Università di Catania

Dipartimento di Fisica e Astronomia "Ettore Majorana"

PhD programme in complex systems

Roberto Grasso

Advanced Algorithms for Subgraph Isomorphism, Motif Discovery, and Graph Embedding in Complex Networks

PhD thesis

Supervisor:
Chiar.mo Prof. A. Pulvirenti

Co-Supervisor:
Prof. G. Micale

academic year 2023/2024

Contents

1 Introduction **1**

1.1 Context and Motivation 1
1.2 Structure of the Thesis 3
1.3 The Subgraph Isomorphism Problem 5
1.3.1 Applications of Subgraph Isomorphism 6
1.3.2 Challenges and Solutions 7
1.3.3 Overview of Related Works 9
1.4 SubMultigraph Matching 11
1.4.1 Applications of SubMultigraph Matching 12
1.4.2 Challenges and Solutions 14
1.4.3 Overview of Related Works 16
1.5 Motif Discovery 18
1.5.1 Applications of Motif Discovery 18
1.5.2 Challenges in Motif Discovery 19
1.5.3 Approaches to Motif Discovery 20
1.5.4 Temporal Motif Discovery 21
1.5.5 Overview of Related Works 23
1.6 Graph Embedding 25
1.6.1 Applications of Graph Embedding 26
1.6.2 Methods for Graph Embedding 26
1.6.3 Challenges in Graph Embedding 27
1.6.4 Multiplex Graphs 28
1.6.5 Overview of Related Works 29

2 Preliminary Definitions **31**
2.1 Graphs . . . 31
2.1.1 Subgraph Isomorphism . . . 32
2.1.2 Summary of the notation for simple graphs . . . 36
2.2 Multigraphs . . . 39
2.2.1 SubMultigraph Matching (SMM) . . . 40
2.2.2 Summary of the notation for multigraphs . . . 41
2.3 Temporal Graphs . . . 44
2.3.1 Temporal Subgraph Isomorphism (TSI) . . . 44
2.3.2 Temporal Motif Search (TMS) . . . 45
2.3.3 Summary of the notation for temporal graphs . . . 46
2.4 Multiplex Graphs . . . 48
2.4.1 Multiplex Graph Embedding Framework . . . 49
2.4.2 Summary of the notation for multiplex graphs . . . 49

3 ArcMatch **52**
3.1 Methods . . . 54
3.1.1 Path-based domain reduction . . . 55
3.1.2 Vertex ordering strategy . . . 62
3.1.3 Extension of partial mappings and dynamic parent selection . . . 66
3.1.4 Safety and completeness . . . 67
3.2 Experiments . . . 76
3.2.1 Benchmarks . . . 79
3.2.2 Subgraphs Isomorphism . . . 82
3.2.3 Subgraphs isomorphism with a limited number of matches 87
3.2.4 Scalability Tests on Synthetic Graphs . . . 88
3.3 Detailed comparison of the techniques involved in *ArcMatch* . 93
3.4 Relation between arc consistency and path-based reduction . . 103
3.5 Discussion . . . 105

4 MultiGraphMatch **107**
4.1 CYPHER Query Language . . . 108
4.2 Description of MultiGraphMatch . . . 111
4.2.1 Indexing of target network . . . 112
4.2.2 Computation of symmetry breaking conditions . . . 115
4.2.3 Computation of compatibility domains . . . 118
4.2.4 Determining the processing order of query edges . . . 121
4.2.5 Matching process . . . 123
4.2.6 Handling queries with WHERE clause . . . 127
4.2.7 Complexity analysis . . . 128

4.3 Experimental Results . . . 131
4.3.1 Experiments on synthetic networks . . . 132
4.3.2 Experiments on real networks . . . 139
4.3.3 Scalability tests . . . 142
4.4 Conclusion . . . 152

5 MODIT **153**
5.1 Methods . . . 153
5.1.1 The MODIT algorithm . . . 153
5.1.2 MODIT complexity analysis . . . 158
5.2 Results . . . 161
5.2.1 Experiments on Dataset 1 . . . 162
5.2.2 Comparison with Paranjape's algorithm . . . 163
5.3 Discussion . . . 170

6 MPXGAT **171**
6.1 Methods . . . 172
6.1.1 The MPXGAT General Framework . . . 172
6.1.2 MPXGAT Implementation for Link Prediction . . . 175
6.2 Results . . . 177
6.2.1 Dataset . . . 177
6.2.2 Experimental setup . . . 179
6.2.3 Competing Methods . . . 180
6.2.4 Embedding Multiplex Networks . . . 180
6.2.5 Measure the impact of Horizontal Embeddings . . . 181
6.3 Discussion . . . 185

7 Conclusions **186**
7.1 Summary of Contributions . . . 186
7.2 Broader Impact . . . 187
7.3 Future Work . . . 189
7.4 Final Remarks . . . 190

Bibliography **191**

List of tables **209**

List of figures **212**

List of algorithms **213**

1 Introduction

1.1 Context and Motivation

Graphs are essential structures in modern computational science, providing an intuitive yet powerful means of modeling relationships and interactions in complex systems across a wide range of disciplines. Their applicability spans bioinformatics, social network analysis, computational chemistry, and beyond, offering a versatile framework for representing entities (as vertices) and their connections (as edges) [10, 112].

In bioinformatics, graphs are extensively used to model biological systems such as protein-protein interaction (PPI) networks, where vertices represent proteins, and edges depict interactions between them. Understanding these interactions is crucial for deciphering cellular processes and identifying potential therapeutic targets for drug discovery [77, 140].

Similarly, in social network analysis, graphs are employed to model and analyze relationships between individuals or organizations, with nodes representing people and edges representing various kinds of social ties, including friendships, professional connections, and communication pathways [54]. Tasks such as community detection, influence analysis, and the modeling of information diffusion depend heavily on advanced algorithms capable of efficiently identifying subgraph structures that reveal underlying patterns of

social dynamics.

In computational chemistry, graphs are widely used to model molecular structures. Atoms are represented by vertices, and chemical bonds are depicted as edges, allowing chemists to search for specific substructures within large chemical databases, predict molecular properties, and design new compounds [8, 112].

As the size and complexity of real-world networks continue to expand, traditional algorithms face significant limitations. Many existing approaches fail to capture the intricacies of labeled graphs, where both vertices and edges can have multiple attributes. Additionally, newer graph models, such as temporal networks and multiplex networks, introduce additional layers of complexity by incorporating the dimension of time or multiple types of relationships between entities [70, 18]. This demands the development of more sophisticated algorithms that can handle not only labeled graphs but also these advanced structures.

In response to these challenges, this thesis introduces a comprehensive set of novel methodologies designed to tackle various aspects of graph analysis. It presents advanced techniques for subgraph matching, not only in simple graphs but also in the more complex context of multigraphs, where multiple types of edges and relationships between nodes exist. Furthermore, the work extends beyond subgraph matching to address motif discovery in temporal graphs, where the timing and sequence of interactions are critical, and explores innovative approaches to graph embedding in multiplex networks, which model systems with multiple layers of relationships. These contributions are aimed at enhancing the scalability, flexibility, and effectiveness of graph querying and analysis across a wide range of applications, including bioinformatics, social sciences, and chemistry.

1.2 Structure of the Thesis

This thesis is organized into several chapters, each detailing different aspects of the research.

Chapter 1: Introduction

Provides an overview of the research context, the significance of the subgraph isomorphism problem, graph embedding, and motif discovery, along with the main contributions of the thesis.

Chapter 2: Preliminary Definitions

This chapter introduces the fundamental concepts and definitions that are essential for the rest of the thesis. It begins with basic graph-theoretic concepts such as graphs, vertices, edges, and their respective properties. Then, it delves into more specialized definitions like subgraph isomorphism, multigraphs, temporal graphs, and multiplex networks. These foundational definitions provide the necessary background for understanding the algorithms and methods discussed in later chapters. Special attention is given to the various forms of graph matching (Subgraph Isomorphism, SubMultigraph Matching, Temporal Subgraph Isomorphism) and motif discovery in temporal networks, all of which are central to the proposed solutions. Additionally, chapter 6 introduces important concepts related to graph embedding in multiplex graphs, setting the stage for the MPXGAT model.

Chapter 3: ArcMatch

In this chapter, we explore the ArcMatch algorithm designed for subgraph matching in labeled graphs. Unlike most existing software that handles only vertex labels, ArcMatch efficiently processes graphs with both vertex and edge labels [22]. The chapter includes a detailed description of ArcMatch's

novel methods. The experimental section demonstrates ArcMatch's superior performance across various real-world scenarios, such as protein-protein interaction graphs and social networks.

Chapter 4: MultiGraphMatch

This chapter introduces MultiGraphMatch, which extends subgraph matching to multi-relational graphs where nodes can be connected by multiple types of edges [107]. MultiGraphMatch uses a novel "bit signature" data structure for efficient indexing and filtering of candidate edges. It also proposes a new edge processing order and symmetry breaking conditions to eliminate redundant matches. The chapter compares MultiGraphMatch's performance with other graph database systems, showcasing its efficiency in diverse synthetic and real-world graphs.

Chapter 5: MODIT

MODIT is presented as a solution for motif discovery in temporal networks, where edges have timestamps. The algorithm is capable of counting motifs of any size in temporal networks. This chapter details the MODIT algorithm, and presents experimental results that highlight its capability to efficiently retrieve large motifs in various networks [122].

Chapter 6: MPXGAT

This chapter focuses on MPXGAT, an attention-based deep learning model for multiplex graph embedding. By leveraging Graph Attention Networks (GATs), MPXGAT captures the complex relationships within and across the layers of multiplex networks. The chapter describes the model architecture, implementation, and performance evaluation on benchmark datasets, demonstrating MPXGAT's superiority over state-of-the-art methods.

Chapter 7: Conclusions

Summarizes the findings, discusses the implications of the research, and suggests directions for future work.

1.3 The Subgraph Isomorphism Problem

A particularly challenging and crucial task within graph theory is the subgraph isomorphism problem. This problem involves determining whether a smaller graph (the query) exists within a larger graph (the target). Subgraph matching is fundamental for various applications, including identifying functional modules in biological networks, querying molecular databases, detecting anomalies in hardware, and recognizing reusable patterns in software design [34, 150].

The subgraph isomorphism problem is formally defined as follows: given a query graph Q and a target graph T, the objective is to find a one-to-one mapping between the vertices of Q and a subset of the vertices of T such that the adjacency relations (i.e., edges) are preserved. This means that for every edge in the query graph, there must be a corresponding edge in the target graph with the same connectivity and, in the case of labeled graphs, the same labels on vertices and edges [32].

The complexity of the subgraph isomorphism problem is well-known; it is NP-complete, which implies that there is no known polynomial-time algorithm that can solve all instances of this problem efficiently. This computational difficulty necessitates the development of heuristic and approximate methods that can provide solutions within a reasonable timeframe, especially for large and complex graphs [52]. Recent advancements like the ArcMatch algorithm [22], which builds on the work of existing backtracking algorithms and domain filtering techniques, have demonstrated significant improvements in solving labeled subgraph matching problems.

1.3.1 Applications of Subgraph Isomorphism

Biological Networks

In bioinformatics, the subgraph isomorphism problem is used to identify functional modules within protein-protein interaction (PPI) networks. These modules can represent groups of proteins that work together to perform specific biological functions. Identifying such modules can help in understanding cellular processes and can lead to the discovery of new drug targets [128]. Additionally, subgraph matching can be used to find occurrences of known biochemical pathways within larger networks, aiding in the annotation and interpretation of experimental data.

Molecular Databases

In computational chemistry, atoms are modeled as vertices labeled with atom symbols, and edges represent chemical bonds [8]. Living cells are often modeled via protein-protein interaction networks [134], where vertices are proteins and edges are their interactions. Vertex labels may denote functional properties of a protein, and edge labels might denote the type of interaction, for example, physical bond or expression correlation. ArcMatch efficiently processes both vertex and edge labels in such graphs, making it a valuable tool for analyzing molecular structures [22].

Hardware Security

Detecting hardware trojans, which are malicious alterations in integrated circuits, is another important application of subgraph isomorphism. By representing circuits as graphs, with components as vertices and connections as edges, security analysts can use subgraph matching to detect unauthorized modifications that could compromise the integrity and functionality of the hardware [109].

Software Design

In software engineering, subgraph isomorphism helps in recognizing reusable patterns in software design. Design patterns are typical solutions to common problems in software design, and identifying instances of these patterns in existing codebases can facilitate software maintenance and evolution. By using subgraph matching, developers can automate the detection of these patterns, making it easier to refactor and improve the design of software systems [50].

1.3.2 Challenges and Solutions

The inherent complexity of the subgraph isomorphism problem poses significant challenges, especially when dealing with large-scale and richly labeled graphs. Traditional algorithms often struggle with the combinatorial explosion of potential mappings, leading to prohibitive computation times. Recent advancements like ArcMatch, which builds on domain filtering and cost-based searching techniques, have focused on several key strategies to address these challenges [22]:

Domain Reduction

Domain reduction techniques aim to prune the search space by eliminating candidate vertices and edges that cannot be part of a valid mapping. By applying constraints based on vertex and edge labels, as well as topological features, these methods can significantly reduce the number of possibilities that need to be explored [35]. ArcMatch extends these methods by incorporating advanced path-based domain reduction techniques to optimize the performance in complex labeled graphs [22].

Efficient Data Structures

The use of sophisticated data structures, such as vertex and edge domains, path indices, and bit signatures, can facilitate faster lookups and matching operations. These structures help in organizing the graph data in a way that supports efficient querying and manipulation [87]. ArcMatch uses efficient indexing methods to handle both vertex and edge domains, significantly improving the time complexity of matching large graphs [22].

Heuristic Search Algorithms

Heuristic approaches, such as those based on the most-constrained-first principle, guide the search process by prioritizing mappings that are more likely to lead to a solution. These methods leverage domain-specific knowledge to improve the efficiency of the search [9].

Parallel and Distributed Computing

Exploiting the power of parallel and distributed computing can help tackle the subgraph isomorphism problem by dividing the work across multiple processors or machines. This approach can lead to significant speedups, especially for large graphs [126].

Machine Learning Techniques

Emerging machine learning techniques are being explored to predict and prioritize subgraph matches. These methods can learn patterns from previously solved instances and apply this knowledge to new queries, potentially reducing the search space and computation time [162].

By addressing these challenges through innovative algorithms and techniques, the field of graph theory continues to make significant strides in solving the subgraph isomorphism problem. This progress not only enhances

our ability to analyze and interpret complex networks but also opens up new possibilities for applications across various domains [21].

1.3.3 Overview of Related Works

We focus our attention only on algorithms that solve the definition of subgraph isomorphism in which distinct query vertices must match distinct target vertices, as originally formulated in [150] and in the main related literature.

The querying problem is known as subgraph isomorphism (SubGI) and is known to be NP-complete [33].

Backtracking algorithms have proven to have among the best performance [150, 36, 27, 21], but other approaches based on Cartesian products or join operations have been proposed [17, 138]. By analogy to the constraint satisfaction problem (CSP), query vertices can be seen as variables and the goal is to assign values, namely target vertices, to such variables such that constraints are satisfied [158]. The constraints include topological constraints, labeling constraints as discussed above, as well as the *all different* property of the mapping [132].

An exponential brute force algorithm tries to assign all possible values to variables, and for each complete assignment, the constraints are verified a posteriori. Heuristic algorithms use backtracking, for which the search space can be represented as a tree. The tree has a dummy root, and it has potentially ${V_q}^{|V_t|}$ nodes, plus the root. Each tree node (except the root) represents a vertex mapping (q_i, t_i). A path from each child of the root to a leaf of the tree represents a complete mapping solution. A path from the root to an intermediate node represents a partial solution. The tree defines the order in which mappings are investigated. The process of tree construction does not check constraints, so constraints must be verified separately. That verification can be performed for a full mapping at the leaves of the tree or

for a partial mapping at intermediate nodes. In the latter case, if a partial solution violates at least one constraint, then the subtree branching from its last tree node can be ignored. In this way, the size of the search space can be drastically reduced.

Several heuristic methodologies for solving subgraph isomorphism have been proposed over the years. An early solution was presented in [150] by Ullmann, which models the problem as a backtracking search. The search space can be seen as a tree which encodes partial mappings and their extensions to complete solutions.

In [36, 27], the authors enhanced the search with a set of look-head rules for predicting unfeasible branches of the search space. In [132, 103], concepts developed for solving constraint satisfaction problems have been applied. Constraint satisfaction aims at verifying constraints among a set of variables once specific values are assigned to them. The application to subgraph isomorphism represents query vertices as variables whose possible values are target vertices. Labels and edges are represented as constraints of the problem. The set of target vertices that are compatible (i.e. they have the same label) with a given query vertex is called vertex domain. Thus, feasible combinations of target vertices are tested to assess the correspondence of their relationships with the query topology and labels. Sophisticated procedures, called reduction techniques, are applied to refine domains before the verification step, thus reducing the search space and thus (hopefully) decreasing the running time. Recently [64, 17, 63, 84, 83], the concept of domains has been extended to edges. Such approaches outperform previous methodologies based on graph algebra [66] and query decomposition strategies [138]. This entailed the definition of a new data structure in which feasible correspondences between vertices and edges are stored. A disadvantage of the domain-based approach is the high memory consumption, especially in the presence of edge domains. In [21], a matching order strategy of the query

vertices has been defined to maximize the number of constraints that are verified at each step of the computation. Such an approach has been shown to work well to define the ordering of the variables [135, 89, 20, 5].

1.4 SubMultigraph Matching

SubMultigraph Matching (SMM) is a generalization of the Subgraph Matching problem, which involves finding occurrences of a query multigraph within a target multigraph. In multigraphs, multiple directed or bidirectional edges may exist between pairs of nodes, representing different types of relationships or interactions. Both nodes and edges in a multigraph can be labeled and attributed, further increasing the complexity of the matching process. This thesis builds on these challenges by introducing MultiGraphMatch, a specialized algorithm designed to handle the intricacies of multi-relational graph structures efficiently [107].

Multigraphs are widely used to model complex systems in which entities interact in multiple ways. For example, in social networks, nodes may represent individuals, and edges can represent different types of relationships, such as friendships, professional connections, or online interactions. Similarly, in biological networks, multiple types of interactions may exist between proteins or genes, such as genetic regulation, physical binding, or co-expression. The diversity of relationships captured in multigraphs makes the SubMultigraph Matching problem (SMM) particularly challenging, as both the structure and the labels of nodes and edges must be accounted for. MultiGraphMatch addresses this challenge through a novel bit signature data structure that efficiently indexes and filters candidate matches, enabling faster querying of multi-relational data [107].

The SMM problem is computationally challenging due to the need to match both the structure of the query graph and the labels and attributes of

nodes and edges. Furthermore, the existence of multiple edges between nodes introduces additional complexity compared to the simpler Subgraph Matching problem, which typically assumes at most one edge between any two nodes. MultiGraphMatch introduces a new processing order for edges and symmetry-breaking conditions that reduce redundant matches, significantly improving performance in both synthetic and real-world graphs [107]. Despite these challenges, SMM has numerous applications across various fields, as described below.

1.4.1 Applications of SubMultigraph Matching

Social Networks

In social network analysis, multigraphs are used to represent the multiple types of interactions that can occur between individuals or groups. Nodes represent people, while edges represent relationships such as friendships, family connections, co-workers, or followers on social media platforms. The ability to perform SubMultigraph Matching allows for the detection of specific patterns of interaction within a larger social network. For instance, it can be used to identify communities of individuals connected through multiple types of relationships, or to detect influential figures who play different roles in various social contexts [163].

Finding subgraphs in social networks can also help in understanding how different types of relationships influence the spread of information or behaviors. For example, a particular pattern of professional and personal connections might accelerate the diffusion of innovations or marketing messages. By identifying such patterns, companies can optimize marketing strategies or design more effective information campaigns [130].

Biological Networks

In bioinformatics, multigraphs are frequently used to model complex biological systems where multiple types of molecular interactions occur between entities such as proteins, genes, and metabolites. For example, in a protein-protein interaction network, different types of edges may represent physical interactions, genetic co-expression, or functional associations. SubMultigraph Matching allows researchers to detect specific interaction patterns, such as protein complexes or signaling pathways, within these networks [77, 128].

This approach is particularly useful for drug discovery, where the goal is to identify molecular substructures that correspond to specific biological functions or disease mechanisms. By finding instances of known biochemical pathways within larger interaction networks, researchers can gain insights into potential targets for therapeutic intervention. SubMultigraph Matching can also assist in identifying conserved patterns of interaction across different species, which may reveal fundamental biological processes [140].

Infrastructure Networks

Infrastructure networks, such as power grids, transportation systems, or communication networks, often involve multiple types of connections between nodes representing physical components (e.g., power stations, routers, or vehicles). In these systems, multigraphs can model various types of interactions, such as electrical connections, data transfers, or transportation routes. SubMultigraph Matching is valuable in this context for identifying vulnerabilities, redundancies, or inefficiencies within the network [123].

For instance, in power grid analysis, finding subgraph patterns corresponding to critical infrastructure components and their interdependencies can help identify points of failure that could lead to cascading outages. Similarly, in transportation networks, SubMultigraph Matching can be used to

detect patterns that lead to traffic congestion or inefficiencies in the routing of goods. By understanding these patterns, engineers can design more resilient and efficient infrastructure systems [145].

Knowledge Graphs

Knowledge graphs are a form of multigraph where nodes represent entities (e.g., people, places, or concepts), and edges represent relationships between these entities (e.g., "is a type of", "works at", "located in"). These graphs are widely used in artificial intelligence for tasks such as information retrieval, natural language processing, and semantic reasoning [2, 153].

SubMultigraph Matching in knowledge graphs allows for the identification of specific patterns of knowledge, such as hierarchies or dependencies between entities. For example, in a knowledge graph representing a scientific domain, matching subgraphs could help discover relationships between concepts that were previously unknown or poorly understood. This capability is critical for tasks such as automatic query answering, where the system must identify relevant subgraphs that correspond to the query's semantics [6].

1.4.2 Challenges and Solutions

The inherent complexity of the SMM problem poses significant challenges, particularly when dealing with large-scale multigraphs where nodes and edges have multiple labels and attributes. The multiplicity of edges, combined with the need to match node and edge attributes, creates a combinatorial explosion of possibilities for matching, which can lead to prohibitive computation times.

Several approaches have been proposed to address these challenges, including:

Indexing Techniques

Indexing techniques are used to pre-process the target graph by constructing data structures that allow for fast lookups of candidate matches for nodes and edges. These indexes capture information about node labels, edge types, and neighborhood structures, allowing the algorithm to quickly eliminate portions of the target graph that cannot possibly match the query [73].

Domain Pruning

Domain pruning techniques aim to reduce the search space by eliminating candidate nodes and edges that cannot be part of a valid match. By applying constraints based on the labels and attributes of nodes and edges, as well as the multiplicity of connections between nodes, these techniques can significantly reduce the number of potential matches that need to be explored [111].

