string searching variant termed $\delta\gamma$ -parameterized matching. Master’s thesis, Universidad Nacional de Colombia, 2010.
- [104] Juan Mendivelso. The graph pattern matching problem through parameterized matching. phd proposal. In *Proceedings of the 8th Colombian Computing Conference (8CCC)*, 2013.

- [105] Juan Mendivelso, Sunghwan Kim, Sameh Elnikety, Yuxiong He, Seung-won Hwang, and Yoan Pinzón. Solving graph isomorphism using parameterized matching. *Lecture Notes in Computer Science, String Processing and Information Retrieval*, 8214:230–242, 2013.
- [106] Juan Mendivelso, Inbok Lee, and Yoan J Pinzón. Approximate function matching under δ -and γ -distances. *Lecture Notes in Computer Science, String Processing and Information Retrieval*, 7608:348–359, 2012.
- [107] Juan Mendivelso, Camilo Pino, Luis F Niño, and Yoan Pinzón. Finding regularities in biological sequences through $\delta\gamma$ -approximate abelian periods. In *Proceedings of the 11th International Meeting on Computational Intelligence Methods for Bioinformatics and Biostatistics (CIBB 2014)*, 2014.
- [108] Juan Mendivelso, Camilo Pino, Luis F Niño, and Yoan Pinzón. Approximate abelian periods to find motifs in biological sequences. *Accepted in Lecture Notes in Bioinformatics (LNBI)*, 2015.
- [109] Juan Mendivelso and Yoan Pinzón. Revisión de diferentes tipos de búsqueda de patrones en cadenas con énfasis en la parametrizada y en la (δ, γ, α) . In *Proceedings of the National Symposium on Research and Development 2010 (ENID 2010)*, 2010.
- [110] Juan Mendivelso and Yoan Pinzón. A new approach to isomorphism in attributed graphs. In *Proceedings of the 9th Colombian Computing Conference (9CCC)*, 2014.
- [111] Juan Mendivelso and Yoan Pinzón. A novel approach to approximate parikh matching for comparing composition in biological sequences. In *Proceedings of the 6th International Conference on Bioinformatics and Computational Biology (BICoB 2014)*, 2014.
- [112] Juan Mendivelso, Yoan Pinzón, and Inbok Lee. Finding overlaps within regular expressions with variable-length gaps. In *Proceedings of the ACM Research in Adaptive and Convergent Systems Conference 2013 (ACM RACS 2013)*, 2013.
- [113] Bruno T Messmer. *Efficient graph matching algorithms for preprocessed model graphs*. PhD thesis, Institute of Computer Science and Applied Mathematics, University of Bern, 1995.
- [114] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, pages 29–42. ACM, 2007.
- [115] Sung H Myaeng and Aurelio López-López. Conceptual graph matching: A flexible algorithm and experiments. *Journal of Experimental & Theoretical Artificial Intelligence*, 4(2):107–126, 1992.

- [116] Eugene W Myers. An o (nd) difference algorithm and its variations. *Algorithmica*, 1(1):251–266, 1986.
- [117] Nils J Nilsson. *Principles of Artificial Intelligence*. Springer-Verlag, 1982.
- [118] Nils J Nilsson. *Artificial intelligence: A new synthesis*. Elsevier, 1998.
- [119] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. *Stanford Digital Libraries Working Paper*, 1999.
- [120] Yun Peng, Byron Choi, and Jianliang Xu. Selectivity estimation of twig queries on cyclic graphs. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 960–971. IEEE, 2011.
- [121] Euripides GM Petrakis and Christos Faloutsos. Similarity searching in medical image databases. *Knowledge and Data Engineering, IEEE Transactions on*, 9(3):435–447, 1997.
- [122] Sherif Sakr. Graphrel: A decomposition-based and selectivity-aware relational framework for processing sub-graph queries. In *Database Systems for Advanced Applications*, pages 123–137. Springer, 2009.
- [123] Sherif Sakr, Sameh Elnikety, and Yuxiong He. G-sparql: A hybrid engine for querying large attributed graphs. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, pages 335–344. ACM, 2012.
- [124] Leena Salmela and Jorma Tarhio. Sublinear algorithms for parameterized matching. In *Combinatorial Pattern Matching*, pages 354–364. Springer, 2006.
- [125] M Salotti and N Laachfoubi. Topographic graph matching for shift estimation. In *Proceedings of the 3rd IAPR-TC15 Workshop Graph-Based Representations in Pattern Recognition*, pages 54–63, 2001.
- [126] Alberto Sanfeliu and King-Sun Fu. A distance measure between attributed relational graphs for pattern recognition. *Systems, Man and Cybernetics, IEEE Transactions on*, SMC-13(3):353–362, 1983.
- [127] Mohamed Sarwat, Sameh Elnikety, Yuxiong He, and Gabriel Kliot. Horton: Online query execution engine for large distributed graphs. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 1289–1292. IEEE, 2012.
- [128] Mohamed Sarwat, Sameh Elnikety, Yuxiong He, and Mohamed F Mokbel. Horton+: A distributed system for processing declarative reachability queries over partitioned graphs. *Proceedings of the VLDB Endowment*, 6(14):1918–1929, 2013.
- [129] Robert W Scheifler and Jim Gettys. The x window system. *ACM Transactions on Graphics (TOG)*, 5(2):79–109, 1986.