Heuristic Search

Heuristic search algorithms guide the matching process by prioritizing the exploration of parts of the target graph that are more likely to contain valid matches. These heuristics may take into account factors such as the density of connections, the similarity of node labels, or the structural features of the graph. By focusing the search on promising areas of the graph, heuristic methods can reduce computation times for large graphs [137].

Parallel and Distributed Computing

Given the complexity of the SMM problem, parallel and distributed computing techniques are often employed to divide the matching process across multiple processors or machines. This approach allows for significant speedups, particularly for large graphs, where different parts of the graph can be processed simultaneously [126].

1.4.3 Overview of Related Works

Solutions proposed to solve the SMM problem can be classified into two categories: in-memory algorithms and graph database systems.

In-memory algorithms include SumGra [73], Moorman's [111] and FGq_t-Match [137]. SuMGra [73] builds two indexes based on specific multigraph properties of the target, such as node and edge multiplicities, i.e. the number of node labels and the number of edges linking two nodes, respectively. The first index is the vertex signature, which captures information about the labels of edges incident on target nodes together with their multiplicities. The second index is the vertex neighborhood index, which contains information about the neighbors of a given node u, the edges connecting u to its neighbors and the labels of each edge. Indexes are then used to select candidate nodes for the initial query node and the subsequent nodes during the search phase. Moorman's algorithm [111] introduces a series of filtering methods to solve subgraph matching and similar problems on multigraphs. Filters are defined based on node labels, node-level statistics, topology, arc consistency and local neighborhood of nodes. Filters are then applied iteratively until convergence, to narrow down the search space. The authors also present an isomorphism counting approach that can be applied after filtering to count the number of query occurrences. FGq_t-Match [137] builds a matching-driven flow graph, called FGq_t, to perform subgraph matching on knowledge graphs. Nodes of FGq_t are candidate target nodes for matching based on defined label constraints and each edge between a pair of candidates is redirected to an edge of FGq_t. A heuristic approach based on a multi-label weight matrix is followed to reduce the number of candidates. Matching is then performed by directly traversing FGq_t.

Graph database systems are databases in which data are modeled as directed and labeled multigraphs and data manipulation is done using graph-oriented operations, including subgraph matching [144]. Examples of graph

database systems include:

- Neo4J (https://neo4j.com/)
- MemGraph (https://memgraph.com/)
- ArangoDB (https://www.arangodb.com)
- OrientDB (https://orientdb.org)
- JanusGraph (https://janusgraph.org)
- TigerGraph [44]
- Titan (https://titan.thinkaurelius.com)

Many systems rely upon storage design schemes of relational and NoSQL databases, such as key-value, document, tuple and object-oriented stores. However, traditional schemes are not suitable in densely connected data like graphs, where most relationships are many-to-many and querying the database requires multiple expensive join operations, impacting the performance. In contrast, native graph databases like TigerGraph, Memgraph and Neo4j are specifically built to maintain and process graphs efficiently. Labeled Property Graphs (LPG) and Resource Description Framework (RDF) schemes [2, 153] are commonly used for data modeling. Most databases use adjacency lists to represent graphs in the storage layer to make traversal operations faster. Data replication is ensured to enable data distribution and parallelization. Many graph databases also provide external interfaces, specific declarative languages to define queries (e.g. AQL, CYPHER), query optimizers and transaction engines.

Despite the progress in this area, finding efficient solutions to the Sub-Multigraph Matching problem remains a significant challenge, particularly when dealing with large-scale graphs or when the query contains complex attribute and label constraints. The need for scalable and flexible algorithms

that can handle the intricacies of multigraphs is crucial in applications ranging from social network analysis to bioinformatics and knowledge graphs.

1.5 Motif Discovery

Motifs are small, recurring subgraphs that appear frequently within a larger graph. They are of particular interest because they often represent fundamental structural patterns or building blocks of the graph. Discovering motifs in graphs, especially temporal networks where edges are associated with timestamps, can provide deep insights into the underlying dynamics and interactions within the network. Motif discovery is a challenging computational problem due to the combinatorial explosion of possible subgraphs as the size of the motif increases [108, 72, 115]. This thesis builds on these challenges by introducing MODIT, an algorithm for motif discovery in temporal networks, specifically designed to handle the complexities of large motifs and dynamic graphs [122].

In social networks, motifs can reveal common interaction patterns, such as triads representing close-knit groups of friends or colleagues [114, 124]. In technological networks, such as the internet or power grids, motifs can help identify structural weaknesses or redundancies that are crucial for the network's robustness and reliability [142].

1.5.1 Applications of Motif Discovery

Biological Networks

In systems biology, motif discovery is used to identify and analyze recurring patterns in gene regulatory networks, metabolic networks, and protein-protein interaction networks. These motifs can reveal fundamental principles of cellular organization and function. For instance, the discovery of motifs in transcriptional regulatory networks has helped elucidate the mechanisms of

gene expression regulation and the modular organization of cellular processes [1, 75].

Social Networks

Motifs in social networks can provide insights into the structure and dynamics of social interactions. By analyzing motifs, researchers can uncover common patterns of communication, collaboration, and influence. This can lead to a better understanding of social cohesion, the spread of information or diseases, and the formation of communities [72, 68].

Ecological Networks

In ecology, motifs are used to study the interactions within food webs and other ecological networks. Identifying recurring patterns of species interactions can help in understanding the stability and resilience of ecosystems. For example, certain predator-prey motifs might be critical for maintaining the balance of an ecosystem [86].

Technological and Infrastructure Networks

Motifs in technological networks, such as transportation systems, power grids, and communication networks, can identify critical components and pathways that are essential for the network's functionality. Understanding these motifs can aid in the design of more robust and efficient networks, and in developing strategies for fault tolerance and disaster recovery [123, 145].

1.5.2 Challenges in Motif Discovery

The primary challenge in motif discovery is the combinatorial explosion of possible subgraphs as the size of the motif increases. For a graph with n nodes, the number of possible subgraphs grows exponentially with n. This

makes the problem computationally intensive, especially for large graphs [95, 93].

Computational Complexity

The process of identifying motifs involves enumerating all possible subgraphs of a given size, counting their occurrences, and comparing these counts to those in randomized versions of the graph. This requires significant computational resources, particularly for large networks and larger motif sizes [72, 95].

Scalability

Scalability is a significant issue in motif discovery. As the size of the graph and the complexity of the motifs increase, the algorithms must efficiently handle large amounts of data and perform computations within a reasonable time frame. This requires advanced data structures, parallel computing, and optimization techniques [72, 29].

1.5.3 Approaches to Motif Discovery

Several approaches have been developed to address the challenges of motif discovery, leveraging various algorithmic and computational strategies:

Subgraph Enumeration Algorithms

These algorithms systematically enumerate all possible subgraphs of a given size and count their occurrences. Techniques such as backtracking, branch-and-bound, and network sampling are commonly used to reduce the search space and improve efficiency [72, 95].

Randomized Algorithms

Randomized algorithms generate randomized versions of the network to establish a baseline for motif significance. By comparing the frequency of subgraphs in the original network to those in randomized networks, these algorithms can identify statistically significant motifs [86, 115].

Machine Learning Techniques

Machine learning approaches, particularly those involving pattern recognition and classification, are increasingly being used to identify motifs. These techniques can learn from known motifs and predict new ones, potentially reducing the need for exhaustive enumeration [93, 123].

Parallel and Distributed Computing

To handle large networks and complex motifs, parallel and distributed computing techniques are employed. By dividing the computational workload across multiple processors or machines, these approaches can significantly speed up the motif discovery process [38, 117].

1.5.4 Temporal Motif Discovery

A wide range of domains can be modeled and studied with static networks but many complex systems are fully dynamic, indeed interactions between entities change over time. Systems of this type can be modeled as *temporal networks*, in which edges between nodes are associated with temporal information such as, for example, the duration of the interaction and the instant in which the interaction begins. Annotations of edges with temporal data is important to understand the formation and the evolution of such systems. MODIT specifically addresses this dynamic nature by extending motif discovery to temporal graphs, where the sequence and timing of interactions are crucial for identifying recurring patterns [122].

Temporal Networks

In literature, several definitions of temporal networks have been proposed [70, 101]. In few works, these are also referenced as dynamic [28], evolutionary [1] or time-varying [29] networks. In this thesis, we define temporal network as a multigraph (i.e a graph where two nodes may interact multiple times). Each edge is associated with an integer, called timestamp, which denotes when two nodes interact [95].

In the last few years, there has been a growing interest in analyzing temporal networks and studying their properties. Analysis of temporal networks includes network centrality [97, 147], network clustering [38], community detection [124], link prediction [46], graph mining [136], graph embedding [146], network sampling [123], random models [131, 117, 67] and epidemic spreading [145, 100, 154]. Extensive reviews of temporal networks and their main features can be found in [70, 69, 101].

Temporal Motifs

In this thesis, we focus on motif search in temporal networks. Different definitions of temporal motifs have been proposed so far [86, 72, 115]. Here, we follow the most recent definition proposed in [115], which is becoming the most accepted one. A temporal motif is a temporal network where edges denote a succession of events. In addition to the original definition proposed by [115], simultaneous events, represented by edges with equal timestamps, are allowed, provided that such edges do not link the same pair of nodes. Temporal graphs Q_1 and Q_2 of Figure 1.1 on page 24 are two examples of motifs. Applications of Temporal Motif Search include the creation of evolution rules that govern the way the network changes over time [16, 149] allowing also to identify all the time an edge participates to particular pattern in a time window. A second application consists in the identification of motifs in temporal network at different time resolution to identify patterns

at different time scale. Another application consists in temporal network classification using a feature representation based on the temporal motifs distribution [148].

Given a time interval Δ, we say that a motif Q Δ-occurs in T, iff: i) Q is isomorphic (i.e. structurally equivalent) to a subgraph S of T (called an occurrence of Q in T), ii) edges in S follow the same chronological order imposed by corresponding matched edges in Q, iii) all interactions in S are observed in a time interval less than or equal to Δ (i.e. they are likely to be related each other). In Figure 1.1 motif Q_1 Δ-occurs ($\Delta = 6$ in the example) in T, while Q_2 does not.

For a given temporal graph T and time interval Δ, motif search aims at retrieving all motifs that Δ-occurs in T. In addition, for each such motif Q, we also count the number of occurrences of Q in T. It has been shown that Temporal Motif Search (TMS) problem is NP-complete, even for star topologies [93]. For this reason, motif search is usually restricted to motifs with up to a certain number of nodes and edges. Given $\Delta = 10$, Figure 1.2 shows all temporal motifs with up to 3 nodes and 3 edges that Δ-occur in a toy temporal graph, together with the corresponding number of occurrences.

1.5.5 Overview of Related Works

Recently, several TMS algorithms have been introduced [86, 115, 93, 72]. However, the proposed solutions are limited to very small motifs or specific topologies.

Temporal motifs have been introduced for the first time by [86]. Authors define a motif as an ordered set of edges such that: i) the difference between the timestamps of two consecutive edges must be less than or equal to a certain threshold Δ and ii) if a node is part of a motif, then all its adjacent edges have to be consecutive (consecutive edge restriction).

In [72] the consecutive edge restriction was relaxed and the authors con-

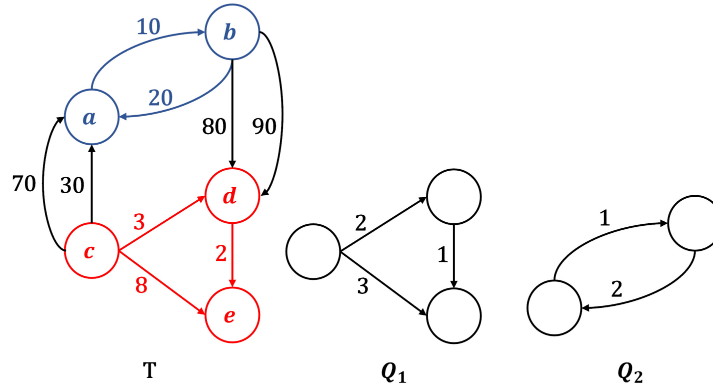


Figure 1.1: Example of motif Δ-occurrence in a temporal graph T, given $\Delta = 6$. Motif $\mathbf{Q_1}$ has exactly one Δ-occurrence in $\mathbf{T}$, which is the subgraph formed by nodes and edges colored in red. Motif $\mathbf{Q_2}$, instead, does not Δ-occur in T. In fact, the subgraph with blue nodes and blue edges is isomorphic to $\mathbf{Q_2}$ and respects the chronological order imposed by Q_2's edges, but its edges are not observed within the time window Δ.

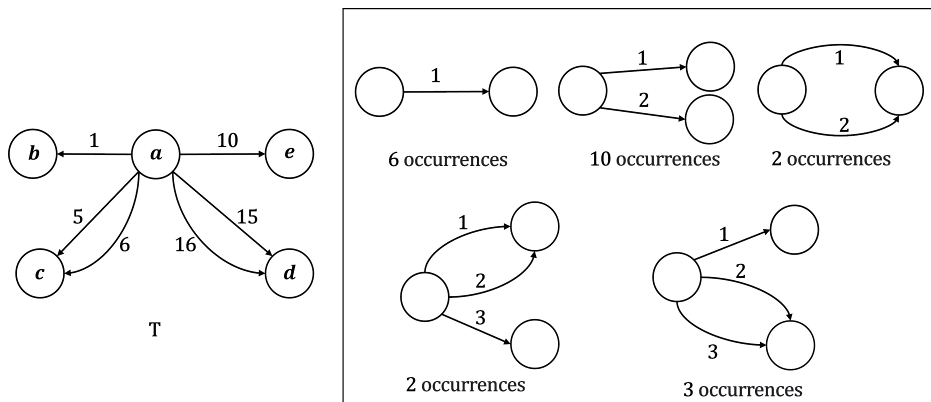


Figure 1.2: Example of application of the Temporal Motif Search (TMS) problem for a temporal graph $\mathbf{T}$, given $\Delta = 10$, $k = 3$ and $l = 3$. For each motif, the relative number of Δ-occurrences in T is reported.

sidered only induced subgraphs, called graphlets, in order to reduce the computational complexity while obtaining approximate results.

[115] describes a temporal motif as a sequence of edges ordered by increasing timestamps. More precisely, the authors define a k-node, l-edge, Δ-temporal motif as a sequence of l edges, $M = (u_1, v_1, t_1), (u_2, v_2, t_2), \dots, (u_l, v_l, t_l)$ that are time-ordered within a Δ duration, i.e., $t_1 < t_2 \dots < t_l$ and $t_l - t_1 \leq \Delta$, such that the static graph induced by the edges is connected and has k nodes. The authors present an algorithm to efficiently calculate the frequencies of all possible directed temporal motifs with 3 edges. For bigger motifs they use a naive algorithm that first computes static matches, then filters out occurrences which do not match the temporal constraints.

To tackle with the NP-completeness of TMS, approximate solutions have been proposed too. [93] propose a general sampling framework to estimate motif counts. It consists in partitioning time into intervals, finding exact counts of motifs in each interval and weighting counts to get the final estimate, using importance sampling.

1.6 Graph Embedding

A topic that has recently received considerable interest in computer science is that of how to efficiently represent large-scale graphs [7, 31, 59]. Particularly, graph embedding methods, which consist in projecting the elements of a graph, i.e., vertices, edges, and motifs, to a low-dimensional vector space by preserving some of the graph properties, have shown to be very successful in graph representation [82].

1.6.1 Applications of Graph Embedding

Node Classification

Node classification aims to assign labels to the nodes of a graph. In social networks, for example, this could involve predicting the interests or demographics of users. Graph embeddings provide a way to represent nodes in a feature space where traditional classification algorithms can be applied effectively [85, 152].

Link Prediction

Link prediction involves predicting the existence of edges between nodes in a graph. This task is crucial in various domains such as recommender systems, social network analysis, and biological network analysis. By embedding nodes into a vector space, the likelihood of future or missing links can be estimated based on the proximity of their embeddings [96, 60].

Clustering

Clustering in graphs refers to grouping nodes into clusters based on their similarity or connectivity. Graph embeddings enable the use of traditional clustering algorithms, such as k-means, by providing a vector representation of nodes that reflects their structural and feature-based similarities [141, 30].

1.6.2 Methods for Graph Embedding

We can broadly categorize graph embedding methods into traditional graph embedding and graph neural networks (GNNs) based embedding methods [82]. The first group consists of algorithms that represent graphs relying on techniques such as random walks [116, 141, 60] and matrix factorization methods. These shallow approaches have several drawbacks that limit their efficiency and effectiveness. First, they do not share any parameters

between the nodes in the encoder function, which maps each node to a vector representation, making them statistically and computationally inefficient. Second, they ignore the node attributes during the encoding process, leading to a lower quality of the embeddings. Third, they are transductive, meaning they can only generate embeddings for the nodes seen during the training phase [62].

Embedding methods based on GNNs go beyond such limitations by using more sophisticated encoders accounting for the graph structure and the node attributes. The key feature of a GNN is that the embedding of a node in the graph is obtained by *aggregating* the embeddings of the node's neighbors [82]. In the last few years, a large variety of methods have been developed [31, 152, 92, 51], with the most notable examples including Graph Convolutional Networks (GCNs) [85], GraphSAGE [61], and Graph Attention Networks (GATs) [152].

The thesis further builds on this line of research by introducing MPX-GAT, an attention-based model specifically designed for embedding multiplex graphs [19]. MPX-GAT leverages the GAT mechanism to capture both intra-layer and inter-layer dependencies in multiplex networks. It enhances traditional GNNs by introducing attention-based layers that prioritize important relationships between nodes, making it especially effective for tasks such as link prediction and community detection across multiple layers of the network.

1.6.3 Challenges in Graph Embedding

While graph embedding has proven to be powerful, several challenges remain:

Scalability

Scalability is a significant challenge, particularly for large-scale graphs with millions of nodes and edges. Efficient algorithms and parallel computing

techniques are essential to handle such large datasets [162, 90].

Dynamic Graphs

Many real-world graphs are dynamic, with nodes and edges appearing or disappearing over time. Embedding techniques need to adapt to these changes without retraining from scratch, necessitating incremental or dynamic embedding methods [164, 57].

Heterogeneous Graphs

Graphs often contain different types of nodes and edges, known as heterogeneous graphs. Embedding techniques must account for this heterogeneity to accurately capture the graph's structure and semantics [30, 129].

Preserving Structural and Semantic Information

Balancing the preservation of structural (topological) and semantic (feature-based) information in the embeddings is a complex task. Embeddings must effectively capture the graph's connectivity patterns and the attributes of nodes and edges [159, 161].

1.6.4 Multiplex Graphs

Although graphs are widely recognized for their versatility in modeling complex systems, traditional graph structures are limited to depicting a singular relationship type among system entities. This simplistic representation is insufficient for systems where entities engage in diverse interactions [88]. For example, social systems often involve individuals connected through various relationships such as friendship, kinship, or professional collaboration, and they may communicate through multiple channels like direct contact, phone, or online platforms [139, 12]. Similarly, in biological systems, proteins may interact genetically, physically, or through spatial proximity [42].

To encapsulate the complexity of such systems, advanced mathematical structures like multidimensional and multiplex graphs are employed [18, 15, 37]. Multidimensional graphs, or edge-labeled multigraphs, feature vertices connected by edges of different labels, each signifying a distinct interaction type within the system. Multiplex graphs, on the other hand, consist of multiple interconnected layers, with each layer dedicated to a specific relationship type.

The key distinction between multigraphs and multiplex networks lies in the representation of system units. In multigraphs, a unit is depicted as a single node linked to others through various connection types. Multiplex networks [41, 12], however, represent units as a set of nodes distributed across different layers, connected by inter-layer links. Typically, a unit is represented in only a subset of layers, indicating connections through certain relationship types. Furthermore, it is often the case that only a portion of the inter-layer links connecting the multiple representations of the same unit is known [37].

1.6.5 Overview of Related Works

Multi-relational network embedding is a challenging problem recently garnering significant research interest. Various methods to embed networks with multiple types of nodes and links, such as multidimensional networks and multiplex networks, have been proposed. Some of these methods are based on shallow embedding approaches [94, 160, 129, 56], while others use graph convolutional networks (GCNs) [155, 74, 71] or graph attention networks (GATs) [30]. However, none of these methods can solve the problem of predicting links between different layers of a multilayer network. This problem is important when the network structure is incomplete or heterogeneous. For example, some methods assume that all nodes are present in every layer, which is not realistic in many cases. Other methods (such as [14]; [13];

[119]) do not distinguish between intra-layer and inter-layer links, ignoring the diversity and complexity of multilayer networks. Very recently, MultiplexSAGE, a generalization of GraphSAGE aimed at embedding multiplex networks by relaxing these hypothesis, have been proposed [49], yet there is still a need for new methods that can address the problem of predicting inter-layer links in a general, flexible and more reliable way.

The inter-layer link prediction problem finds many crucial applications: i) linking user identities across different online social networks (OSNs), an emerging task in social media that has attracted increasing attention in recent years [130]. User identity linkage finds potential impact in different domains, from recommendation systems to cybersecurity [143]; ii) identifying the same genes or proteins across different biological networks, such as gene expression, protein interaction, metabolic pathways, etc. This can help to discover the molecular mechanisms of diseases and to find potential drug targets [76]; iii) matching the same entities across different knowledge graphs [6], which can help to enrich the semantic information and to improve query answering and reasoning capabilities. The range of possible applications of inter-layer link prediction ultimately motivates the development of embedding algorithms for multiplex networks that are able to distinguish between intra-layer and inter-layer links, and to reconstruct both connectivity patterns.

Predicting inter-layer connections in multiplex networks differs from graph matching or alignment. Graph matching [102] identifies isomorphisms between graphs, ensuring identical topology. In contrast, inter-layer prediction uncovers missing links between nodes representing the same unit across layers, with diverse connectivity. Global graph alignment [98] finds optimal correspondences between multiple graphs, considering both structure and node labels. However, inter-layer link prediction focuses on a specific bijection between two graphs, i.e., layers of a multiplex network. Also, while graph alignment is general, inter-layer prediction is specific to multiplex networks.

2 Preliminary Definitions

This chapter introduces the fundamental concepts and definitions that are used across the various topics discussed in this thesis. These definitions provide the foundation for understanding the algorithms and techniques in ArcMatch, MultiGraphMatch, MODIT and MPXGAT.

2.1 Graphs

Graphs are mathematical objects used to represent the overall topological relationship among a given set of entities. Entities can be represented by vertices, while their pairwise relationships can be represented by edges. Such a topological structure is often enriched with labels that represent specific properties of vertices and edges.

Definition 1 (Graph)**.** A simple graph is a tuple $G = (V, E)$, where V is the set of vertices, and E is the set of edges. An undirected edge is a set $\{v, u\}$, with $v, u \in V$. A path is a sequence of vertices $(v_1, v_2, \ldots, v_n)$, such that $v_j \in V$ for $1 \leq j \leq n$ and $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq n-1$. A vertex may appear multiple times in a path. A ring (or cycle) is a path $(v_1, v_2, \ldots, v_n)$ such that $v_1 = v_n$ and these two vertices are the only two to be repeated in the path.

Definition 2 (Degree)**.** The degree of a vertex u, denoted $deg(u)$, is the number of edges incident to u. The neighborhood of a vertex u, denoted $N(u)$, is the set of vertices adjacent to u.

Definition 3 (Labeles)**.** A graph can be equipped with labels on vertices and edges. Let A be the set of vertex labels, and B the set of edge labels. A function $\alpha : V \mapsto A$ assigns a label to each vertex, and a function $\beta : E \mapsto B$ assigns a label to each edge.

2.1.1 Subgraph Isomorphism

Definition 4 (Subgraph Isomorphism (SubGI))**.** Given a query graph $G_q = (V_q, E_q)$ and a target graph $G_t = (V_t, E_t)$, the subgraph isomorphism problem (SubGI) consists in finding occurrences of G_q in G_t. Each occurrence is an injective mapping $M : V_q \mapsto V_t$ that preserves the topology and labels of the query.

Figure 2.1 on page 37 gives an example of a use case where distinct query nodes should map to distinct target nodes. The query is to verify within a co-authorship network how many cliques of 3 authors (q_0, q_1 and q_2) that are affiliated with the same department have co-authored a paper with at least two other authors (q_3 and q_4). This example query might be part of research studying cooperation among members of the same department.

Definition 5 (Occurrence)**.** An occurrence is an injective mapping of query vertices to target vertices that preserves the topology and the labeling of the query graph. A mapping can be seen as a function $M : V_q \mapsto V_t$, such that $M(q_i) = t_i$. It is also represented as an ordered vector $M = ((q_1, t_1), (q_2, t_2), \ldots, (q_n, t_n))$, such that $q_i \in V_q$, $t_i \in V_t$. Each pair in M represents the mapping of a query vertex q_i to the target vertex t_i. The *all different* property must hold, viz. $\forall i, j \ s.t. \ i \neq j$, $q_i \neq q_j$, and

$t_i \neq t_j$. The mapping preserves the query topology if there exists a set of edges between the mapped target vertices that is equivalent to the set of query edges. Thus, $\forall i, j \ s.t. \{q_i, q_j\} \in E_q \Rightarrow \{M(q_i), M(q_j)\} \in E_t$. The preservation of labels implies $\forall i \ s.t. \ 1 \leq i \leq n \ \alpha(q_i) = \alpha(t_i)$, and $\forall i, j \ \{q_i, q_j\} \in E_q \Rightarrow \beta(\{q_i, q_j\}) = \beta(\{M(q_i), M(q_j)\})$.

Note 1. Recall that there is at most one edge between two nodes and each edge has one label.

Multiple mappings obeying the above constraints can exist, and each mapping identifies a given occurrence. The goal of SubGI is either to find all such occurrences or all occurrences up to a certain limit (e.g. 100,000).

Figure 2.2 on page 37 shows two involved graphs having vertex labels represented by the colors, grey and white. The edge labels are represented by their trait, solid or dashed. The query graph occurs twice in the target graph of the example.

A first match is given by $M^1 = ((q_4, t_0), (q_0, t_1), (q_1, t_4), (q_3, t_2), (q_2, t_5))$, and the second occurrence is obtained by exchanging t_2 with t_4, and thus obtaining $M^2 = ((q_4, t_0), (q_0, t_1), (q_1, t_2), (q_3, t_4), (q_2, t_5))$. The set of mappings of a query graph within a target graph is referred to as $\mathbb{M} = \{M^1, M^2\}$.

Given a partial solution $M_i = ((q_1, t_1), (q_2, t_2), \ldots, (q_i, t_i))$, with $i < |V_q|$, the following constraints are verified: (i) $\forall j < i \Rightarrow t_i \neq t_j$; (ii) $\alpha(q_i) = \alpha(t_i)$; (iii) $\forall \{q_i, q_j\} \in E_q \ s.t. \ j < i \Rightarrow \{M(q_i), M(q_j)\} \in E_t$, and if the graph is directed $\forall (q_j, q_i) \in E_q \ s.t. \ j < i \Rightarrow (M(q_j), M(q_i)) \in E_t$. Because we impose an ordering in M_i, subscript indexes represent the order of a given element in M_i.

Variable ordering

The performance of SubGI algorithms depends to a large extent on the order in which query vertices are included in the mapping.

Definition 6 (Ordering)**.** Formally, given a query graph $G_q = (V_q, E_q)$, an ordering $\theta = (v_1, v_2, \ldots, v_{|V_q|}) : v_i \in V_q$ is a sequence of the vertices in V_q.

Definition 7 (Partial order)**.** Given the order θ, a partial order $\theta_i = (v_1, v_2, \ldots, v_i)$ is defined as the first i vertices of θ.

The order can be chosen statically (before the search process begins) or dynamically by exploiting the information in the current partial solution [20]. Each approach has advantages for certain graphs and disadvantages for others.

Reference [21] introduced a static variable ordering strategy. It is based on the most-constrained fail-first principle. The vertex that is most constrained is the one that is likely to cause a constraint failure. Constraints may be semantic (i.e. vertex label) or topological (i.e. the vertex degree, and edges). Intuitively, the vertex with the highest number of constraints is the one with the highest pruning power. Constraints come in the form of the edges that link a possible next vertex to other vertices that are already in a partial mapping. The more vertices that are already in the ordering and are linked to the next vertex, the more constraining the next vertex will be. Moreover, since multiple vertices with the same constraint value can exist, a tie-breaking rule is adopted. Such a strategy has been shown empirically to be an excellent strategy [135, 89, 20, 5].

Vertex domains and vertex features

One way to improve the performance of SubGI solvers is to try to filter out some of the assigned values before the backtracking search process begins. For example, for a given query vertex q_i only those elements having the same label as q_i should be considered, rather than trying to assign all target vertices to it. Thus, for each query vertex q_i, a domain $D(q_i)$ is defined. Each domain contains the target vertices that are compatible with the corresponding query

vertex. Compatibility is assessed based on the label, but also on the degree. Thus, the degree of the vertices in $D(q_i)$ must be greater than or equal to $deg(q_i)$. Compatibility can also be assessed by degree probability [27] or by looking at the neighbors of a vertex and their labels [150].

Even more sophisticated compatibility criteria can be exploited, for example, in [63] the directed acyclic graph (DAG) induced by a vertex of the query graph is compared to the DAG induced by vertices in the target graph. The DAG induced by a vertex is retrieved by performing a breath-first visit of the graph. In general, every kind of invariant based on substructures of the graph topology can be used as a feature to establish vertex compatibility [39, 151, 127].

Paths, trees and subgraphs are the most used feature types for describing vertices [125, 65] by extracting their neighborhood. Such features are often involved in graph indexing techniques. The aim is to build an index of the target graph that can be used to reduce the searching time. The cost for indexing the graph is amortized if multiple queries are run. Surprisingly, sometimes indexing yields advantages in single-query executions [55, 80, 79].

Domains reduction

Initial vertex domains can be refined via *reduction* procedures in which the global topology of the query is used to discard candidate portions of the target graph.

A well-known procedure is called *arc consistency* [99], which ensures that target vertices belonging to the domains of two connected query vertices must be connected too. Formally, the procedure verifies that for each query edge $\{q_i, q_j\}$, $\forall t_k \in D(q_i) \exists t_h \in D(q_j) : \{t_k, t_h\} \in E_t$. In addition, we require $\beta(\{t_k, t_h\}) = \beta(\{q_i, q_j\})$. The verification procedure is applied to a given domain $D(q_i)$ in order to discard target vertices in that domain that violate arc consistency. The removal of a target vertex may influence the state of

other domains, previously reduced. Thus, it is necessary to run multiple iterations of the procedure. The number of iterations can either be bounded or the algorithm can be executed until convergence. Convergence is reached when a run produces no reduction of the current domains.

Edge domains and domain graph

The concept of vertex domain naturally extends to edges. Given two query vertices, q_i and q_j, their domains $D(q_i)$ and $D(q_j)$, and an edge $\{q_i, q_j\} \in E_q$ linking them, the domain of such an edge is given by $D(\{q_i, q_j\}) = \{\{t_k, t_h\} \in E_t : t_k \in D(q_i), t_h \in D(q_j)\}$. Vertex domains and edge domains can be combined into a single structure, here called a *domain graph* (DG), first introduced in [64] and then refined in [17] and subsequently in [63].

An example of a domain graph is given in Figure 2.3 (c). The complex data structure is able to represent vertex domains and all the edges connecting their elements.

2.1.2 Summary of the notation for simple graphs

A summary of the notation used for simple graphs is reported in Table 2.1 on page 38.

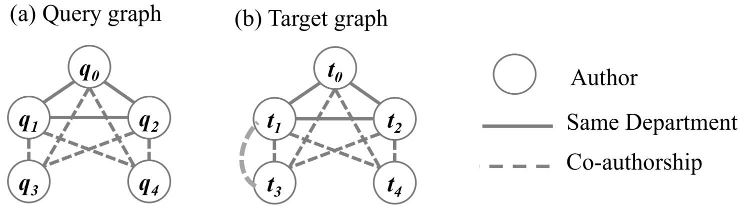


Figure 2.1: A use case example of subgraph searching over a co-authorship network.

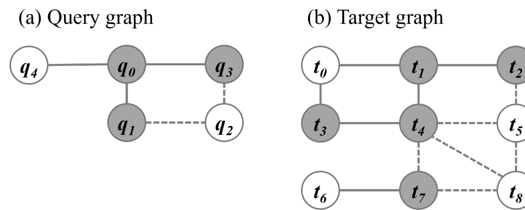


Figure 2.2: A query graph and a target graph. Each vertex is labelled white or gray. Each edge is labelled solid or dashed. The query occurs twice in the target graph.

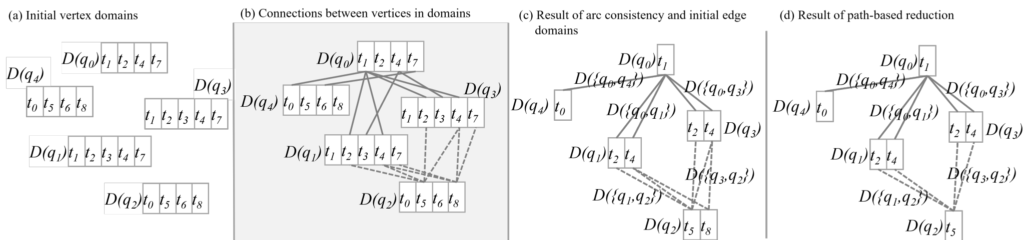


Figure 2.3: Domains, and their transformation by means of reduction techniques, obtained in solving the example of Figure 2.2.

Table 2.1: Main notation used for simple graphs.

Symbol	Definition	Description
G	$(V, E) : E \subseteq V \times V$	a graph as a set of vertices V plus a set of edges E
α	$\alpha : V \mapsto A$	a function mapping vertices to labels
β	$\beta : E \mapsto B$	a function mapping edges to labels
G_q	(V_q, E_q)	a query graph
G_t	(V_t, E_t)	a target graph
$deg(u)$	$\lvert\{(v_i, v_j) : v_i = u \text{ or } v_j = u\}\rvert$	the degree of the vertex u
$N(u)$	$\{v : (u, v) \in E \text{ or } (v, u) \in E\}$	the neighborhood of the vertex u
$N(V')$	$(\cup_{u \in V'} N(u)) \setminus V' : V' \subseteq V$	the neighborhood of the set of vertices V'
ω	$(v_1, v_2, \ldots, v_n) : v_i \in V, (v_i, v_{i+1}) \in E, \forall i : 1 \leq i \leq n-1$	a path of length $\lvert\omega\rvert = n$
ω_i	$(v_1, v_2, \ldots, v_i) : \omega = (v_1, v_2, \ldots, v_n), 1 \leq i \leq n$	the first i vertices of ω
$\omega[i]$	$v_i : \omega = (v_1, v_2, \ldots, v_n), 1 \leq i \leq n$	the i-th vertex of the path ω
$\omega[i \ldots j]$	$(v_i, v_{i+1}, \ldots, v_{j-1}, v_j) : \omega = (v_1, v_2, \ldots, v_n), 1 \leq i \leq j \leq n$	the subpath of ω from the i-th to the j-th vertex of it, both included
θ	$(v_1, v_2, \ldots, v_{\lvert V\rvert}) : v_i \in V$	an ordering of the vertices of V
θ_i	$(v_1, v_2, \ldots, v_i) : \theta = (v_1, v_2, \ldots, v_{\lvert V\rvert}), 1 \leq i \leq n$	a partial ordering, namely the first i vertices of θ
$\theta[i]$	$v_i : \theta = (v_1, v_2, \ldots, v_{\lvert V\rvert}), 1 \leq i \leq n$	the i-th vertex of θ
(q_i, t_i)	$q_i \in V_q, t_i \in V_t$	a matching pair
M	$M : V_q \mapsto V_t$	a SubGI mapping of G_q into G_t as an injective function
M	$((q_1, t_1), (q_2, t_2), \ldots, (q_n, t_n))$	a SubGI mapping of G_q into G_t as an ordered set of matching pairs
M_i	$((q_1, t_1), (q_2, t_2), \ldots, (q_i, t_i)) : 1 \leq i \leq n$	the first i-th pairs of the mapping M
$\mathbb{M}$	$\{M^1, M^2, \ldots, M^k\}, \lvert\mathbb{M}\rvert = k$	the entire set of matches between a query G_q and a target graph G_t
$D(q_i)$	$\{t_h \in V_t : t_h \simeq q_i\}$	the domain of the vertex q_i
$D((q_i, q_j))$	$\{(t_h, t_k) \in E_t : t_h \in D(q_i), t_k \in D(q_j), \beta((t_h, t_k)) = \beta((q_i, q_j))\}$	the domain of the edge (q_i, q_j)

2.2 Multigraphs

Definition 8. A multigraph is a tuple $G = (V, E, \mathcal{L}, \sigma, \mathcal{T}, \pi, \mathcal{A}, \mathcal{B})$, where:

- V is the set of nodes, each of which has a unique identifier;
- E is the set of edges. Each edge $e \in E$ has a unique identifier and is associated with a pair (u, v), with $u, v \in V$. Two or more edges with different identifiers may be associated with the same pair of nodes;
- $\mathcal{L}$ is the set of node labels;
- $\sigma : V \to \mathcal{L}^+$ is the node label function, which associates one or more distinct labels from $\mathcal{L}$ to each node;
- $\mathcal{T}$ is the set of edge types;
- $\pi : E \to \mathcal{T}$ is the edge type function, which associates one type from $\mathcal{T}$ to each edge;
- $\mathcal{A} : N \to P_N \times V_N$ is the node property function, which associates to each node a set of property-value pairs $\{(a_1, x_1), ..., (a_l, x_l)\}$;
- $\mathcal{B} : E \to P_E \times V_E$ is the edge property function, which associates to each edge a set of property-value pairs $\{(b_1, y_1), ..., (b_m, y_m)\}$;

Based on this definition, a multigraph is a graph with (potentially) multiple edges between two nodes, in which nodes and edges are annotated with labels and types, respectively, denoting classes they belong to. In addition, both nodes and edges are associated with properties having specific values. The definition of multigraph presented here is a generalization of the property graph [3], though the latter is not a multigraph.

We denote with $id(u)$ the identifier of node u. Similarly, we denote with $id(e)$ the identifier of edge e. Given an edge $e = (u, v)$, u is the *source* of e,

v is the *destination* of e, and u and v are the *endpoints* of e. If $\forall(u, v) \in E$, $(v, u) \in E$, then G is *undirected*, otherwise G is *directed*. In directed graphs, where edges have a specific direction, the *out-degree* of a node u, $outDeg(u)$, is the number of edges having u as the source, while the *in-degree* of u, $inDeg(u)$, is the number of edges having u as the destination. In undirected graphs, where edges have no orientation, these concepts coincide, and the degree of a node is simply the total number of incident edges. Given an edge type t, the *t-dependent out-degree (in-degree)* of u, $t - outDeg(u)$ ($t - inDeg(u)$), is the number of edges of type t having u as source (destination).

Given a node u, we denote as $\mathcal{A}(u) = \{(a_1, x_1), ..., (a_l, x_l)\}$ the *property set* of u. Likewise, given an edge e of G, we denote as $\mathcal{B}(e) = \{(b_1, y_1), ..., (b_m, y_m)\}$ the property set of e.

An example of a multigraph is depicted in Figure 2.4 on page 42, where nodes are actors and directors, two actors are linked by bidirectional edges iff they acted together in a movie and a directed edge connects a director D to an actor A iff D directed A in a movie. Node labels represent the main professions of a movie star ('actor' and/or 'director'), while edge labels denote the type of the relationship between two nodes ('directed' or 'acted_with'). Node and edge properties represent, respectively, name and sex of a movie star and name and year of production of a movie referenced by a specific edge.

2.2.1 SubMultigraph Matching (SMM)

Definition 9 (SubMultigraph Matching (SMM))**.** Given two multigraphs $Q = (V_Q, E_Q, \mathcal{L}_Q, \sigma_Q, \mathcal{T}_Q, \pi_Q, \mathcal{A}_Q, \mathcal{B}_Q)$ and $T = (V_T, E_T, \mathcal{L}_T, \sigma_T, \mathcal{T}_T, \pi_T, \mathcal{A}_T, \mathcal{B}_T)$, called query and target, respectively, the SubMultigraph Matching (SMM) problem consists in finding an injective function $f : V_Q \to V_T$, called node mapping, and an injective function $g : E_Q \to E_T$, called edge mapping, such that the following conditions hold:

1. $\forall\, e = (u, v) \in E_Q,\ g(e) = (f(u), f(v));$
2. $\forall\, u \in V_Q,\ \sigma(u) \subseteq \sigma(f(u));$
3. $\forall\, e \in E_Q,\ \pi(e) = \pi(g(e));$
4. $\forall\, u \in V_Q,\ \forall\, (p : v) \in \mathcal{A}_Q(u)\ (p : v) \in \mathcal{A}_T(f(u));$
5. $\forall\, e \in E_Q,\ \forall\, (p : v) \in \mathcal{B}_Q(e)\ (p : v) \in \mathcal{B}_T(g(e));$

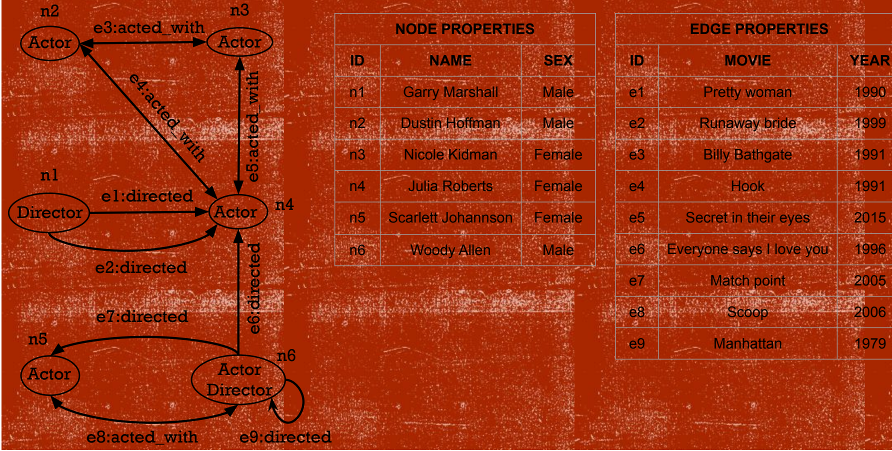
Condition 1 ensures that edge mapping g is consistent with node mapping f. Condition 2 states that all the labels of a query node must be present among the labels of the corresponding mapped target node. Likewise, condition 3 states that mapped query and target edges must have the same type. Condition 4 implies that all property-value pairs of each query node be also present in the mapped target node. A similar constraint is expressed by condition 5 for the property set of mapped query and target edges.

Definition 10 (Occurrence)**.** Given an edge mapping g, an *occurrence* of Q in T is a graph O formed by edges $g(e_1), g(e_2), ..., g(e_k)$ where all nodes that are sources or destinations of at least one of these edges.

Figure 2.5 shows an example of application of the SMM problem, where a possible node mapping is represented by black dashed lines and a possible edge mapping is depicted by orange dotted lines. The occurrence associated to these mappings is the subgraph of T formed by nodes t_1, t_3, t_4 and t_5 and edges d, f, b and h.

2.2.2 Summary of the notation for multigraphs

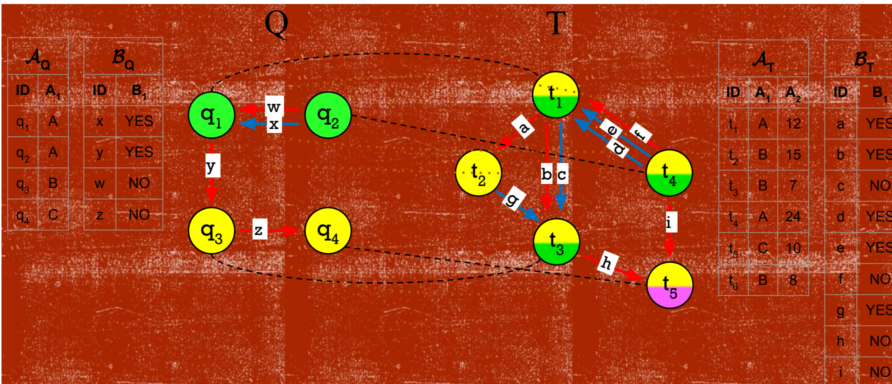
A summary of the notation used for multigraphs is reported in Table 2.2 on page 43.



NODE PROPERTIES		
ID	NAME	SEX
n1	Garry Marshall	Male
n2	Dustin Hoffman	Male
n3	Nicole Kidman	Female
n4	Julia Roberts	Female
n5	Scarlett Johannson	Female
n6	Woody Allen	Male

EDGE PROPERTIES		
ID	MOVIE	YEAR
e1	Pretty woman	1990
e2	Runaway bride	1999
e3	Billy Bathgate	1991
e4	Hook	1991
e5	Secret in their eyes	2015
e6	Everyone says I love you	1996
e7	Match point	2005
e8	Scoop	2006
e9	Manhattan	1979

Figure 2.4: Example of multigraph with actors and directors of the movie industry.



$\mathcal{A}_Q$	
ID	A_1
q_1	A
q_2	A
q_3	B
q_4	C

$\mathcal{B}_Q$	
ID	B_1
x	YES
y	YES
w	NO
z	NO

$\mathcal{A}_T$		
ID	A_1	A_2
t_1	A	12
t_2	B	15
t_3	B	7
t_4	A	24
t_5	C	10
t_6	B	8

$\mathcal{B}_T$	
ID	B_1
a	YES
b	YES
c	NO
d	YES
e	YES
f	NO
g	YES
h	NO
i	NO

Figure 2.5: Toy example of SubMultigraph Matching (SMM) with a query Q and a target T. Colors in nodes and edges represent different labels or types. Target nodes t_1, t_3, t_4 and t_5 have two labels. Black dashed lines denote a possible node mapping, with query nodes q_1, q_2, q_3 and q_4 mapped to target nodes t_1, t_4, t_3 and t_5, respectively. Orange dotted lines depict a possible edge mapping, with query edges w, x, y and z mapped to target edges f, d, b and h, respectively.