- [130] Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing*, 17:1253, 1988.
- [131] Nigel Shadbolt, Wendy Hall, and Tim Berners-Lee. The semantic web revisited. *Intelligent Systems, IEEE*, 21(3):96–101, 2006.
- [132] Dennis Shasha, Jason TL Wang, and Rosalba Giugno. Algorithmics and applications of tree and graph searching. In *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 39–52. ACM, 2002.
- [133] Kim Shearer, Horst Bunke, and Svetha Venkatesh. Video indexing and similarity retrieval by largest common subgraph detection using decision trees. *Pattern Recognition*, 34(5):1075–1091, 2001.
- [134] Lei Sheng, Z Meral Ozsoyoglu, and Gultekin Ozsoyoglu. A graph query language and its query processing. In *Data Engineering (ICDE), 1999 IEEE 15th International Conference on*, pages 572–581. IEEE, 1999.
- [135] Ali Shokoufandeh and Sven Dickinson. A unified framework for indexing and matching hierarchical shape structures. *Visual Form 2001*, 2059:67–84, 2001.
- [136] Klaus Simon. An improved algorithm for transitive closure on acyclic digraphs. *Theoretical Computer Science*, 58(1):325–346, 1988.
- [137] Daniel D Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.
- [138] Frits Steenhof, Harry Duque, Björn Nilsson, Kees Goossens, and Rafael Peset Llopis. Networks on chips for high-end consumer-electronics tv system architectures. In *Design, Automation and Test in Europe, DATE'06. Proceedings*, volume 2, pages 1–6. IEEE, 2006.
- [139] M.J. Swain and D.H. Ballard. Color indexing. *International Journal for Parasitologyournal of Computer Vision*, 7(1):11–32, 1991.
- [140] Anastasios Tefas, Constantine Kotropoulos, and Ioannis Pitasi. Using support vector machines to enhance the performance of elastic graph matching for frontal face authentication. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 23(7):735–746, 2001.
- [141] Greg Daniel Tener. *Attacks on difficult instances of graph isomorphism: sequential and parallel algorithms*. PhD thesis, University of Central Florida, 2009.
- [142] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

- [143] Hanghang Tong, Christos Faloutsos, Brian Gallagher, and Tina Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 737–746. ACM, 2007.
- [144] Andrea Torsello and Edwin R Hancock. Learning structural variations in shock trees. *Structural, Syntactic, and Statistical Pattern Recognition*, 2396:101–117, 2002.
- [145] Silke Trißl and Ulf Leser. Fast and practical indexing and querying of very large graphs. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 845–856. ACM, 2007.
- [146] Wen-Hsiang Tsai and King-Sun Fu. Error-correcting isomorphisms of attributed relational graphs for pattern analysis. *Systems, Man and Cybernetics, IEEE Transactions on*, 9(12):757–768, 1979.
- [147] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
- [148] Julian R Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.
- [149] Jacques Van Helden, Avi Naim, Renato Mancuso, Matthew Eldridge, Lorenz Wernisch, David Gilbert, and Shoshana J Wodak. Representing and analysing molecular and cellular function using the computer. *Biological chemistry*, 381(9/10):921–936, 2000.
- [150] Michael A Van Wyk, Tariq S Durrani, and Barend J Van Wyk. A rkhs interpolator-based graph matching algorithm. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(7):988–995, 2002.
- [151] Haixun Wang, Hao He, Jun Yang, Philip S Yu, and Jeffrey Xu Yu. Dual labeling: Answering graph reachability queries in constant time. In *Data Engineering (ICDE), 2006 IEEE 22nd International Conference on*, pages 75–75. IEEE, 2006.
- [152] Mei Wang, Yoshio Iwai, and Masahiko Yachida. Expression recognition from time-sequential facial images by use of expression change model. In *Automatic Face and Gesture Recognition, 1998. Proceedings. 3rd IEEE International Conference on*, pages 324–329. IEEE, 1998.
- [153] Takashi Washio and Hiroshi Motoda. State of the art of graph-based data mining. *ACM SIGKDD Explorations Newsletter*, 5(1):59–68, 2003.
- [154] Fang Wei. Tedi: Efficient shortest path query answering on graphs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 99–110. ACM, 2010.

-
- [155] Peter Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11, 1973.
- [156] Xifeng Yan, Philip S Yu, and Jiawei Han. Graph indexing: A frequent structure-based approach. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 335–346. ACM, 2004.
- [157] Jeffrey Xu Yu and Jiefeng Cheng. Graph reachability queries: A survey. In *Managing and Mining Graph Data*, pages 181–215. Springer, 2010.
- [158] Peixiang Zhao, Charu C Aggarwal, and Min Wang. gsketch: On query estimation in graph streams. *Proceedings of the VLDB Endowment*, 5(3):193–204, 2011.
- [159] Peixiang Zhao and Jiawei Han. On graph query optimization in large networks. *Proceedings of the VLDB Endowment*, 3(1):340–351, 2010.
- [160] Lei Zou, Lei Chen, and M Tamer Özsu. Distance-join: Pattern match query in a large graph database. *Proceedings of the VLDB Endowment*, 2(1):886–897, 2009.
- [161] Lei Zou, Lei Chen, M Tamer Özsu, and Dongyan Zhao. Answering pattern match queries in large graph databases via graph embedding. *The VLDB Journal*, 21:97–120, 2012.