Table 2.2: Main notation used for multigraphs.

Symbol	Definition	Description
G	$(V, E, \mathcal{L}, \sigma, \mathcal{T}, \pi, \mathcal{A}, \mathcal{B})$	a multigraph with nodes, edges, labels, types, and properties
V	Set of vertices	the nodes in the multigraph
E	Set of edges	the edges between nodes in the multigraph
$\mathcal{L}$	Set of node labels	the labels that can be assigned to vertices
σ	$\sigma : V \mapsto \mathcal{L}^{+}$	function that assigns labels to nodes
$\mathcal{T}$	Set of edge types	the types of edges in the multigraph
π	$\pi : E \mapsto \mathcal{T}$	function that assigns types to edges
$\mathcal{A}(u)$	Property set of a node	the properties and values assigned to a node u
$\mathcal{B}(e)$	Property set of an edge	the properties and values assigned to an edge e
f	$f : V_Q \rightarrow V_T$	node mapping in SubMultigraph Matching
g	$g : E_Q \rightarrow E_T$	edge mapping in SubMultigraph Matching

2.3 Temporal Graphs

Definition 11 (Temporal graph)**.** A *temporal graph* (or network) G is a pair of sets (V, E), where V is the set of nodes and $E \subseteq V \times V \times \mathbb{R}$ is the set of edges. Each edge is a triple (s, d, t) where s is the source node, d is the destination node and t is the timestamp, denoting the moment or the time interval in which the two nodes interact.

By definition, a temporal graph is a multigraph, because there can be multiple edges between two nodes. However, triplets in E are distinct, therefore such edges need to have different timestamps. With $e.source$, $e.dest$ and $e.time$ we denote the source, the destination and the timestamp of edge e, respectively. A temporal graph $G = (V, E)$ is *undirected* if $\forall(s, d, t) \in E(G)$ we have $(d, s, t) \in E(G)$, otherwise it is *directed*. With $Inc(v)$ we denote the set of all edges that are incident to node v, i.e. having v as source or destination.

2.3.1 Temporal Subgraph Isomorphism (TSI)

Definition 12 (Temporal Subgraph Isomorphism (TSI))**.** Given two temporal graphs $Q = (V_Q, E_Q)$ and $T = (V_T, E_T)$, called, respectively, *query* and *target*, and an integer Δ, the *Temporal Subgraph Isomorphism* (TSI) problem consists in finding an injective function $f : V_Q \to V_T$, called *node mapping*, and an injective function $g : E_Q \to E_T$, called *edge mapping*, such that the following conditions hold:

1. $\forall e_Q = (u, v, t_Q) \in E_Q \; \exists \, e_T \in E_T$ s.t. $e_T = g(e_Q) = (f(u), f(v), t_T)$;

2. $\forall e_Q, e'_Q \in E_Q$ s.t. $e_Q.time \leq e'_Q.time \Rightarrow g(e_Q).time \leq g(e'_Q).time$;

3. $\forall e_Q, e'_Q \in E_Q \; |g(e_Q).time - g(e'_Q).time| \leq \Delta$.

The first condition ensures that the edge mapping is consistent with the node mapping. The second condition requires that the chronological order

between query edges is respected in the target network. The third condition imposes that all matching target edges are observed within a fixed time interval Δ.

The TSI problem can have one or more solutions. In this case, we say that Q Δ-*occurs* in T. Given an edge mapping g between Q and T, a Δ-*occurrence* of Q in T is a temporal graph S formed by edges $g(q_1), g(q_2), ..., g(q_k)$ and all nodes that are sources or destinations of at least one of these edges.

In Figure 1.1 on page 24, given $\Delta = 6$, query Q_1 Δ-occurs in target T and the corresponding occurrence is the subgraph of T highlighted in red. Query Q_2, instead, has no Δ-occurrences in T. Indeed, there is only one subgraph of T (highlighted in blue) that is isomorphic to Q_2 but violates the Δ constraint on edge timestamps.

2.3.2 Temporal Motif Search (TMS)

Definition 13 (Temporal motif)**.** Let $Q = (V, E)$ a connected temporal graph with l edges and $(t_1, t_2, ..., t_l)$ the sequence of Q's edges timestamps in ascending order. Q is a *temporal motif* iff: i) $t_1 = 1$, ii) $t_{i+1} - t_i \leq 1 \; \forall \; 1 \leq i \leq l - 1$.

In other words, a temporal motif Q can be considered a sort of standardized temporal graph, in which edge timestamps denote an order in which events happen starting from the initial event (i.e. event 1). Edges with equal timestamps (if any) in Q represent simultaneous event. Examples of temporal motifs are the graphs Q_1 and Q_2 depicted in Figure 1.1.

Definition 14 (Temporal Motif Search (TMS))**.** Given a temporal graph $T = (V_T, E_T)$ and three integers k, l and Δ, the Temporal Motif Search (TMS) problem consists in: i) retrieving all temporal motifs that Δ-occurs in T and have at most k nodes and l edges, ii) counting the number of Δ-occurrences of such motifs.

An example of application of the TMS problem is shown in Figure 1.2 on page 24 where $k = 3$, $l = 3$ and $\Delta = 10$.

2.3.3 Summary of the notation for temporal graphs

A summary of the notation used for temporal graphs is reported in Table 2.3 on the following page.

Table 2.3: Main notation used for temporal graphs.

Symbol	Definition	Description
G	(V, E)	a temporal graph with vertices and edges
E	Set of edges	the edges with time information
(u, v, t)	Edge with time t	an interaction between u and v at time t
Δ	Time window	the time interval for motif search
M	Motif	a small repeated pattern in a temporal graph
f	$f : V_Q \to V_T$	node mapping in Temporal Subgraph Isomorphism
g	$g : E_Q \to E_T$	edge mapping in Temporal Subgraph Isomorphism

2.4 Multiplex Graphs

We can define multiplex graphs as a graphs composed by two different sub-networks, which we refer to as horizontal and vertical. The horizontal network consists of a collection of simple graphs, called (horizontal) layers. In this setup, each unit of a system can be represented as a node in one or more layers, with each layer referring to a certain type of relationship. We refer to the connections within a given layer as intra-layer links. The second sub-network, namely the vertical network, consists of a single-layer graph formed by the set of edges connecting nodes across different layers. We assume that a node i on a layer α can be connected to at most one node j on another layer β, i.e., the two nodes represent the same unit of the system. Also, we assume that if node i on layer α is connected to node j on layer β, and if node j is connected to node k on layer γ, then nodes i and k are also connected. Under these hypotheses, the vertical network consists in a collection of connected components, which can be either cliques or isolated nodes. The edges in the vertical network will be referred to as inter-layer links.

Definition 15 (Multiplex Graph)**.** Formally, a multiplex graph is a set $\mathbb{V}$ of N nodes that are connected through L different layers. We assume that N_α nodes are present in each layer $\alpha \in \{0, 1, \dots, L\}$, such that $N_1 + \dots + N_L = N$. Each layer α is a graph $\mathcal{G}_\alpha = (\mathbb{V}_\alpha, \mathcal{E}_\alpha)$ where $\mathbb{V}_\alpha \in \mathbb{V}$ is the set of the N_α nodes and $\mathcal{E}_\alpha$ is the set of edges connecting them. The node sets for each layer are disjoint, i.e., $\mathbb{V}_\alpha \cap \mathbb{V}_\beta = \emptyset$ for $\alpha \neq \beta$, and their union makes the set of all nodes $\mathbb{V}$. The set of intra-layer links for the horizontal network can be defined as $\mathcal{E}_{\text{intra}} = \bigcup_{\alpha=1}^{L} \mathcal{E}_\alpha$. Hence, we define the horizontal network as $\mathcal{G}_H = (\mathbb{V}, \mathcal{E}_{\text{intra}})$. The vertical network can be instead defined as $\mathcal{G}_V = (\mathbb{V}, \mathcal{E}_{\text{inter}})$, where $\mathcal{E}_{\text{inter}} = \Big\{(i,j) \in \mathbb{V} \times \mathbb{V} \mid \exists \alpha, \beta \in \{0, 1, \dots, L\} \; s.t. \; (i,j) \in \mathcal{E}_\alpha \times \mathcal{E}_\beta\Big\}$ is the set of inter-layer edges. Figure 2.6 on page 50 shows an illustration of a multiplex graph.

2.4.1 Multiplex Graph Embedding Framework

In the context of multiplex graphs, embedding refers to generating a low-dimensional representation of nodes that captures both intra-layer and inter-layer relationships. This embedding framework is critical for tasks such as link prediction and node classification, where interactions across different layers need to be considered.

Definition 16 (Multiplex Graph Embedding)**.** The *Multiplex Graph Embedding* problem consists of generating a representation for each node $v \in \mathbb{V}$, considering both its intra-layer and inter-layer connections. The embedding is designed to preserve the structure and relationships of the nodes across all layers.

The embedding process involves creating a separate representation for each layer (horizontal embedding) and then combining them with the inter-layer connections (vertical embedding) to form a comprehensive representation of the node.

2.4.2 Summary of the notation for multiplex graphs

A summary of the notation used for multiplex graphs is reported in Table 2.4 on page 51.

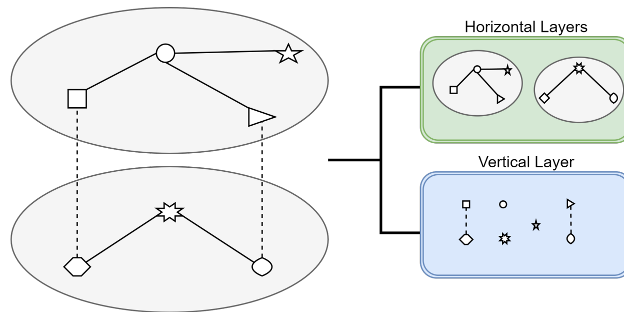


Figure 2.6: A toy example of a multiplex network with 2 horizontal layers. The solid edges represent the intra-layer connections while the dashed edges are the inter-layer edges.

Table 2.4: Main notation used for multiplex graphs. Note that superscripts $\cdot^H$ and $\cdot^V$ are omitted for the sake of readability.

Symbol	Definition	Description
$N \in \mathbb{N}$	Number of nodes	Number of nodes in the multiplex graph
$\mathbb{V}$	Set of nodes	Set of nodes in the multiplex graph
$L \in \mathbb{N}$	Total number of layers	Total number of horizontal layers of the multiplex graph
$\alpha \in \mathbb{N}$	Layer index	Index of the horizontal layer in the multiplex graph
N_α	Number of nodes in layer α	Number of nodes within the horizontal layer of index α
$\mathbb{V}_\alpha$	Set of nodes in layer α	Set of nodes within the horizontal layer of index α
$\mathcal{E}_\alpha$	Set of intra-layer edges	Set of edges within the horizontal layer α
$\mathcal{G}_\alpha$	Horizontal layer α	Layer of index α in the horizontal network
$\mathcal{E}_{\text{intra}}$	Set of intra-layer edges	Set of intra-layer edges for the horizontal network
$\mathcal{E}_{\text{inter}}$	Set of inter-layer edges	Set of inter-layer edges for the horizontal network
$\mathcal{G}_H$	Horizontal network	Horizontal network of the multiplex graph
$\mathcal{G}_V$	Vertical network	Vertical network of the multiplex graph
$F \in \mathbb{N}$	Initial feature dimensionality	Initial node feature dimensionality
$F' \in \mathbb{N}$	Final feature dimensionality	Final node feature dimensionality
$i \in \mathbb{N}$	Source node	Source node in an edge
$j \in \mathbb{N}$	Destination node	Destination node in an edge
$k \in \mathbb{N}$	Source node's layer index	Source node horizontal layer's index
$q \in \mathbb{N}$	Destination node's layer index	Destination node horizontal layer's index
$\mathcal{N}_i$	Neighbors of node i	Set of nodes connected to node i
$\boldsymbol{h}_i \in \mathbb{R}^F$	Node embedding	Embedding of node i
$\boldsymbol{v} \in \mathbb{R}^{F'}$	Attention weight vector	Attention weight vector for the model
$\boldsymbol{W} \in \mathbb{R}^{F' \times F}$	Weight matrix	Weight matrix used to linearly transform the node embeddings
$e_{i,j} \in \mathbb{R}$	Attention coefficient	Attention coefficient between node i and j
$\alpha_{i,j} \in \mathbb{R}$	Normalized attention coefficient	Normalized attention coefficient between nodes i and j
$\boldsymbol{h}_i \in \mathbb{R}^{F'}$	Updated node embedding	Updated embedding of node i after a forward pass
σ	Activation function	Activation function, e.g., LeakyReLU
f	Embedding transformation function	Function used to transform the horizontal embeddings to match the vertical embedding space
g	Combination function	Function used to combine vertical and horizontal embeddings

3

ArcMatch
High-performance subgraph matching for labeled graphs by exploiting edge domains

This chapter presents a novel approach, called *ArcMatch* which works on graphs having labels on vertices and/or edges. It solves subgraph isomorphism and it returns the matching mappings between the query and the target graph, as well as the count of such mappings. *ArcMatch* introduces the following new methodologies for optimizing reduction techniques [22].

- *ArcMatch* filters using vertex domains before filtering using edge domains. This reduces memory consumption.
- A further filtering technique, called path-based domain reduction, verifies the correspondence between paths in the query and the target graph.
- The ordering defined in [21] is extended in order to exploit the information on domains.
- The search process is cost-based where the cost is assumed proportional to the size of edge domains. Moreover, it is equipped with a dynamic parent selection that explores the search space according to the effective number of candidate vertices for a given partial mapping.

Path-based domain reduction including their theoretical properties as well as the cost-based search process constitute the main novel algorithmic features of the present work. In addition, the pipeline itself is new and contributes to the high performance of *ArcMatch*.

Tests conducted on real graphs, where both vertices and edges are labeled, demonstrate that our approach exhibits notably faster performance compared to existing state-of-the-art tools. This is especially pronounced when dealing with an increasing number of target and query labels. A *light* version of *ArcMatch*, called *ArcMatch-lt* (in which path reduction is disabled) is particularly competitive when the results are capped at a certain number of mappings.

The novel techniques of *ArcMatch* are introduced in Section 3.1. Section 3.2 describes the experiments to assess the performance of *ArcMatch* as compared with the state-of-the-art. The section also reports a study regarding the scalability of the compared approaches on varying topological and labeling properties of the involved graphs. Section 3.3 provides a detailed comparison of the techniques involved in *ArcMatch*, analyzing the contributions of each feature and how they perform in combination under different experimental conditions. Furthermore, Section 3.4 explores the relationship between arc consistency and the proposed path-based reduction techniques, highlighting their differences and examining the impact of each on the overall performance. Lastly, Section 3.5 concludes the chapter.

My contribution to this work involved conducting extensive tests to evaluate the algorithm's performance under different conditions and writing parts of the article.

3.1 Methods

The main steps of the proposed methodology for scanning all the occurrence of a query graph G_q over a target graph G_t are:

1. compute initial vertex domains
2. run arc consistency over vertex domains
3. compute initial edge domains
4. run path-based reduction procedure
5. choose a static variable ordering
6. run the backtracking phase to locate matching occurrences.

Step (1) is performed by comparing the labels and degrees of query and target vertices such that, for a query node q_i, $D(q_i) = \{t \in V_t : \alpha(q_i) = \alpha(t), deg(q_i) \leq deg(t)\}$. Arc consistency in step (2) is performed as described in Section 2.1.1, and it can be run until convergence or not. Initial edge domains, in step (3), are computed as described in Section 2.1.1. Thus, given an edge $\{q_i, q_j\}$, its domain is computed as $D(\{q_i, q_j\}) = \{\{t_k, t_h\} \in E_t : t_k \in D(q_i), t_h \in D(q_j)\}$. The domain graph is composed of vertex and edge domains (defined in Section 2.1.1). In step (4), a novel path-based reduction (presented in Section 3.1.1) is applied to the domain graph. The search phase is driven by a new static ordering (step (5)) which combines domain cardinality with the filtering power of a vertex, presented in Section 2.1.1. The backtracking SubGI searching phase (step (6)) is completely driven by the domain graph, and it is described in Section 3.1.3. Candidates of the next vertex to be mapped are chosen by selecting an already mapped vertex, called *parent*. A novel procedure for its selection is given. Furthermore, the search is equipped with a novel *ad hoc* procedure for managing a particular type of

query vertices, called *peripheral*, which is introduced in Section 3.1.2. Thus, steps 4, 5, and 6 constitute the main algorithmic novelty of the proposed approach.

Algorithm 1 on the next page is the pseudo-code of the overall approach. It solves the SubGI problem by calling at line 13 the function that is defined in Algorithm 8 on page 75, `SearchOccurrences`.

3.1.1 Path-based domain reduction

Paths represent a good compromise between complexity and efficacy when used as graph and/or vertex features in graph indexing approaches for SubGI [23, 55]. Thus, we decided to exploit this concept in implementing a path-based reduction procedure, here simply called path reduction, to discard elements in the domain graph.

The path reduction procedure is performed after arc consistency has been run on vertex domains until convergence. Initial edge domains are extracted as described in Section 2.1.1. Then, the reduction is run based on comparing the paths that start from a given query vertex with the paths starting from target vertices. *ArcMatch* compares query paths to target paths as follows. For each target vertex t_h in the domain $D(q_i)$, the reduction verifies that for each path (up to a given length) starting from q_i, at least one corresponding path starts from t_h. The correspondence is verified by looking at vertex and edge domains. Formally, given a path $(u_1, u_2, \ldots, u_n)$ such that $u_1 = q_i$ (the starting node), $u_j \in V_q$ and $\{u_{j-1}, u_j\} \in E_q$ for $2 \leq j \leq n$, a target vertex t_h is included in the domain of q_i if and only if there exists at least one path $(v_1, v_2, \ldots, v_n)$ such that $v_1 = t_h$, $v_j \in D(u_j)$ and $\{v_j, v_{j+1}\} \in D(\{u_j, u_{j+1}\})$. Since target paths are extracted from the domain graph, label compatibility is implicitly ensured. A *maximal path* is a path whose length is exactly lp or is a ring, or it cannot be extended. Formally, given a graph $G = (V, E)$ and a path length parameter lp, let $paths_G(lp)$ be the

Algorithm 1 Matching procedure

1: **procedure** MATCH($G_q = (V_q, E_q), G_t = (V_t, E_t), lp$)
2: **for** $q_i \in V_q$ **do** ▷ initial vertex domains
3: $D(q_i) \leftarrow \{v \in V_t : \alpha(q_i) = \alpha(v)$ AND $deg(q_i) \leq deg(v)\}$
4: **end for**
5: $ArcConsistency()$ ▷ arc consistency over vertex domains
6: **for** $\{q_i, q_j\} \in E_q$ **do** ▷ initial edge domains
7: $D(\{q_i, q_j\}) \leftarrow \{(x, y) \in E_t : x \in D(q_i)$ AND $y \in D(q_j)$ AND $\beta(\{x, y\}) = \beta(\{q_i, q_j\})\}$
8: **end for**
9: **for** $q_i \in V_q$ **do** ▷ path-based reduction
10: $\omega \leftarrow (q_i)$
11: $PathReduction(\omega, lp)$
12: **end for**
13: $\theta \leftarrow variable_ordering()$ ▷ variable ordering
14: **for** $u_t \in D(\theta[1])$ **do** ▷ backtracking
15: $\widehat{\theta} \leftarrow (u_t)$
16: $SearchOccurrences(\theta, \widehat{\theta})$
17: **end for**
18: **end procedure**

set of paths in G having length lp, and let $rings_G(lp)$ be the set of rings in G having length lp. Moreover, let $mpaths_G(n)$ the set of paths in G of length n such that $\forall \omega = (v_1, v_2, \ldots, v_n), \omega \in mpaths_G(n) : N(v) \setminus \{v : v \in \omega\} = \emptyset$. The set of maximal paths of G with parameter lp is defined as $maxpaths_G(lp) = paths_G(lp) \cup rings_G(lp) \cup \{mpaths_G(n) : 1 \leq n < lp\}$.

Maximal paths from a minimal length of 3 to a maximal length of 6 are taken into account. Such a length is a user-defined parameter. A path length of 6 achieves a good trade-off between computational costs and filtering power [23, 55]. Note that the consistency of paths of length 2 is already ensured by arc consistency if edge labels are taken into account during the process. Longer paths would increase the filtering power by better pruning candidate vertices and edges that do not participate in valid mappings. However, this comes at a higher computational cost in the reduction phase, as longer paths require more processing time to evaluate and compare, particularly for large graphs.

Algorithm 2 on page 64 shows a procedure for applying arc consistency until convergence. The procedure searches for edge existence between vertex domains by also verifying edge label compatibility.

Figure 2.3 (d) on page 37 shows an example of path reduction after the application of arc consistency. The figure shows that path reduction is more powerful than simple arc consistency. The target vertex t_8 does not violate arc consistency. In fact, it has at least one valid edge between vertices in the domains $D(q_1)$ and $D(q_3)$. However, the query graph (shown in Figure 2.1 on page 37) has the ring $(q_2, q_3, q_0, q_1, q_2)$. Thus, from vertex t_8 of domain $D(q_2)$, we should be able to navigate the domain graph (Figure 2.3 (c) on page 37) and to obtain a compatible path. However, such a path can not be extracted from the domain graph. The only ring that can be obtained starting form t_8 is $(t_8, t_4, t_1, t_4, t_8)$ but that would violate the *all different* constraint because t_8 appears twice. Thus, t_8 is removed form $D(q_2)$.

Figure 2.3 on page 37 also gives an example to motivate why arc consistency should be applied before path-based reduction. The intermediate step of the panel (b) of the figure is never materialized by the proposed methodology for two reasons. First, arc consistency is able to substantially reduce the domains. This can be a substantial cost because each edge domain can have up to $|E_t|$ elements. Thus, arc consistency represents a fast low-memory technique for reducing both vertex domains and edge domains.

The example of Figure 2.3 on page 37 shows the effectiveness of arc consistency. Panel (b) of the figure shows the complete initial data structure of the proposed SubGI instance. It contains a high number of candidates in both vertex and edge domains. Applying arc consistency on the initial vertex domains (panel (a) of the figure) yields an instance of edge domains as shown in panel (c) of the figure. Panel (c) of the figure is sensibly smaller than panel (b).

Algorithm 3 on page 65 implements the procedure for extracting maximal paths from the query graph. The procedure is a depth-first search (DFS) which takes as input the current visited path ω and the max path length lp. The variable ω can also be seen as the stack which drives the DFS visits. PathReduction visits only query vertices, and it is run by setting each query vertex as the source, namely $PathReduction(\omega = [v])$ for $v \in V_q$. Push and pop operations are implemented by adding a vertex to the tail of the vector $\omega[i+1] \leftarrow v$, and by removing the vertex from the tail $\omega \leftarrow \omega[1 \ldots i]$. In each recursive DFS call, the neighbors of the vertex that are at the top of the stack are scanned in order to extend the path. If a neighbor is at the bottom of the stack, then such a path is a ring. Otherwise, only vertices that are not already in the stack are visited. The assertion of *not visited* is here expressed as $v \notin \omega$. However, an array of Booleans can be used to efficiently implement the search such that the `visited` flag is activated/deactivated during the recursive process. When a valid neighbor is pushed on the stack,

it is checked and whenever a maximal path of length lp is formed, a recursive DFS call is run. After this, the neighbor is removed from the stack.

The algorithm takes a trace of paths that cannot be extended by exploiting the variable `extended`, such that each time a maximal path is obtained as an extension of the current path, or the current path is a non-expandable path, the `VerifyPath` procedure is called to verify the domains.

Once a maximal path ω is found, Algorithm 7 on page 74 verifies that each target graph vertex in the domain of the source vertex $\omega[1]$ is compatible with $\omega[1]$ If a given target vertex is not compatible with $\omega[1]$, then it is removed from the domain of $\omega[1]$.

The compatibility reduction is performed by Algorithm 5 on page 72 which extracts paths starting from the target vertex. The extraction is done recursively by a DFS visit over the target graph and according to the current state of edge domains. Given a query path ω, a corresponding target path $\widehat{\omega}$ is searched by matching vertices in ω with vertices in the corresponding domains. Edge domains guide how to go forward in the path. In extending the path from position i to position $i+1$, we take into account the two consecutive query vertices $\omega[i]$ and $\omega[i+1]$. The partial path $\omega[1 \dots i]$ is already mapped to some target vertices, and the procedure has to find a vertex to which to map $\omega[i+1]$. Because the two query vertices are consecutive in the path, the edge $\{\omega[i], \omega[i+1]\}$ links them. Moreover, domains are in a consistent state such that if $\{u_t, v_t\}$ in $D(\{\omega[i], \omega[i+1]\})$ then $u_t \in D(\omega[i])$ and $v_t \in D(\omega[i+1])$. Thus, a vertex for $\omega[i+1]$ can be extracted from the edges in $D(\{\omega[i], \omega[i+1]\})$. The procedure also verifies that each extension does not contain duplicate vertices, except in the case of rings. In a ring the first and the last vertex are equal. The procedure checks that constraint separately.

The reason why we decided to perform Algorithm 3 (page 65) before Algorithm 7 (page 74) is to avoid an exponential explosion in memory re-

quirements. The reason is that for each query path, an exponential number of paths could match within the target graph. Thus, if the DFS visit over the query graph is performed together with the multiple DFS visits over the target graph, an exponential number of parallel DFSs must be run and stored at the same time.

Similarly to arc consistency, the removal of one candidate in a domain may affect the consistency of the other domains. Thus, a consistency check procedure must be applied. The procedure is described in Algorithm 6 on page 73. It first checks the edge domains linked to the query vertex whose domains have been changed. Then, the effects of such a reduction are propagated in an iterative way. At each iteration, the edge domain consistency is verified. Subsequently, vertices that are inconsistent with edge domains are removed.

Section 3.3 further investigates the relation between arc consistency and the proposed path-based reduction.

Relation among different values of lp.

Theorem 1. *Given two different values of path length, lp_1 and lp_2, such that $lp_1 \leq lp_2$, if a target vertex is discarded from a given domain by using lp_1 then it is also discarded by using lp_2.*

Proof. Given a query path $\omega = (q_1, q_2, \cdots, q_n)$ such that $n = lp_2$, the verification of ω entails the verification of each prefix of the path. Moreover, all the sets of paths that are scanned for lp_1 are scanned for lp_2 because paths of length lp_2 includes all the path of length $lp_1 < lp_2$.
Thus, when evaluating a path of length $n = lp_2$, the procedure `VerifyPath` will discard at least the vertices that `VerifyPath` discards when evaluating the smaller path length lp_1. □

Let $R_{FC}(G_q, G_t, lp)$ be the set of removal operations that are performed

by a single run of `PathReduction` for a given value of lp and without propagation, namely by discarding the call to the procedure `RefineDomains`.

Theorem 2. *Given two different values of path length, lp_1 and lp_2, such that $lp_1 < lp_2$, $R_{FC}(G_q, G_t, lp_1) \subseteq R_{FC}(G_q, G_t, lp_2)$.*

Proof. According to Theorem 1, given a query path $\omega = (q_1, q_2, \ldots, q_n)$ of length $n = lp_2$, the procedure `VerifyPath` includes the verification of ω_i such that $1 \leq i < n$. For $i = n-1$, the extension of a target path $\widehat{\omega_{n-1}}$ to a path $\widehat{\omega_n}$ is performed by the procedure `VerifyPathDFS`. If no valid extension is found, then the procedure stops with a negative result. This means that a path that is consistent until length $n-1$ may be inconsistent at length n. Thus, at length n a further set of removals is produced. Such a consideration can be applied inductively from $n-2$ to $n-1$, and in general from i to $i-1$ with $1 < i < n$. Thus, for any $i = lp_1$ such that $1 < i < lp_2$, $R_{FC}(G_q, G_t, lp_1) \subseteq R_{FC}(G_q, G_t, lp_2)$. □

Safety of the path-based reduction

In what follows, we show the safety of *PathReduction*. Namely, the procedure does not discard from the domains any vertex or edge that is included in at least one match between the query and the target graph. Given a query graph G_q, we let $\mathbb{M}$ be the set of matches between G_q and a target graph G_t. In order to ensure the safety of the procedure, we must ensure that the reduced domains are safe for $\mathbb{M}$. Safety means that $\forall M \in \mathbb{M}, \forall v_q \in V_q \Rightarrow M(v_q) \in D(v_q)$ AND $\forall \{q_i, q_j\} \in E_q \Rightarrow \{M(v_i), M(v_j)\} \in D(\{v_i, v_j\})$.

Theorem 3. *Given a query graph G_q, a target graph G_t, their corresponding set of matches $\mathbb{M}$, and a state of vertex and edge domains that are safe for $\mathbb{M}$, then `PathReduction` alters such domains such that they are still safe for $\mathbb{M}$.*

Proof. Given a query graph G_q, let Ω_q to be the set of paths of G_q, from length 1 to the maximum path length in G_q. `PathReduction` visits a subset $\Omega'_q \subseteq \Omega_q$ of paths, depending on the *lp* parameter. Let v_t be a target vertex in the domain of v_q such that $\exists M \in \mathbb{M} : M(v_q) = v_t$. For each path $\omega_q \in \Omega'_q$ such that $\omega_q[1] = v_q$, there exists a corresponding path ω_t such that $\omega_q[i] = M(\omega_q[i]) = \omega_t[i]$ for $2 \leq i \leq |\omega_q|$. If we suppose that all the domains, except $D(v_q)$, are safe, then v_t can not be removed from $D(v_q)$ because for each path ω_q starting from v_q, $M(\omega_q[i])$ is in $D(\omega_q[i])$. The reduction of edge domains is performed by the procedure `RefineDomains`. Since we suppose the safety of initial domains, `RefineDomains` cannot remove any target edge $\{t_i, t_j\}$ for which a corresponding mapping to $\{v_i, v_j\}$ exists in $\mathbb{M}$. This last assertion comes from the fact that a target edge is removed from an edge domain only after one removal from a vertex domain. Therefore, the cause for removing the edge $\{t_i, t_j\}$ from $D(\{v_i, v_j\})$ is that t_i has been removed from $D(v_i)$ or t_j has been removed from $D(v_j)$. Thus, the safety of the removal of an edge from a domain is ensured by the safety of the removal of one of its vertices from the corresponding domain. Thus, the overall path reduction procedure alters domains such that they are still safe for $\mathbb{M}$. □

3.1.2 Vertex ordering strategy

ArcMatch adopts a modified version of the strategy proposed in [21] for defining a matching ordering among query vertices. Given a partial ordering θ_i, the node v to be chosen next in the ordering is selected among the neighbors of θ_i. Then, the neighborhood of v is divided into three sets:

- $\mathbb{N}_1(v) = \{u \in \theta_i : \{u, v\} \in E\}$
- $\mathbb{N}_2(v) = \{u \in N(\theta_i) : \{u, v\} \in E\}$
- $\mathbb{N}_3(v) = N(v) \setminus \{\mathbb{N}_1(v) \cup \mathbb{N}_2(v)\}$

Figure 3.1 shows an example of such a partition of the neighborhood of the vertex v.

We define five measures on a given vertex v for ordering query vertices:

- $N_1(v) = |\mathbb{N}_1(v)|$
- $N_2(v) = |\mathbb{N}_2(v)|$
- $N_3(v) = |\mathbb{N}_3(v)|$
- $N_4(v) = deg(v)$
- $N_5(v) = |D(v)|$

Thus, given a partial ordering θ_i and two query vertices, v' and v'' that are neighbors of at least one vertex in θ_i, the next vertex to be included in the ordering is chosen by comparing sequentially their N_i, for $1 \leq i \leq 5$, measures. The vertex v' with the smallest i such that $N_i(v') > N_i(v'')$, for $1 \leq i \leq 4$, is chosen. If two vertices have exactly the same values, then N_5 is taken into account such that the v' is selected if $N_5(v') < N_5(v'')$. If still no vertex can be chosen, then the one with the lowest identifier is selected.

An exception to the ordering rule is given by vertices with singleton domains. Such vertices are put before the non-singleton vertices in the ordering.

Peripheral vertices

Postponing Cartesian products of disjoint domains is a well-known heuristic for SubGI [17]. Here, we propose a methodology which combines the postponing of peripheral query vertices with an *ad hoc* procedure for matching vertices with disjoint domains.

Given a query graph $Q = (V_q, E_q)$ and a partial ordering $\theta = (q_0, q_1, \ldots, q_j)$ defined on the query vertices, a query vertex q_i is *peripheral* if $|\{q_k : \{q_i, q_k\} \in E_q\}| = 1$

Algorithm 2 Application of arc consistency until convergence.
G_q and domains are supposed to be globally defined.

1: **procedure** ARCCONSISTENCY
2: $\quad reduced \leftarrow true$
3: $\quad$ **while** *reduced* **do**
4: $\quad\quad reduced \leftarrow false$
5: $\quad\quad$ **for** $\{q_i, q_j\} \in E_q$ **do**
6: $\quad\quad\quad$ **for** $t_i \in D(q_i)$ **do**
7: $\quad\quad\quad\quad$ **if** $\nexists t_j \in D(q_j) : \{t_i, t_j\} \in E_t$ AND $\beta(\{t_i, t_j\}) = \beta(\{q_i, q_j\}))$ **then**
8: $\quad\quad\quad\quad\quad D(q_i) \leftarrow D(q_i) \setminus \{t_i\}$
9: $\quad\quad\quad\quad\quad reduced \leftarrow true$
10: $\quad\quad\quad\quad$ **end if**
11: $\quad\quad\quad$ **end for**
12: $\quad\quad$ **end for**
13: $\quad$ **end while**
14: **end procedure**

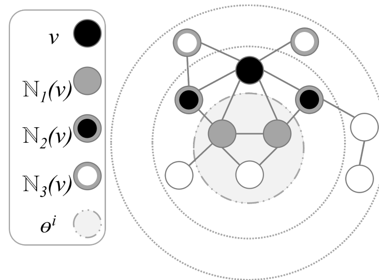


Figure 3.1: A 3-ways division of the neighborhood of a vertex v given a partial ordering θ_i. $\mathbb{N}_1(v)$ (dark nodes) represents the neighbors of v that are already included in the current partial ordering θ_i; $\mathbb{N}_2(v)$ (gray nodes) contains the neighbors of θ_i that are not yet in the ordering but are connected to v; $\mathbb{N}_3(v)$ (light nodes) includes the remaining vertices in $N(v)$ that are not adjacent to θ_i.

Algorithm 3 Path-based reduction procedure. It implements a recursive DFS over the query graph for retrieving maximal paths (up to length lp). ω is the DFS stack.

```
 1: procedure PathReduction(ω, lp)
 2:     i ← |ω|
 3:     u ← ω[i]
 4:     extended ← false
 5:     for v ∈ N(u) do
 6:         if v = ω[1] then                          ▷ It is a ring
 7:             ω[i + 1] ← v
 8:             VerifyPath(ω, true)
 9:             ω ← ω[1 . . . i]
10:         else                                      ▷ Try to extend it
11:             if v ∉ ω then
12:                 extended ← true
13:                 ω[i + 1] ← v
14:                 if i + 1 = lp then                ▷ Max-length path
15:                     VerifyPath(ω, false)
16:                 end if
17:                 PathReduction(ω, lp)
18:                 ω ← ω[1 . . . i]
19:             end if
20:         end if
21:     end for
22:     if not extended then                          ▷ Unextendable path
23:         VerifyPath(ω, false)
24:     end if
25: end procedure
```

Two vertices, q_i and q_j, have joint domains if at least one element of $D(q_i)$ is also in $D(q_j)$. We do not take into account peripheral vertices each of whose domains contains only one target vertex since they are put at the being of the ordering. Then, the set of peripheral vertices having joint domains is retrieved, and only one vertex is selected for each set. The selected vertex is the one with the biggest domain. Selected vertices are put at the end of the matching ordering.

Once a partial solution regarding all non-peripheral disjoint vertices is found, domains of peripheral disjoint vertices can be independently reduced. They have disjoint domains, thus the *all different* property does not need to be tested among them. This reduces the time requirement. Moreover, if just the counting of the subisomorphisms is required, it can be quickly computed by multiplying the size of their reduced domains.

3.1.3 Extension of partial mappings and dynamic parent selection

Algorithm 8 on page 75 implements the backtracking procedure of the search process. It recursively extends partial matches to complete solutions. In each recursive step, a parent query vertex s is chosen dynamically. Such a parent is the currently matched vertex which minimizes the number of candidates to the current state i (line 3). The minimization is obtained by counting the query edges that are in the domain $D(\{s, \theta[i]\})$. Once a parent is chosen, candidate target vertices are extracted by retrieving the subset of edges $\{u_t, v_t\} \in D(\{s, \theta[i]\})$ such that $M(s) = u_t$ (line 4). Thus, the target vertices of type $v_t \notin \widehat{\theta}$ (if not already matched) are evaluated. The evaluation is made by verifying the connectivity of each candidate with the vertices that are in the partial matching, according to the query topology (line 5-10). The verification is performed by exploiting the edge domains (line 7). If a candidate verifies the connectivity, then it is added to the partial match

in order to extend it (line 11-16). If the length of the formed partial match is actually equal to the number of query vertices, then a complete solution has been found and it is reported by the algorithm (line 13).

Please recall that the proposed algorithm is not intended to work with multi-graphs, which admit multiple edges between two vertices. The restriction to simple graphs ensures that the list of candidates does not contain duplicate vertices, thus no duplicate matches are reported. Moreover, because the `SearchOccurrences` procedure performs tail recursion, we implement it in an iterative fashion, thus eliminating the call stack overhead.

3.1.4 Safety and completeness

In what follows, we prove the safety and completeness of Algorithm 1 on page 56. Given a query G_q and a target G_t graph, and let $\mathbb{M}$ be the entire set of mappings of G_q in G_t, safety means that every match found by Algorithm 1 is in $\mathbb{M}$. Completeness means that every match in $\mathbb{M}$ is found by Algorithm 1.

A match is an assignment of values to every query variable, that is the assignment of a target vertex to every query vertex. Domains define the compatibility between query and target vertices. The initial domains contain the entire set of target vertices.

A brute force algorithm generates all possible assignments of target vertices to query vertices, and, for each assignment, verifies the constraints of the SubGI instance. It is safe because only assignments that verify constraints are returned. A brute force algorithm is complete because it generates every possible assignment, which implies that $\mathbb{M}$ is a subset of the generated assignment. By contrast, Algorithm 1 applies an initial filtering to vertex domains such that, for each query vertex, only target vertices having the same labels and compatible degrees are selected. In addition, similarly to vertex domains, an initial safe filtering is applied by checking label compatibility.

Thus, initial edge domains do not violate safety.

`ArcConsistency` is safe for $\mathbb{M}$, because Theorem 6 shows that the filtering operations executed by `ArcConsistency` are a subset of those executed by `PathReduction`, and Theorem 3 shows that `PathReduction` is safe. Therefore, `ArcConsistency` does not violate safety and completeness. Edge domains are built from a set of vertex domains that are safe for $\mathbb{M}$. Thus, Algorithm 8 on page 75 is run on a safe set of domains.

The backtracking procedure for generating the assignment, namely `SearchOccurrences`, is run after setting the first variable in the ordering to a value that is in its domain. This means that every partial mapping of length 1 does not violate safety. Thus, safety and completeness depend on Algorithm 8.

The difference between Algorithm 8 and the brute force algorithm described above is that it generates potential assignments by scanning edge domains rather than vertex domains. However, the two procedures have the same aim, which is to find target vertex combinations that satisfied the SubGI constraints.

Theorem 4. *Algorithm 8 is safe.*

Proof. Algorithm 8 is safe because for every partial assignment, and thus for every complete assignment, it verifies the *all different* constraint and all the topological constraints. The *all different* constraint is verified by requiring $v_t \notin \widehat{\theta}$. For a given position i ($1 \leq i \leq |V_q|$). The topological constraints that apply to $\theta[i]$ can be divided into three groups. Group one contains the edges that link to previous positions of θ. Those are verified by lines from 6 to 9. Group two contains the edge that link $\theta[i]$ with its parent positions, namely $\{s, \theta[i]\}$, where s is the chosen parent state. This constraint is verified because every extracted target edge $\{u_t, v_t\} \in D(\{s, \theta[i]\})$ is, by construction of $D(\{s, \theta[i]\})$, in E_t and this verifies edge label compatibility. Group three contains the edges that link to future states, and that will be

verified in such future states. At position $i = |\theta|$, only the first two groups of edges can occur because no further states exist. Vertex label compatibility is verified because $\{u_t, v_t\} \in D(\{s, \theta[i]\})$ implies $v_t \in D(\theta[i])$, which implies $\alpha(v_t) = \alpha(\theta[i])$, by construction. In conclusion, every SubGI constraint is verified, thus Algorithm 8 is safe. □

Safety implies that the set of assignments generated by the algorithm is a subset of $\mathbb{M}$. However, to prove completeness, we need to show that the assignment set is exactly $\mathbb{M}$.

Theorem 5. *Algorithm 8 is complete, if domains are safe for a given* $\mathbb{M}$.

Proof. Given a state i $(1 \leq i \leq |\theta|)$, an algorithm is complete for θ_i if: (i) it is complete for θ_{i-1}; (ii) given the set of assignments generated for θ_{i-1}, identified by $\mathbb{M}_{i-1}$, it generates all the assignments in $\mathbb{M}_i$.

For $i = 1$, the completeness of the algorithm in ensured by the safety of $D(w[1])$. For $i > 1$, let A be the set of target vertices that extend a given mapping in $\mathbb{M}_{i-1}$ to one or more mappings in $\mathbb{M}_i$. Let s be the parent state chosen by the algorithm. Let B be the set of target vertices retrieved by extracting from $D((s, \theta[i]))$ all the target edges (u_t, v_t) for which $u_t = M(\theta[s])$. That is $B = \{v_t : \{u_t, v_t\} \in D(\{s, \theta[i]\}) \text{ AND } u_t = M(s)\}$. Thus B is the set of unverified target vertices that Algorithm 8 retrieves at line 4 to extend partial solutions. Unverified means that the SubGI constraints have still to be checked for those vertices. If the algorithm is complete for θ_{i-1}, then $A \subseteq B$ because of the safety of edge domains. The verification is performed at line 4 by ensuring $v_t \notin \widehat{\theta}$, and at lines 5-10 by verifying topological constraints. The verification produces a set $B' \subseteq B$ of target vertices. $B' = A$ because of the safety of the algorithm. Then, since, by hypothesis, the algorithm is complete for θ_{i-1}, it is necessarily complete for θ_i because B' contains all and only those vertices that extend mappings of $\mathbb{M}_{i-1}$ to mappings in $\mathbb{M}_i$. When $i = |\theta|$, $\mathbb{M}_i$ equals $\mathbb{M}$, which means that the

produced mappings are exactly the complete solutions of the SubGI instance. Therefore, Algorithm 8 is complete. □

In general, the ordering is not a relevant aspect regarding safety and completeness, because matches in $\mathbb{M}$ are independent of it. However, it is important to notice that every possible ordering includes each query vertex exactly once.

Algorithm 4 Path reduction by verifying the existence of at least one path in the domain graph corresponding to the query path ω. Domains are globally defined.

1: **procedure** VERIFYPATH(ω, $isRing$)
2: $\quad reduced \leftarrow false$
3: $\quad$ **for** $v \in D(\omega[1])$ **do**
4: $\quad\quad \widehat{\omega} \leftarrow (v)$
5: $\quad\quad$ **if** not $VerifyPathDFS(\omega, \widehat{\omega}, isRing)$ **then**
6: $\quad\quad\quad D(\omega[1]) \leftarrow D(\omega[1]) \setminus \{v\}$
7: $\quad\quad\quad reduced \leftarrow true$
8: $\quad\quad$ **end if**
9: $\quad\quad$ **if** $reduced$ **then**
10: $\quad\quad\quad RefineDomains(v)$
11: $\quad\quad$ **end if**
12: $\quad$ **end for**
13: **end procedure**

Algorithm 5 Path reduction to verify the existence of at least one path $\widehat{\omega}$ in the domain graph corresponding to the query path ω.
Domains are globally defined.

```
1: procedure VerifyPathDSF(ω, ω̂, isRing)
2:     if |ω̂| = |ω| then return true
3:     else
4:         i ← |ω̂|
5:         u_q ← ω[i]
6:         v_q ← ω[i + 1]
7:         for ({u_t, v_t} ∈ D({u_q, v_q}) : u_t = ω̂[i]) do
8:             if isRing AND (|ω̂| = |ω| − 1) then
9:                 if v_t = ω̂[1]  then return true
10:                end if
11:            else
12:                if v_t ∉ ω̂  then
13:                    ω̂[i + 1] ← v_t
14:                    if VerifyPathDFS(ω, ω̂, isRing) then return true
15:                    end if
16:                    ω̂ ← ω̂[1 . . . i]
17:                end if
18:            end if
19:        end forreturn false
20:    end if
21: end procedure
```

Algorithm 6 Refinement of domains after the alteration of the domain of the query vertex s. G_q and domains are globally defined.

```
 1: procedure RefineDomains(s)
 2:     for {s, v_q} ∈ E_q do
 3:         for {u_t, v_t} ∈ D({s, v_q}) do
 4:             if u_t ∉ D(s) then
 5:                 D({s, v_q}) ← D({s, v_q}) \ {u_t, v_t}
 6:             end if
 7:         end for
 8:     end for
 9:     reduced ← true
10:     while reduced do
11:         reduced ← false
12:         for {u_q, v_q} ∈ E_q do
13:             for {u_t, v_t} ∈ D({u_q, v_q}) do
14:                 if u_t ∉ D(u_q) OR v_t ∉ D(v_q) then
15:                     D({u_q, v_q}) ← D({u_q, v_q}) \ {u_t, v_t}
16:                     reduced ← true
17:                 end if
18:             end for
19:         end for
20:         for {u_q, v_q} ∈ E_q do
21:             for u_t ∈ D(u_q) do
22:                 if ∄{x, y} ∈ D({u_q, v_q}) : x = u_t then
23:                     D(u_q) ← D(u_q) \ u_t
24:                     reduced ← true
25:                 end if
26:             end for
27:         end for
28:     end while
29: end procedure
```

Algorithm 7 Path reduction by verifying the existence of at least one path in the domain graph corresponding to the query path ω. Domains are globally defined.

1: **procedure** VERIFYPATH(ω, $isRing$)
2: $reduced \leftarrow false$
3: **for** $v \in D(\omega[1])$ **do**
4: $\widehat{\omega} \leftarrow (v)$
5: **if** not $VerifyPathDFS(\omega, \widehat{\omega}, isRing)$ **then**
6: $D(\omega[1]) \leftarrow D(\omega[1]) \setminus \{v\}$
7: $reduced \leftarrow true$
8: **end if**
9: **if** $reduced$ **then**
10: $RefineDomains(v)$
11: **end if**
12: **end for**
13: **end procedure**

Algorithm 8 Backtracking procedure for searching all the occurrences starting from a given target vertex $\widehat{\theta}[1]$ by following the variable ordering θ. Edge domains and E_q are globally defined.

```
 1: procedure SearchOccurrences(θ, θ̂)
 2:     i ← |θ̂|
 3:     s ← θ[argmin_x {{u, v} ∈ D({θ[x], θ[i]}) : 1 ≤ x < i}]
 4:     for {u_t, v_t} ∈ D({s, θ[i]}) : u_t = M(s) AND v_t ∉ θ̂ do
 5:         feasible ← true
 6:         for {θ[i], θ[j]} ∈ E_q : j < i do
 7:             if {v_t, θ̂[j]} ∉ D({θ[i], θ[j]}) then
 8:                 feasible ← false
 9:                 break
10:             end if
11:         end for
12:         if feasible then
13:             θ̂[i] ← v_t
14:             if |θ| = |θ̂| then
15:                 report match M = ((θ[i], θ̂[i]) : 1 ≤ i ≤ |θ|)
16:             else
17:                 SearchOccurrences(θ, θ̂)
18:             end if
19:             θ̂ ← θ̂[1 . . . i − 1]
20:         end if
21:     end for
22: end procedure
```

3.2 Experiments

We run a detailed evaluation of all the techniques that are explained above. After considering the results of such an evaluation, that are reported in Section 3.3, we decided to release two different versions of *ArcMatch* which exploit a different combination of the techniques described above. The standard version, simply called *ArcMatch*, fully exploits edge domain reduction. By contrast, *ArcMatch-lt* performs vertex domain reduction until convergence but does not perform edge domain reduction. Both configurations use edge domains during the search process and employ specialized techniques for peripheral query vertices (see Sections 2.1.1, 3.1.1 and 3.1.2).

Note 2. *ArcMatch* is intended to be the very general-purpose version of the proposed approach. However, some existing tools focus on solving a special case of SubGI in which they report only up to a certain predefined maximum number of matches. To compare with such systems when the predefined number is relatively small, *ArcMatch-lt* is better.

We have compared *ArcMatch* with other systems by taking into account the features and limitations of the state-of-the-art approaches RI-DS [21], DAF [63], VEQ [84], Glasgow [103]. Because RI-DS already includes vertex domains and arc-consistency, it can be used to evaluate the effectiveness of introducing edge domains and path-based reduction. DAF is a recent method that uses a combined vertex and edge domain structure. Instead of path-based reduction, it reduces domains by comparing the directed acyclic graphs induced by vertices. DAF manages graphs with labels only on vertices, not on edges. This limitation also affects its predecessors Turbo$_{ISO}$ [64] and CFL-Match [17]. VEQ is based on the DAF techniques but it exploits the constraints induced during the matching process to reduce the search space. The released VEQ software searches only up to 100k matches, and, similarly to DAF, allows labels on vertices but not on edges. The Glasgow solver

is a recent algorithm based on constraint programming. It makes use of domain-specific search and inference techniques for solving computationally hard SubGI instances which are often represented by graphs with a few labels. With respect to the techniques introduced in VF2 [36] and VF3 [27], in [21] it is shown that RI-DS outperforms VF2. Regarding VF3, the currently available executable can only solve induced subgraph isomorphism, which is not the focus of *ArcMatch*. For this reason, we decided to exclude VF3 from the comparisons.

Table 3.1 on page 81 reports the functional features of the state-of-the-art software packages with which we compare.

RI-DS, *ArcMatch* (and *ArcMatch*-lt), and Glasgow are able to list all the subgraph isomorphism matches between the query and the target graph. By contrast, current implementations of DAF and VEQ return only the count of the matches.

DAF (and similarly VEQ) first performs the combinatorial search on a core set of query vertices with constraints affecting other vertices in the core. Peripheral vertices, whose constraints do not affect subsequent vertices, are examined later. Thus, the core set is examined using a SubGI searching approach. Then, for each instance of the core set, the number of global instances produced by extending the core is calculated by exploiting the current domain sizes of peripheral vertices. This strategy implies that DAF does not give the details of global matches, so it cannot return the mapping instances between the target graph and the query. Moreover, the available version of VEQ does not retrieve all the matches between a query graph and a target graph, but it stops at a maximum of 100,000. Thus, the results reported here refer to the counting problem. Further, in the tests in which VEQ is included, algorithms were modified to stop at the first 100k occurrences.

Each test in each benchmark consisted of a single query on a single target graph, and by setting a timeout of 600 seconds. Queries were randomly

extracted subgraphs from target graphs. The query extraction procedure uses a uniform random distribution to pick up a vertex from the target graph. That constitutes the initial vertex of the query. Subsequently, within the target graph, a neighbor of the vertices that currently comprise the query is randomly selected. The selected vertex is added to the query together with all the edges that link it with any of the vertices already in the query. The process is repeated until the desired number of vertices (or edges) is reached.

The command `/usr/bin/time -f"%e %s %M"` was used to measure time and memory consumption. In addition, internal timers have been injected into the source code. All tests were run on an Intel(R) Core(TM) i7-5960x with 64-Gb of RAM machine running a Ubuntu 64-bit 18.04 LTS system. All the evaluated tools are written in C++. DAF and VEQ are provided as executable files but without source code.

We applied the empirical non-parametric paired test described in [78] to evaluate the statistical significance of a predominance of an algorithm with respect to the others. This method employs a bootstrapping procedure, a resampling technique used to estimate the distribution of a statistic by generating numerous samples from the observed data. By analyzing the resulting distribution, we obtain confidence intervals for the performance differences, which allows us to assess the statistical significance without making strong assumptions about the underlying data distribution. In particular, we tested whether the algorithm with the lowest average running time was also statistically the best one. We put an asterisk on the charts if the fastest algorithm is statistically significantly better than all other algorithms at a p-value level of 0.05 or less.

In what follows, Section 3.2.1 gives details of the benchmarks used for the comparisons. Section 3.2.2 reports the main tests that regard the search for subgraph isomorphism with an unlimited number of matches to be reported. The results of tests performed by limiting the number of reported matches

to the first 100k occurrences in Section 3.2.3. This type of search is not the main goal of the proposed approach, but we show such results to enable comparison with VEQ. Lastly, Section 3.2.4 reports scalability tests.

We show only the time requirements of the compared approaches, without details regarding their memory occupancy of them. In general, the memory requirement of all the compared algorithms is below 300Mb, so we consider memory consumption to be irrelevant and do not show the details. RI-DS and Glasgow have lower memory requirements (compared to *ArcMatch*) because they do not employ additional data structures, except the vertex domains. VEQ and DAF have similar requirements. *ArcMatch* generally requires less memory than VEQ and DAF.

3.2.1 Benchmarks

We conducted the experiments on two different types of graphs, depending on the type of labels that are taken into account. *Fully labelled graphs* refers to experiments on graphs in which both vertices and edges have one label each. By contrast, *vertex labelled graphs* concerns experiments in which only vertex labels are used. Graphs also differ by the application domain: protein-protein interactions networks and co-authorship networks.

The PPI (Protein-Protein interaction) networks data set is a popular benchmark for subgraph isomorphism [21], because it is composed of different target graphs varying in size and density. It contains PPIs belonging to 11 different species, that have from 5,000 to 12,000 vertices, and an average degree ranging from 8 to 54. Target graphs have labels on vertices and edges. The number of labels varies from 8, 16, 32, 64, 128 to 256. Queries were randomly extracted from the labelled target graphs by setting the number of vertices to 4, 8, 12, 16 or 32. For each number of query vertices, 10 queries were extracted from each labelled target. The average query density is 0.169, with minimum and maximum values of 0.057 and 0.437, respectively.

This method helps identify biologically significant patterns, such as protein complexes or signaling pathways, within large and complex PPI networks. By efficiently handling both vertex and edge labels, it aids in understanding key cellular processes and has potential applications in drug discovery.

DBLP is a co-authorship network in which vertices are authors and they are connected if they published at least one research paper together. We downloaded the network from the SNAP database [1]. The graph has 317,080 vertices and 1,049,866 edges. We randomly labelled the network with 20 (as proposed by DAF's authors [63]), 100 and 1000 vertex labels. Moreover, we labelled the network with a set of 11,760 labels corresponding to the output of a community detection procedure, as described in [156]. Each label identifies a specific community: each vertex is labelled according to the one community it belongs to. Figure 3.2 on the next page reports statistics regarding the size of communities in terms of the number of vertices. The largest community, consisting of 56k vertices, represents vertices which were not assigned to a particular community. Thus, 17% of the vertices are labelled with this particular 'unassigned' community. The second largest community consists of 7.5k (2%) vertices. The 50 (over 11k) largest communities cover in total 50% of the network.

Sets of 100 queries were randomly extracted for each of the four target networks (three random labelling plus one community membership). Each set is composed of queries having the same number of vertices ranging from 5, 10, 15, 20 to 50. Thus, 500 queries were extracted from each target graph, with a total of 2,000 queries for the entire benchmark. The average query density is 0.191, with minimum and maximum values of 0.040 and 0.640, respectively.

[1] `https://snap.stanford.edu/data/com-DBLP.html`

Table 3.1: Functional features of the compared approaches.

Algorithm	Vertex labels	Edge labels	Listing	Counting	Count limit
ArcMatch	✓	✓	✓	✓	No limit
RI-DS	✓	✓	✓	✓	No limit
Glasgow	✓	✓	✓	✓	No limit
DAF	✓			✓	No limit
VEQ	✓			✓	100k

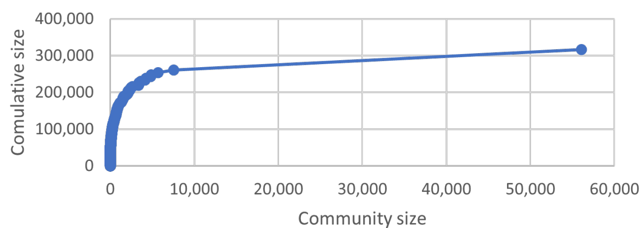


Figure 3.2: Size (x-axis) and cumulative size (y-axis) of communities in the co-authorship network in ascending order by community size. By assigning a distinct label to each community. the chart shows that labels are not uniformly distributed within the graph. In particular, there is one community which predominates with more than 50k members.

3.2.2 Subgraphs Isomorphism

Our comparisons regarding subgraph isomorphism involve the state-of-the-art methodologies that have the same functionality as *ArcMatch* (see the functionality discussion in Section 3.2 and reported in Table 3.1 on the preceding page) in solving such a problem. This means that they can handle graphs with labels on vertices and edges, and they search and return all the occurrences of the query graph within the target graph. Those are RI-DS, Glasgow and *ArcMatch*.

Table 3.2 on the next page and Figure 3.3 on page 85 show the comparison over the PPI benchmark on target graphs having both vertex and edges labels. The proposed approach results in the fastest method for the majority of the problem instances that were taken into account. It is outperformed by RI-DS in large queries (32 vertices) and on graphs having a low number of edge labels (8 edge labels). This behaviour is mainly due to the costs of the reduction techniques implemented by *ArcMatch*, that result to be expensive on large queries or when the information coming from edge labels is not enough to cover the cost of exploiting it through reduction. In fact, RI-DS has lower expensive reduction techniques and it completely avoids the handling of edge labels in such techniques.

Table 3.2: Average running times in seconds over the PPI benchmark, grouping queries by: i) the number of query vertices along with varying numbers of vertex and edge labels, ii) the number of vertex labels along with varying numbers of query vertices and edge labels, iii) the number of edge labels along with varying numbers of query vertices and vertex labels. The command `/usr/bin/time -f"%e %s %M"` was used to measure time. *ArcMatch* is nearly almost the fastest.

algorithm	Query vertices					Vertex labels						Edge labels					
	4	8	12	16	32	8	16	32	64	128	256	8	16	32	64	128	256
ArcMatch	0.02	0.02	0.02	0.03	0.07	0.04	0.03	0.03	0.03	0.03	0.03	0.04	0.03	0.03	0.03	0.03	0.03
RI-DS	0.05	0.05	0.05	0.05	0.05	0.05	0.04	0.05	0.04	0.05	0.05	0.06	0.05	0.05	0.05	0.05	0.05
Glasgow	1.81	2.03	2.25	1.92	1.99	2.70	1.98	1.89	1.80	1.77	1.76	3.71	1.89	1.88	1.88	1.89	1.89

Subsequently, we restricted the test to a graph having labels only on vertices in order to compare the proposed approach with DAF. It is a methodology in which the internal domain structure is more similar to *ArcMatch*, thus it represents a more close approach. Figure 3.4 on the next page shows the comparison on the PPI benchmark for target graphs having vertex labels but ignoring edge labels. In this case, *ArcMatch* is the fastest algorithm when the results are investigated by looking at the trends (Figure 3.4 (a)) w.r.t. the query size. Up to 16 query vertices, *ArcMatch* is statistically better than all the other approaches. For 32 query vertices is still the fastest on average, but it becomes comparable to DAF and thus the statistical significance of its performance is lost. When the results are examined by looking at the trends on varying the number of vertex labels (Figure 3.4 (b)), again *ArcMatch* is the fastest approach for the majority of the instances except when graphs are equipped with a small number of labels (8 vertex labels). In this case, *ArcMatch* becomes comparable to DAF. However, the subfigure 3.4 (c) shows the details of such results. In particular, it shows that *ArcMatch* is statistically the best approach for queries having between 8 and 16 vertices, independently of the number of vertex labels.

Table 3.3 and Figure 3.5 (a, b and c) on page 86 report the results of the proposed approach, RI-DS and DAF over the co-authorship network. The smaller the number of random labels, the greater the running time of all the compared approaches. As noted, however, the unassigned community covers 17% of the network, so the overall label frequency distribution is skewed (see Figure 3.2 on page 81). Both factors affect the filtering power of each algorithm. *ArcMatch* outperforms other systems when the number of query vertices is sufficiently high, for example, when that number exceeds 15.

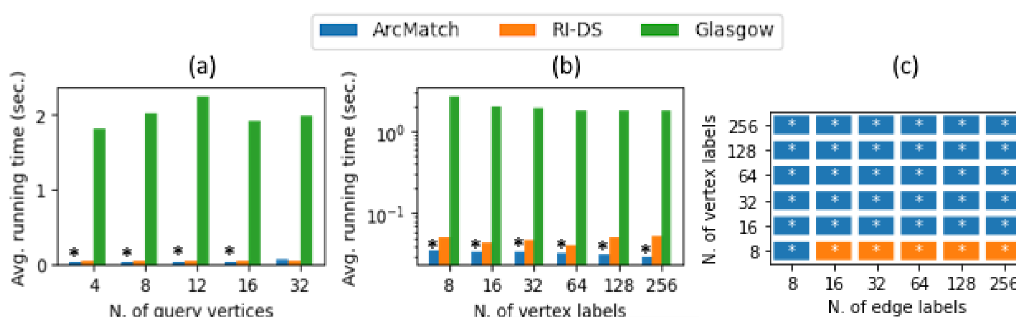


Figure 3.3: Comparison of *ArcMatch* with those systems that allow an arbitrarily large number of returned matches on the Protein-Protein Interaction (PPI) graphs with both vertex and edges labels. a) Average running times grouped by the number of query vertices along with varying numbers of vertex and edge labels. b) Average running times grouped by the number of vertex labels along with varying numbers of query vertices and edge labels. c) Algorithm with the lowest average running time, grouping queries by the number of query vertices and edge labels, along with varying numbers of vertex labels. *ArcMatch* is nearly always the fastest, sometimes significantly so. An asterisk (*) indicates that the fastest algorithm is statistically significantly better than all others at a p-value level of 0.05 or less. We applied the empirical non-parametric paired test described in [78].

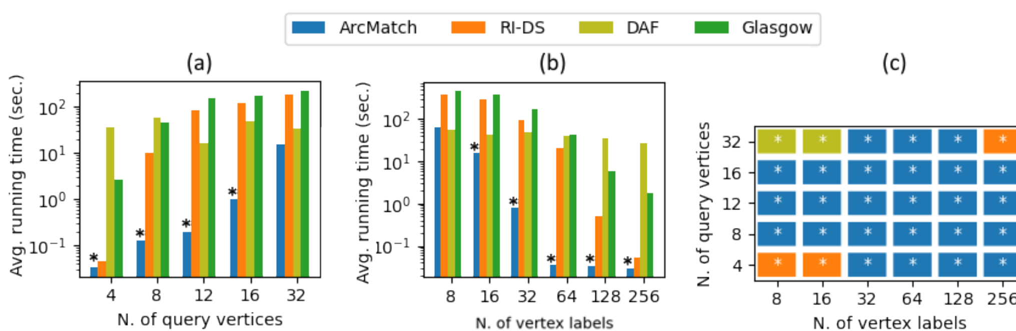


Figure 3.4: Comparison of *ArcMatch* with other systems that allow an arbitrarily large number of returned matches on the Protein-Protein Interaction (PPI) graphs with vertex labels but ignoring edge labels. Subfigures a) and b) depict the average running times, grouping queries by the number of query vertices along with varying the numbers of vertex labels, and by the number of vertex labels along with varying numbers of query vertices, respectively. Subfigure c) shows the algorithm with the lowest average running time, grouping queries by the number of query vertices and the number of vertex labels. *ArcMatch* tends to dominate with 8 or more query vertices, as long as there are a sufficient number of vertex labels. An asterisk (*) indicates that the fastest algorithm is statistically significantly better than all others at a p-value level of 0.05 or less. We applied the empirical non-parametric paired test described in [78].

Table 3.3: Average running times over the co-autorship network, grouping queries by: i) the number of query vertices along with varying the numbers of vertex labels, ii) the number of vertex labels along with varying numbers of query vertices. The command `/usr/bin/time -f"%e %s %M"` was used to measure time. For large numbers of query vertices or vertex labels, *ArcMatch* is the fastest.

algorithm	Query vertices					Vertex labels			
	5	10	15	20	50	20	100	1000	11760
ArcMatch	4.68	20.44	64.26	140.32	224.25	271.86	0.51	0.49	90.30
RI-DS	0.39	5.53	21.11	40.66	185.65	82.02	0.41	0.39	119.84
DAF	235.05	305.37	333.79	345.76	410.08	321.08	185.80	197.16	600.00

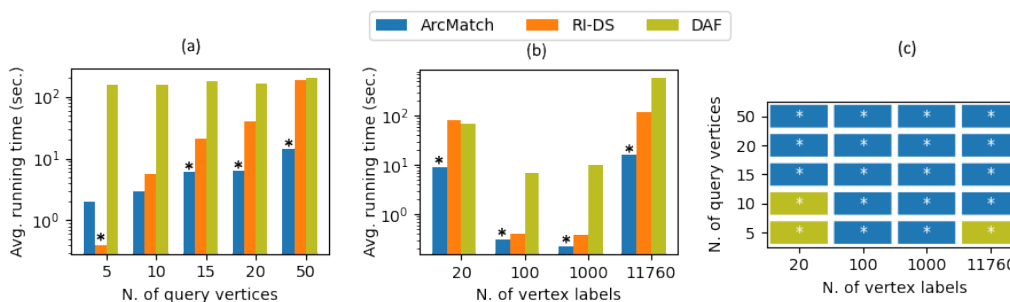


Figure 3.5: Comparison of *ArcMatch* with those systems that allow an arbitrarily large number of returned matches over the co-authorship benchmark (which has vertex labels only). Subfigures a) and b) depict the average running times, grouping queries by the number of vertex labels along with varying numbers of query vertices, and by the number of query vertices along with varying the number of vertex labels, respectively. Subfigure c) shows the algorithm with the lowest average running time, grouping queries by the number of query vertices and the number of vertex labels. *ArcMatch* demonstrates superior performance when the number of query vertices is sufficiently high (e.g., 15 or more) or there are many vertex labels. An asterisk (*) indicates that the fastest algorithm is statistically significantly better than all others at a p-value level of 0.05 or less. We applied the empirical non-parametric paired test described in [78].

3.2.3 Subgraphs isomorphism with a limited number of matches

Figure 3.4 (d, e and f) on page 85 present a comparison of retrieving the first 100k matches on the Protein-Protein Interaction (PPI) benchmark. The analysis focuses on target graphs that have vertex labels but ignoring edge labels. For these experiments, we compared *ArcMatch-lt*, DAF, and VEQ. *ArcMatch-lt* is much faster than the other systems, especially when there is a large number of possible vertex labels, exceeding 64.

In searching for the first 100k occurrences, all the approaches finished a similar number of instances as when searching for all the occurrences (not shown here). The analysis focuses on target graphs that have vertex labels but ignore edge labels in order to compare the proposed approach with VEQ, which is equipped with an evolved set of techniques introduces in DAF but whose available implementation only reports the first 100k occurrences. *ArcMatch-lt* always outperforms *ArcMatch* for the 100,000 match problem. Complex path reduction procedures is not necessary for this sort of limited search. The convergence of vertex domain reduction is enough to reduce overall execution times. For this reason, we only show the performance of *ArcMatch*-lt.

Figure 3.6 on page 91 presents a comparison of retrieving the first 100k matches on the two real benchmarks. In general, VEQ finishes a number of instances that is comparable to that of *ArcMatch*-lt (not shown here). Regarding the PPI benchmark, *ArcMatch-lt* is much faster than the other systems, especially when there is a large number of possible vertex labels, exceeding 64. for the co-authorship network, *ArcMatch-lt* exhibits the highest performance when the number of vertex labels is between 100 and 1000. VEQ performs better in other cases.

3.2.4 Scalability Tests on Synthetic Graphs

We performed a scalability analysis of the compared algorithms in order to investigate how the running time of the algorithm depends on various properties of the involved query and target graphs.

The number of target vertices is considered to be the most important factor affecting the running time of SubGI algorithms [5, 24, 104, 25, 26, 63]. For that reason, we used synthetic graphs generated by means of specific random models: Erdos-Rényi [47], Barabási-Albert [10], and the Forest Fire [91] models. These models were selected to represent different topological features and structures, providing a comprehensive evaluation of algorithm performance. Such a collection was previously used in [5].

The **Erdos-Rényi** model generates random graphs where each possible edge between two nodes is included with a fixed probability p. This leads to graphs with a binomial degree distribution, where most vertices have similar degrees, making this model suitable for evaluating algorithm performance on homogeneous, unclustered networks. Erdos-Rényi graphs tend to lack large hubs or strong community structures, thus allowing us to assess how algorithms handle more uniform and less structured networks.

The **Barabási-Albert** model is designed to generate scale-free networks, where the degree distribution follows a power law, meaning that a few nodes act as hubs with many connections, while most nodes have relatively few connections. These graphs are more representative of real-world networks, such as social or biological systems, where preferential attachment mechanisms lead to the formation of hubs. This model tests the algorithm's ability to deal with high-degree nodes and long-tail degree distributions.

The **Forest Fire** model simulates networks with a high clustering coefficient and community structure. It captures the property of densification and shrinking diameters over time, often seen in citation and collaboration networks. Forest Fire graphs exhibit hierarchical and densely connected com-

munity substructures, making them suitable for evaluating how well the algorithm handles networks with strong clustering and varying community sizes.

Target graphs were created by fixing the desired number of vertices, then varying the parameters of the random models, such as the probability p in Erdos-Rényi or the number of edges to attach in Barabási-Albert, to control graph density. Query graphs were extracted randomly from target graphs. See [5] for a detailed description of the benchmark.

We investigate the scalability of the monomorphism problem alone and only for those algorithms which have no strict limitation on the number of return matches. For that reason, VF3, VEQ and *ArcMatch*-lt were excluded from the analysis. We also excluded the Glasgow algorithm for the analysis of synthetic graphs because it is optimized for search on unlabeled graphs.

For each benchmark, the running times of the algorithm were grouped by specific properties of the involved graph in order to investigate the dependency of the algorithm w.r.t. the specified properties. In particular, we took into account the number of vertices of the target graph, the density of the target graph, the number of distinct labels in the target graph and the number of vertices of the query graph. These properties are most often used to compare the performance of SubGI algorithms [5, 24, 104, 25, 26, 63].

Results regarding the Barabási-Albert [10], the Erdos-Rényi [47] and the Forest Fire [91] models are shown in Figures 3.7 on page 91, 3.8 and 3.9 on page 92, respectively. All the analyses show that the running time of *ArcMatch* does not depend on the number of target vertices and on variations in target density. By contrast, when the number of distinct target labels is large, *ArcMatch* exploits the filtering power of vertex labels very well. As a result, the running time of *ArcMatch* decreases as the number of distinct target labels increases. While all algorithms improve with large numbers of vertices, *ArcMatch* exploits such information the best. Similarly, but with an opposite trend, all the algorithms show a clear dependence on the number

of query vertices. Each algorithm shows a specific trend, but in general *ArcMatch* is the fastest algorithm for reasonable query sizes (up to 32).

There is a slight anomaly regarding Barabási-Albert and Forest Fire networks. In generators for that model, large target graphs have lower edge density because generating them with high-density values requires large computational resources. That lower edge density is the reason the graphs show a decrease in running time as the number of nodes in the target graph increases.

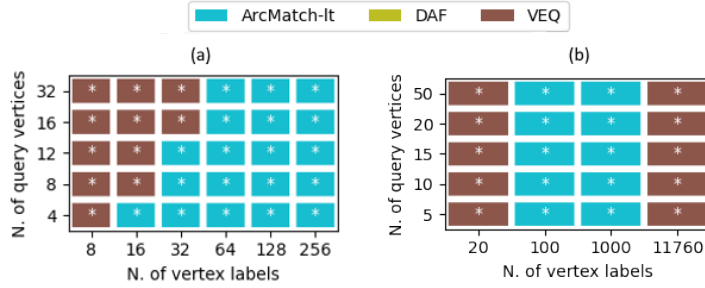


Figure 3.6: The figure shows the algorithm with the lowest average running time, grouping queries by the number of query vertices and the number of vertex labels for the PPI benchmark (subfigure a)) and for the co-authorship benchmark (subfigure b)). *ArcMatch*-lt outperforms the other approaches for a sufficiently high number of vertex labels when they are uniformly distributed (thus excluding the 11760-labels labeling of the co-authorship network) over the target graph vertices. An asterisk (*) indicates that the fastest algorithm is statistically significantly better than all others at a p-value level of 0.05 or less. We applied the empirical non-parametric paired test described in [78].

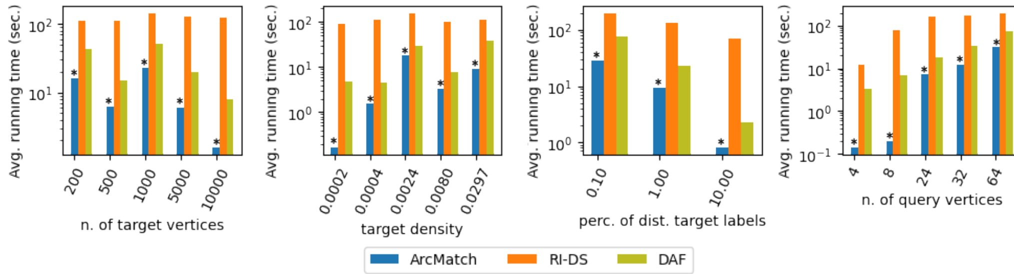


Figure 3.7: Scalability analysis for the synthetic benchmark built by means of the Barabási-Albert model. Charts are drawn by grouping the running times of each algorithm according to the properties of the target and query graphs. While the number of target vertices and overall target density does not affect the relative computational requirements of the algorithms, the number of query vertices has a strong effect. *ArcMatch* consistently wins, often in a statistically significantly manner (low p-value). An asterisk (*) indicates that the fastest algorithm is statistically significantly better than all others at a p-value level of 0.05 or less. We applied the empirical non-parametric paired test described in [78].

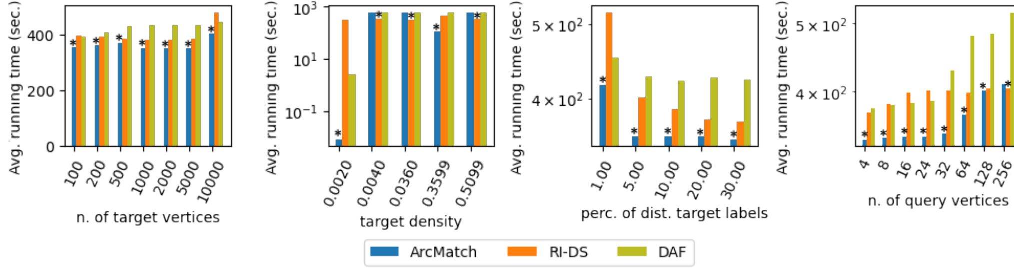


Figure 3.8: Scalability analysis for the synthetic benchmark built by means of the Erdos model. Charts are drawn by grouping the running times of each algorithm according to properties of the target and query graphs. Number of target vertices and target density do not affect trends in computational requirements of the algorithms. Dependencies emerge for number of distinct target labels and number of query vertices. *ArcMatch* consistently wins, often with a low p-value. An asterisk (*) indicates that the fastest algorithm is statistically significantly better than all others at a p-value level of 0.05 or less. We applied the empirical non-parametric paired test described in [78].

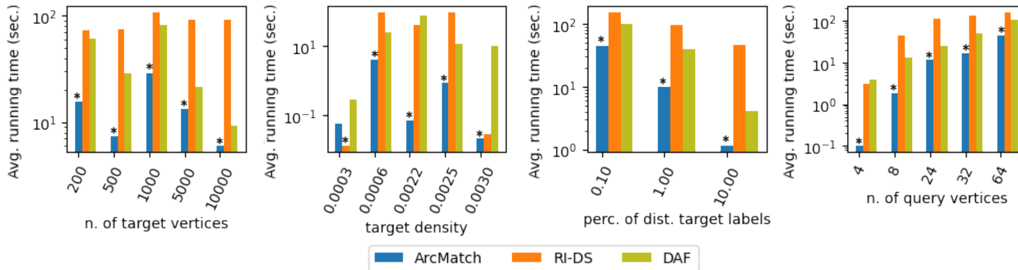


Figure 3.9: Scalability analysis for the synthetic benchmark built by means of the Forest Fire model. Charts are drawn by grouping the running times of each algorithm according to properties of the target and query graphs. Number of target vertices and target density do not affect trends in computational requirements of the algorithms. Dependencies emerge for number of distinct target labels and number of query vertices. *ArcMatch* consistently wins, often with a low p-value. An asterisk (*) indicates that the fastest algorithm is statistically significantly better than all others at a p-value level of 0.05 or less. We applied the empirical non-parametric paired test described in [78].

3.3 Detailed comparison of the techniques involved in *ArcMatch*

In what follows, we evaluate the advantage that each feature gives to *ArcMatch*. A baseline version of the methodology is included: it builds vertex and edge domains, reduces vertex domains with a single run of arc consistency, computes the variable ordering as in [21] (measures N_1, N_2 and N_3 are used), but it does not apply further techniques. In the baseline version, the search process is not performed on top of the domain graph as described in Section 3.1.3. In this way, the baseline is very close to the RI-DS algorithm with the difference that it computes the initial content of edge domains. The baseline version is then augmented with the following features in order to determine both the advantage/disadvantage of each feature individually and in combination.

The features are:

- **NS**: the variable ordering is performed by also using measures N_4 and N_5 (see Section 3.1.2).
- **ED**: the search process is driven by the domain graph. Thus, candidates are extracted from edge domains (see Section 3.1.3). However, no dynamic parent selection is performed. Parents are chosen statically before the search process begins as in [21].
- **DY**: as for ED but dynamic parent selection is enabled (see Section 3.1.3).
- **VC**: arc consistency is applied to vertex domains until convergence (see Section 2.1.1).
- **RE**: apply path reduction procedure (`PathReduction`) as it is described in Section 3.1.1, but disable the call to the function `RefineDomains`,

which reduces edge domains.

- **RR**: reduce edge domains by also enabling `RefineDomains`

- **PV**: enable the management of peripheral vertices by using the procedure described in Section 3.1.2.

We evaluated the advantages brought by the *ArcMatch* features and their combinations.

16 versions of *ArcMatch* were obtained by combining the different features (NS, ED, DY, VC, RE, RR and PV). Table 3.4 on page 97 shows the number of queries that were finished within the 600 seconds timeout by the tested solutions.

Results are grouped by number of target vertex labels (from 1 to 1024). In that grouping, graphs having more than one edge label are discarded. In subsequent tests, only instances regarding target graphs having one vertex label are taken into account, and they are grouped by number of edge labels (from 1 to 1024). The last column reports the total number of finished instances, as a function of the number of labels that are assigned to vertices or edges. Each reported grouping is composed of a total of 250 instances except for the *all* grouping that includes all the corresponding instances. For each column, the solution that solved the maximum number of instances is highlighted in bold.

The table shows that RI-DS and Glasgow are always outperformed by at least one *ArcMatch* solution. As expected, all the approaches are more effective with increasing numbers of labels, regardless of whether the labels are assigned to vertices or edges. The reason is that increasing the number of labels decreases the number of target vertices (edges) that are compatible with a given query vertex (edge). With more that 32 labels, almost every instance is solved by all the approaches within the timeout. *ArcMatch*-7 is the configuration that performs best overall. It finished a total of 8,761

instances. Such a configuration does not go to convergence when reducing vertex domains. Thus, it postpones a more accurate filtering of domains to path-based reduction. This result shows the advantages of using vertex and edges domains but without forcing their reduction to convergence.

Similarly, when there are no labels or, equivalently, the same label for all vertices and the same label for all edges, solutions that do not exploit complex techniques, such as domain reduction, enjoy substantial advantages (see for example *ArcMatch*-1). When there are a small number of labels, the filtering power of reduction techniques is minimal, so their benefits are not worth their computational costs.

When the number of vertex/edge candidates is high, a specialized procedure for managing peripheral query vertices is worthless. The technique can only exploit domains that do not overlap, but, with a few labels, domains tend to have non-null intersections. Thus, overriding the general ordering strategy by postponing the processing of peripheral vertices is counterproductive.

The second configuration that performs well overall is *ArcMatch*-13. It does not reduce edge domains but it exploits them during the searching process. Moreover, it reduces vertex domains until convergence to make up for the reduced candidate filtering power.

Table 3.5 on page 98 reports the running time of the compared solutions. Average (and standard deviation) on running times for different groupings are reported. Results are grouped by vertex labels by taking into account only instances with one edge label. Instances with one label have an average running time close to 600 because a timeout of 600 seconds is applied and most of the instances reached the timeout. The average running time correlates with the number of finished instances (see Table 3.7 on page 100). With 128 and more labels, the instances are solved in less than 1 second. In Table 3.8 on page 101, all instances are grouped by the number of query

vertices, independently of from the number of target labels. The table shows that the running time depends on the number of query vertices. For some algorithms, there is an order of magnitude increase from 4 to 32 query vertices (for example *ArcMatch*-1 and *ArcMatch*-10), while other configurations just double their time. The predominance of *ArcMatch*-7 is very clear. Such a configuration is always the best solution except for instances with only one label and for queries with just 4 vertices. These types of instances are too simple to benefit from the advantages of sophisticated filtering techniques, such as edge domain reduction. In fact, configurations that do not reduce edge domains are the fastest ones on queries with four vertices. Similar considerations apply when no edge labels are present, as it is shown in Table. 3.6 on page 99.

Lastly, we evaluated the significance of the features in searching for the first 100k matches. Results are shown in Table 3.9 on page 102. Combinations of features that do not exploit complex reduction techniques outperform more sophisticated approaches. For this number of matches, *ArcMatch*-8 , which exploits arc-consistency only, is the fastest solution, and *ArcMatch*-7 is slower. More generally, combinations that exploit edge domains enjoy significant advantages when there are no labels. When there are many labels, gaps between the studied solutions shrink. From 32 labels on, all solutions have compatible running times.

In conclusion, we select *ArcMatch*-7 as the main version of the proposed approach and propose *ArcMatch*-13 as a light version that can be used, for example, for finding the first 100k matches.

Table 3.4: Number of finished instances for the PPI benchmark, by varying number of vertex and edge labels, on searching for all the occurrences.

algorithm	*ArcMatch* features							vertex labels (1 edge label)							edge labels (1 vertex label)							All
	NS	ED	DY	VC	RE	RR	PV	1	16	32	128	256	1024	all	1	16	32	128	256	1024	all	
ArcMatch-1								**46**	128	210	**250**	**250**	**250**	1,134	**46**	46	48	177	211	244	772	8,107
ArcMatch-2		✓						39	129	211	**250**	**250**	**250**	1,129	39	95	199	249	**250**	**250**	1,082	8,422
ArcMatch-3		✓	✓					**46**	143	220	**250**	**250**	**250**	1,159	**46**	**244**	**248**	**250**	**250**	**250**	**1,288**	8,651
ArcMatch-4	✓	✓	✓					45	140	218	**250**	**250**	**250**	1,153	45	**244**	**248**	**250**	**250**	**250**	1,287	8,645
ArcMatch-5	✓	✓	✓		✓			34	140	219	**250**	**250**	**250**	1,143	34	225	242	**250**	**250**	**250**	1,251	8,610
ArcMatch-6	✓	✓	✓				✓	20	186	237	**250**	**250**	**250**	1,193	20	218	227	246	247	247	1,205	8,628
ArcMatch-7	✓	✓	✓		✓		✓	42	**245**	**250**	**250**	**250**	**250**	**1,287**	42	230	244	**250**	**250**	**250**	1,266	**8,761**
ArcMatch-8				✓				**46**	131	212	**250**	**250**	**250**	1,139	**46**	54	189	**250**	**250**	**250**	1,039	8,382
ArcMatch-9		✓		✓				39	132	217	**250**	**250**	**250**	1,138	39	119	234	**250**	**250**	**250**	1,142	8,491
ArcMatch-10		✓	✓	✓				**46**	143	220	**250**	**250**	**250**	1,159	**46**	**244**	**248**	**250**	**250**	**250**	**1,288**	8,651
ArcMatch-11	✓	✓	✓	✓				45	140	219	**250**	**250**	**250**	1,154	45	**244**	**248**	**250**	**250**	**250**	1,287	8,646
ArcMatch-12	✓	✓	✓	✓	✓			33	140	219	**250**	**250**	**250**	1,142	33	229	247	**250**	**250**	**250**	1,259	8,618
ArcMatch-13	✓	✓	✓	✓			✓	19	187	239	**250**	**250**	**250**	1,195	19	226	243	**250**	**250**	**250**	1,238	8,664
ArcMatch-14	✓	✓	✓	✓	✓		✓	12	186	241	**250**	**250**	**250**	1,189	12	215	240	**250**	**250**	**250**	1,217	8,644
ArcMatch-15	✓	✓	✓	✓	✓	✓		33	140	219	**250**	**250**	**250**	1,142	33	230	**248**	**250**	**250**	**250**	1,261	8,620
ArcMatch-16	✓	✓	✓	✓	✓	✓	✓	12	186	241	**250**	**250**	**250**	1,189	12	218	242	**250**	**250**	**250**	1,222	8,649

Table 3.5: Average running times (and standard deviation) over varying number of vertex labels on the PPI benchmark when searching for all the occurrences. The command `/usr/bin/time -f"%e %s %M"` was used to measure time.

algorithm	*ArcMatch* features							vertex labels (1 edge label)					
	NS	ED	DY	VC	RE	RR	PV	1	16	32	128	256	1024
ArcMatch-1								504.61 (207.37)	308.61 (292.66)	113.61 (226.97)	0.70 (5.19)	0.03 (0.02)	0.02 (0.02)
ArcMatch-2		✓						537.77 (152.51)	309.28 (291.17)	103.03 (219.34)	0.05 (0.24)	**0.02** (0.02)	**0.02** (0.01)
ArcMatch-3		✓	✓					**503.02** (208.77)	273.83 (291.06)	81.66 (197.48)	0.11 (1.11)	**0.02** (0.02)	**0.02** (0.02)
ArcMatch-4	✓	✓	✓					505.19 (206.59)	278.31 (290.58)	88.23 (203.57)	0.23 (3.21)	**0.02** (0.02)	**0.02** (0.02)
ArcMatch-5	✓	✓	✓		✓			563.60 (107.77)	278.90 (290.47)	88.11 (203.79)	0.08 (0.61)	0.03 (0.06)	0.03 (0.05)
ArcMatch-6	✓	✓	✓				✓	559.47 (144.67)	167.44 (261.26)	39.85 (139.82)	**0.02** (0.02)	**0.02** (0.01)	**0.02** (0.02)
ArcMatch-7	✓	✓	✓		✓		✓	549.25 (124.52)	**160.60** (89.73)	**20.79** (58.50)	0.03 (0.07)	0.03 (0.06)	0.03 (0.05)
ArcMatch-8				✓				504.69 (207.34)	303.95 (292.20)	102.71 (218.46)	0.34 (3.58)	**0.02** (0.02)	**0.02** (0.02)
ArcMatch-9		✓		✓				537.98 (152.17)	301.72 (290.69)	90.28 (205.76)	0.04 (0.17)	**0.02** (0.02)	**0.02** (0.02)
ArcMatch-10		✓	✓	✓				503.64 (208.41)	273.86 (291.26)	81.67 (197.75)	0.11 (1.12)	**0.02** (0.02)	**0.02** (0.02)
ArcMatch-11	✓	✓	✓	✓				506.10 (206.15)	278.57 (290.71)	87.60 (202.71)	0.07 (0.67)	**0.02** (0.02)	**0.02** (0.02)
ArcMatch-12	✓	✓	✓	✓	✓			563.99 (106.86)	278.97 (290.43)	88.37 (204.12)	0.08 (0.61)	0.03 (0.06)	0.03 (0.05)
ArcMatch-13	✓	✓	✓	✓			✓	559.97 (144.66)	163.34 (259.42)	31.67 (126.59)	**0.02** (0.01)	**0.02** (0.01)	**0.02** (0.01)
ArcMatch-14	✓	✓	✓	✓	✓		✓	587.13 (64.90)	163.99 (260.70)	28.13 (117.27)	0.03 (0.06)	0.03 (0.06)	0.03 (0.05)
ArcMatch-15	✓	✓	✓	✓	✓	✓		563.47 (106.68)	278.98 (290.55)	87.92 (203.59)	0.09 (0.69)	0.03 (0.06)	0.03 (0.06)
ArcMatch-16	✓	✓	✓	✓	✓	✓	✓	587.31 (64.85)	163.70 (260.52)	29.02 (118.25)	0.03 (0.06)	0.03 (0.06)	0.03 (0.05)

Table 3.6: Average running times (and standard deviation) over the fully labelled PPI benchmark extracting only instances where target graphs have one edge label, when searching for all the occurrences. The command `/usr/bin/time -f"%e %s %M"` was used to measure time. Results are grouped by the number of query vertices.

algorithm	*ArcMatch* features							query vertices					All
	NS	ED	DY	VC	RE	RR	PV	4	8	12	16	32	
ArcMatch-1								20.53 (86.53)	111.31 (231.98)	170.27 (264.51)	208.89 (283.14)	261.99 (295.58)	154.60 (257.92)
ArcMatch-2		✓						48.16 (134.30)	111.87 (231.61)	168.43 (263.41)	205.84 (282.51)	257.51 (295.67)	158.36 (258.45)
ArcMatch-3		✓	✓					**19.20** (81.77)	**105.62** (227.25)	157.68 (259.83)	185.95 (273.17)	247.10 (293.65)	143.11 (251.22)
ArcMatch-4	✓	✓	✓					21.01 (87.67)	105.80 (227.33)	160.73 (261.29)	188.67 (273.45)	250.46 (293.90)	145.33 (252.28)
ArcMatch-5	✓	✓	✓		✓			69.69 (171.99)	105.95 (227.31)	161.02 (260.99)	189.15 (273.89)	249.82 (294.26)	155.13 (256.92)
ArcMatch-6	✓	✓	✓				✓	66.24 (184.19)	100.23 (223.88)	**111.80** (232.41)	**132.03** (242.21)	**228.72** (288.61)	127.80 (242.61)
ArcMatch-7	✓	✓	✓		✓		✓	62.44 (157.59)	95.40 (216.02)	100.16 (223.91)	100.84 (223.79)	112.99 (232.94)	**94.36** (212.99)
ArcMatch-8				✓				20.59 (86.83)	111.37 (232.09)	169.63 (264.31)	204.66 (280.68)	253.52 (295.63)	151.95 (256.51)
ArcMatch-9		✓		✓				48.33 (134.63)	111.87 (231.63)	167.46 (263.35)	199.11 (278.89)	248.27 (293.71)	155.01 (256.29)
ArcMatch-10		✓	✓	✓				19.71 (84.00)	105.60 (227.24)	157.83 (259.93)	186.02 (273.35)	246.94 (293.77)	143.22 (251.40)
ArcMatch-11	✓	✓	✓	✓				21.77 (90.96)	105.81 (227.34)	160.93 (260.98)	188.81 (273.56)	249.68 (294.19)	145.40 (252.41)
ArcMatch-12	✓	✓	✓	✓	✓			70.01 (172.46)	105.95 (227.30)	161.07 (260.97)	189.26 (273.91)	249.93 (294.35)	155.25 (256.99)
ArcMatch-13	✓	✓	✓	✓			✓	66.66 (185.32)	100.20 (223.90)	112.34 (234.02)	128.51 (239.17)	221.48 (287.61)	125.84 (241.63)
ArcMatch-14	✓	✓	✓	✓	✓		✓	89.30 (207.19)	100.25 (223.88)	112.37 (234.01)	128.77 (240.27)	218.76 (286.33)	129.89 (243.92)
ArcMatch-15	✓	✓	✓	✓	✓	✓		69.58 (171.11)	106.25 (227.39)	160.82 (260.96)	189.04 (273.95)	249.74 (294.28)	155.09 (256.81)
ArcMatch-16	✓	✓	✓	✓	✓	✓	✓	89.45 (207.53)	100.25 (223.88)	112.36 (234.02)	128.51 (240.04)	219.54 (286.18)	130.02 (243.96)

Table 3.7: Number of finished instances for the vertex-labelled PPI benchmark, by varying the number of vertex labels and the number of query vertices, on searching for the first 100k occurrences.

algorithm	*ArcMatch* features							vertex labels						query vertices					All
	NS	ED	DY	VC	RE	RR	PV	1	16	32	128	256	1024	4	8	12	16	32	
ArcMatch-1								241	238	**249**	**250**	**250**	**250**	**300**	**300**	299	299	279	1,477
ArcMatch-2		✓						241	238	**249**	**250**	**250**	**250**	**300**	**300**	299	299	279	1,477
ArcMatch-3		✓	✓					238	**246**	**249**	**250**	**250**	**250**	**300**	**300**	**300**	300	283	1,483
ArcMatch-4	✓	✓	✓					241	245	**249**	**250**	**250**	**250**	**300**	**300**	**300**	**300**	285	**1,485**
ArcMatch-5	✓	✓	✓		✓			39	243	**249**	**250**	**250**	**250**	282	257	250	250	242	1,281
ArcMatch-6	✓	✓	✓				✓	203	240	**249**	**250**	**250**	**250**	298	293	290	287	274	1,442
ArcMatch-7	✓	✓	✓		✓		✓	34	240	**249**	**250**	**250**	**250**	279	255	250	249	240	1,273
ArcMatch-8				✓				**245**	243	**249**	**250**	**250**	**250**	**300**	**300**	299	**300**	**288**	1,487
ArcMatch-9		✓		✓				241	242	**249**	**250**	**250**	**250**	**300**	**300**	299	299	284	1,482
ArcMatch-10		✓	✓	✓				238	**246**	**249**	**250**	**250**	**250**	**300**	**300**	**300**	**300**	283	1,483
ArcMatch-11	✓	✓	✓	✓				242	244	**249**	**250**	**250**	**250**	**300**	**300**	**300**	**300**	285	**1,485**
ArcMatch-12	✓	✓	✓	✓	✓			38	244	**249**	**250**	**250**	**250**	282	256	250	250	243	1,281
ArcMatch-13	✓	✓	✓	✓			✓	207	244	**249**	**250**	**250**	**250**	298	293	290	292	277	1,450
ArcMatch-14	✓	✓	✓	✓	✓		✓	38	241	**249**	**250**	**250**	**250**	282	256	250	249	241	1,278
ArcMatch-15	✓	✓	✓	✓	✓	✓		41	243	**249**	**250**	**250**	**250**	284	257	250	250	242	1,283
ArcMatch-16	✓	✓	✓	✓	✓	✓	✓	39	241	**249**	**250**	**250**	**250**	283	256	250	249	241	1,279

Table 3.8: Average running times over the fully labelled PPI benchmark and over varying number of query vertices, when searching for all the occurrences. The command `/usr/bin/time -f"%e %s %M"` was used to measure time.

algorithm	*ArcMatch* features							query vertices					All
	NS	ED	DY	VC	RE	RR	PV	4	8	12	16	32	
ArcMatch-1								7.87 (53.92)	59.82 (178.46)	69.43 (190.07)	77.36 (199.87)	96.16 (218.69)	62.13 (180.54)
ArcMatch-2		✓						8.13 (57.61)	26.73 (120.94)	47.21 (158.54)	57.30 (174.94)	67.20 (188.38)	41.31 (149.26)
ArcMatch-3		✓	✓					**3.27** (34.09)	18.86 (103.44)	27.11 (122.61)	32.20 (133.17)	41.85 (151.28)	24.66 (116.91)
ArcMatch-4	✓	✓	✓					3.57 (36.58)	18.89 (103.49)	27.62 (123.67)	32.65 (133.79)	42.37 (152.03)	25.02 (117.64)
ArcMatch-5	✓	✓	✓		✓			11.94 (74.79)	20.51 (106.34)	29.61 (126.61)	35.54 (137.98)	48.43 (160.38)	29.20 (125.25)
ArcMatch-6	✓	✓	✓				✓	17.12 (96.58)	23.90 (115.99)	22.13 (111.15)	26.98 (121.13)	42.37 (151.39)	26.50 (120.88)
ArcMatch-7	✓	✓	✓		✓		✓	10.75 (68.35)	**17.69** (98.16)	**18.96** (102.63)	**20.25** (105.40)	**25.51** (117.17)	**18.63** (99.77)
ArcMatch-8				✓				6.75 (50.72)	43.20 (154.62)	49.81 (163.57)	53.42 (169.44)	61.86 (181.80)	43.01 (152.82)
ArcMatch-9		✓		✓				8.14 (57.76)	25.43 (118.15)	40.70 (146.63)	51.10 (166.19)	56.41 (174.46)	36.36 (140.26)
ArcMatch-10		✓	✓	✓				3.34 (35.02)	18.81 (103.44)	27.08 (122.68)	32.14 (133.26)	41.59 (151.19)	24.59 (116.96)
ArcMatch-11	✓	✓	✓	✓				3.69 (37.96)	18.84 (103.50)	27.60 (123.61)	32.60 (133.87)	42.05 (151.95)	24.96 (117.69)
ArcMatch-12	✓	✓	✓	✓	✓			11.94 (75.01)	20.27 (106.25)	29.12 (126.53)	34.71 (137.42)	44.51 (155.09)	28.11 (123.67)
ArcMatch-13	✓	✓	✓	✓			✓	13.97 (87.55)	20.70 (108.77)	20.62 (108.34)	24.05 (114.64)	38.47 (145.50)	23.56 (114.76)
ArcMatch-14	✓	✓	✓	✓	✓		✓	17.89 (97.62)	22.55 (112.85)	21.85 (111.18)	25.47 (118.08)	39.19 (145.81)	25.39 (118.37)
ArcMatch-15	✓	✓	✓	✓	✓	✓		11.86 (74.45)	20.20 (106.28)	28.82 (126.14)	34.00 (136.68)	43.64 (154.64)	27.71 (123.22)
ArcMatch-16	✓	✓	✓	✓	✓	✓	✓	17.90 (97.77)	21.80 (110.83)	21.13 (109.76)	24.86 (117.11)	38.85 (146.12)	24.91 (117.64)

Table 3.9: Average running times (and standard deviation) over the vertex-labelled PPI benchmark, by varying number of vertex labels in the target graph, on searching for the first 100k occurrences. The command `/usr/bin/time -f"%e %s %M"` was used to measure time.

algorithm	*ArcMatch* features							vertex labels						All
	NS	ED	DY	VC	RE	RR	PV	1	16	32	128	256	1024	
ArcMatch-1								43.47(112.01)	30.52(130.56)	4.85(53.55)	0.02(0.02)	0.02(0.01)	0.02(0.01)	13.15(75.46)
ArcMatch-2		✓						43.92(112.09)	30.53(130.58)	4.85(53.55)	0.02(0.02)	0.02(0.02)	0.02(0.02)	13.23(75.51)
ArcMatch-3		✓	✓					49.94(127.63)	**10.48**(75.67)	2.44(37.94)	**0.02**(0.01)	0.02(0.01)	0.02(0.01)	10.49(64.97)
ArcMatch-4	✓	✓	✓					44.25(112.39)	13.29(84.87)	2.44(37.95)	0.02(0.02)	0.02(0.02)	0.02(0.02)	10.01(61.57)
ArcMatch-5	✓	✓	✓		✓			557.68(114.28)	21.89(106.01)	3.08(38.25)	0.03(0.06)	0.03(0.06)	0.03(0.05)	97.12(216.31)
ArcMatch-6	✓	✓	✓				✓	147.81(228.20)	27.03(120.13)	6.24(50.59)	0.02(0.01)	**0.02**(0.01)	**0.02**(0.01)	30.19(119.72)
ArcMatch-7	✓	✓	✓		✓		✓	561.77(111.81)	31.46(126.85)	3.57(39.05)	0.03(0.06)	0.03(0.05)	0.03(0.05)	99.48(218.86)
ArcMatch-8				✓				**32.41**(85.70)	21.03(104.92)	3.49(40.83)	0.03(0.16)	0.02(0.02)	0.02(0.01)	**9.50**(59.04)
ArcMatch-9		✓		✓				43.92(112.12)	22.31(110.70)	4.30(44.56)	0.02(0.02)	0.02(0.02)	0.02(0.02)	11.77(68.72)
ArcMatch-10		✓	✓	✓				50.29(127.72)	10.49(75.69)	2.44(37.95)	**0.02**(0.01)	0.02(0.01)	0.02(0.01)	10.55(65.04)
ArcMatch-11	✓	✓	✓	✓				43.11(107.90)	15.44(92.46)	**2.43**(37.95)	**0.02**(0.01)	**0.02**(0.01)	**0.02**(0.01)	10.17(61.97)
ArcMatch-12	✓	✓	✓	✓	✓			559.34(110.71)	19.34(97.17)	2.88(38.08)	0.03(0.06)	0.03(0.06)	0.03(0.04)	96.94(216.06)
ArcMatch-13	✓	✓	✓	✓			✓	143.84(224.04)	17.03(94.69)	2.47(37.95)	**0.02**(0.01)	**0.02**(0.01)	**0.02**(0.01)	27.23(113.24)
ArcMatch-14	✓	✓	✓	✓	✓		✓	555.04(117.78)	28.28(119.65)	3.38(38.90)	0.03(0.06)	0.03(0.06)	0.03(0.05)	97.80(216.51)
ArcMatch-15	✓	✓	✓	✓	✓	✓		555.32(114.75)	20.33(101.55)	2.77(38.02)	0.03(0.06)	0.03(0.06)	0.03(0.05)	96.42(215.26)
ArcMatch-16	✓	✓	✓	✓	✓	✓	✓	555.11(118.20)	27.16(118.00)	2.77(38.02)	0.03(0.06)	0.03(0.06)	0.03(0.05)	97.52(216.50)

3.4 Relation between arc consistency and path-based reduction

In the constraint satisfaction field, the concept of arc consistency is extended to *path consistency* [43]. The reader might think there is a correspondence between the path-based reduction technique proposed in this study and path consistency. They differ, however. Path consistency is defined only in terms of the CSP (Constraint Satisfaction Problem) and aims to verify the entire set of rules relating to two or more variables. Translated into a graph theory formulation, such a concept will check for the correspondence of the subgraph between a given query vertex and a target vertex. In our path-based reduction, we are interested in verifying only that there is a corresponding path starting from the two vertices. No edges (rules) between vertices (variables) are examined outside of the path. Path consistency also differs from the reduction presented in [63], since DAF checks for non-induced substructures, rather than simple paths. Thus, DAF requires higher computational costs, it is potentially more effective in reducing domains but there is no guarantee of such performance.

Given two query vertices, v_q and u_q, connected by the edge (v_q, u_q), arc consistency verifies that each target vertex $v_t \in D(v_q)$ is consistent with respect to such a constraint. Thus, v_t must be connected with at least one vertex in $D(u_q)$. If v_t makes $D(v_q)$ inconsistent, than v_t is removed from $D(v_q)$, namely $D(v_q)$ is reduced. The removal may have a cascading effect on the other domains. Thus the reduction of $D(v_q)$ propagates to the other domains. The propagation can be performed in two different ways: i) in conjunction with the domain reduction; (ii) or it can be performed by multiple runs of arc-consistency. In the first case, each removal is systematically propagated to the other domains. In the second, case, each run is performed over the result of the previous one, and the process is repeated

until convergence. In any case, the two approaches produce the same effects. In *ArcMatch*, the first solution is adopted, and it is implemented by the procedure `RefineDomains`.

`PathReduction` generalizes the concept of arc consistency by imposing consistency not at the edge level but at the path level. For each path ω such that $\omega[1] = v_q$, each target vertex in $D(v_q)$ must be the starting vertex of a corresponding path supported by the current state of the domains. Thus, it is trivial to show that `PathReduction` is equivalent to arc-consistency when $lp = 2$, namely when paths composed of only one edge are taken into account.

Note that the propagation is an additional feature with respect to one simple application of arc-consistency. In some situations, propagation is a costly operation whose benefits are not worth the cost. Given a query graph $G_q = (V_q, E_q)$, the cost of each run of arc-consistency is proportional to $|E_q|$ because edges are scanned. By contrast, the cost of `PathReduction` is proportional to the number of query paths, which may grow exponentially with respect to $|E_q|$.

A question arises: if no propagation is performed, is `PathReduction` more powerful than arc-consistency?

In what follows, we refer to $r_{v_q}^{v_t}$ as the operation which removes the target vertex v_t from $D(v_q)$. Given a query graph G_q and a target graph G_t, $R_{AC}(G_q, G_t)$ represents the set of removal operations that are performed by a single run of arc-consistency over the initial domains. $R_{AC}^{i}(G_q, G_t)$ represents the set of removal operations obtained in i runs of arc-consistency. Lastly, $R_{FC}(G_q, G_t, lp)$ represents the set of removal operations that are performed by a single run of `PathReduction` without propagation, namely by discarding the call to the procedure `RefineDomains`.

Theorem 6. *Given a query graph G_q and a target graph G_t, $R_{AC}(G_q, G_t) \subseteq R_{FC}(G_q, G_t, lp)$ if $lp > 2$.*

Proof. If $lp = 2$, $R_{FC}(G_q, G_t, 2)$ equals $R_{AC}(G_q, G_t)$, because the two tech-

niques both scan for edges and produce the same result.

Thus, if no propagation is applied, arc consistency is equivalent to `PathReduction` with $lp = 2$, namely $R_{AC}(G_q, G_t) \subseteq R_{FC}(G_q, G_t, 2)$. However, Theorem 1 states that $R_{FC}(G_q, G_t, lp_1) \subseteq R_{FC}(G_q, G_t, lp_2)$ with $lp_1 < lp_2$. Thus, $R_{AC}(G_q, G_t) \subseteq R_{FC}(G_q, G_t, 2) \subseteq R_{FC}(G_q, G_t, lp > 2)$. □

3.5 Discussion

The domain of a query element such as a vertex is the set of compatible elements in the target graph. Recently, the concept of edge domains has been the focus of increasing interest [63, 84], especially when they are combined with vertex domains into a unified data structure.

Using such a data structure, *ArcMatch* introduces a new technique for reducing domains which exploits the topological relationship of domains, specifically, their correspondence to paths of the query graph. *ArcMatch* also introduces a new technique, called path-based reduction, for reducing the domain graph [22].

Tests on real networks have shown that combining these new approaches with existing techniques effectively reduces running times when the number of distinct labels on vertices and/or edges of the target graphs is high. A slight variant of *ArcMatch*, called *ArcMatch-lt*, performs better when few matches have to be retrieved.

The authors of state-of-the-art algorithms have studied the scalability of their algorithms with respect to the number of target vertices[5, 24, 104, 25, 26, 63]. By contrast, our approach is most sensitive to the number of target labels and the query size.

We have seen that there is a trade-off between computing constraints (which takes time) and matching time (which is reduced by constraints). This raises two questions for future work: (i) Are there lighter-weight constraints

that might still give similar benefits in matching time? (ii) Is there a way to auto-tune the amount of constraint calculation to do?

4

MultiGraphMatch
A subgraph matching algorithm for multigraphs

In this chapter, we present a new algorithm for the SMM problem, called MultiGraphMatch. Our algorithm first indexes both the query and the target using compact data structures, called bit matrices. Indexes are then exploited to efficiently build compatibility domains, i.e. sets of target edges that are matchable with each query edge. MultiGraphMatch also introduces a new scoring scheme based on the cardinalities of compatibility domains to retrieve a heuristically promising order of processing query edges for the matching process. The ordering is performed edge by edge, by considering only target edges in the compatibility domains. The algorithm also defines a set of symmetry breaking conditions on query nodes and edges to filter out redundant matches during the search.

To fully support queries on multigraphs, MultiGraphMatch uses a declarative SQL-like language called CYPHER which allows: (i) the specification of logical conditions on both labels and properties; (ii) the formation of different kinds of outputs from the set of matches found.

The proposed algorithm runs entirely in main memory, in contrast to several graph database systems proposed to query multigraphs. We compare

MultiGraphMatch to other systems supporting queries on multigraphs, such as Sumgra [73] as well as the graph databases Neo4J and Memgraph. Experiments show that MultiGraphMatch is generally faster than the compared systems in both synthetic and real networks and for both sparse and dense queries of different sizes.

The chapter is organized as follows. Section 4.1 briefly describes the main features of the CYPHER query language. Section 4.2 explains each step of the MultiGraphMatch algorithm. Section 4.3 shows experimental results performed on both synthetic and real networks. Finally, in Section 4.4, we conclude the chapter.

4.1 CYPHER Query Language

Queries in MultiGraphMatch are defined using the CYPHER query language [48]. CYPHER was originally integrated within the graph database system Neo4J[1], but in 2015 has become an open language thanks to the openCypher project[2]. It extends the capabilities of the Structured Query Language (SQL), designed for managing and querying data stored in relational databases.

CYPHER language can be used to define query graphs in a multigraph that match specific topological constraints and optionally semantic conditions on node labels, edge types and properties. MultiGraphMatch implements a subset of the CYPHER query language related to the SMM problem. A complete specification of the CYPHER language is available at `https://opencypher.org/resources`.

Here, we briefly summarize the main features of CYPHER used in this project. The example queries in the CYPHER language presented here are taken from the IMDB bipartite multigraph used in our experiments (see Sec-

[1] https://neo4j.com/
[2] https://opencypher.org/

tion 4.3). In IMDB the two classes of nodes are people involved in movie industry (e.g. actors, directors, composers) and movies. Directed edges connect people to movies.

The heart of the CYPHER query language is represented by the following three clauses: `MATCH`, `WHERE`, and `RETURN`. The `MATCH` clause is used to define the query's topology. The `WHERE` clause is used to optionally filter occurrences of a query based on conditions imposed on labels, types or properties. The `RETURN` clause specifies the type of output we want to extract from the set of query's occurrences found.

In the `MATCH` clause the query's topology is defined as a list of edges. Nodes forming an edge must be enclosed in round (parenthetic) brackets, while edges have to be enclosed within square brackets. Nodes and edges can be named and referenced multiple times within the `MATCH` clause (e.g. if they are involved in more than one edge or path). For query nodes (or edges) that are defined for the first time in the `MATCH` clause, we can also specify their labels (or types) by using semicolons. Node or edge properties can be indicated as well, by means of the notation `{p1:v1, p2:v2, ..., pk:vk}`, where `p1, p2, ..., pk` and `v1, v2, ..., vk` are the property names and their corresponding values. Edges are specified by line or curve segments that may or may not have an arrowhead. If the arrowhead is absent, the link is bidirectional.

For instance, the following `MATCH` clause:

```
MATCH (a:director)-[:DIRECTED {year:2007}]->(b:drama),
(c nationality:USA)-[:DIRECTED]->(b)
```

specifies a query in which we are looking for drama movies, referenced as b in the query, that were co-directed in 2007 by two different directors, named a and c, one of which (node c) is American.

The `WHERE` clause is optional and is formed by a list of conditions on labels, types and properties. Conditions are expressed using relational operators (e.g. $<, \leq, >, \geq, !=$) or special string operators, such as `STARTS WITH`, `ENDS WITH` and `CONTAINS`. Logical operators (e.g. `NOT`, `AND`, `OR`) can be used to combine conditions. The following is an example of a CYPHER query with a `WHERE` clause:

```
MATCH (a:producer)-[r:PRODUCED]->(b:adventure)
WHERE a.surname STARTS WITH 'Sp' AND r.year >= 2006
```

In this case, we are looking for producers who produced adventure movies in 2006 or later and whose last name starts with 'Sp' (e.g. Steven Spielberg). Note that all nodes or edges for which we want to impose conditions must be previously referenced in the `MATCH` clause.

The `RETURN` clause is always mandatory. The information returned by the clause can be either the number of occurrences found (using the `count()` function) or a table containing the values of labels (using the function `labels()`), types (using the function `type()`) or properties of one or more nodes or edges involved in the occurrences found, provided that the latter have been previously referenced in the `MATCH` clause. A list of nodes and edges forming each occurrence can be obtained using functions `nodes()` and `relationships()`. The set of a query's occurrences to be returned can be limited to the first k occurrences using `LIMIT k`. For instance, in the query:

```
MATCH (a:writer)-[r:WROTE]->(b:thriller)
RETURN a.surname, b.name, r.year LIMIT 10
```

we look for writers of thriller movies and output the writer's surname, the movie's name and the year of writing. In addition, we are limiting our search

to the first 10 occurrences found.

4.2 Description of MultiGraphMatch

In this section, we provide the details of MultiGraphMatch algorithm, which has been implemented in the Java language. MultiGraphMatch takes as input a target and a query written in CYPHER language.

Before matching the query, MultiGraphMatch creates different data structures to index the whole target network into main memory. Then, the algorithm performs the following steps:

a) Computation of symmetry breaking conditions for query nodes and edges;

b) Computation of compatibility domains;

c) Ordering of processing query edges for the matching;

d) Matching query with target.

In the next subsections, we detail each of these steps. For ease of explanation, we will consider queries with no `WHERE` clause. In the last subsection, we will illustrate how to adapt MultiGraphMatch to handle logical conditions expressed by the `WHERE` clause. For the description of the algorithm, we refer to a query $Q = (V_Q, E_Q, \mathcal{L}_Q, \sigma_Q, \mathcal{T}_Q, \pi_Q, \mathcal{A}_Q, \mathcal{B}_Q)$ with k nodes and a target $T = (V_T, E_T, \mathcal{L}_T, \sigma_T, \mathcal{T}_T, \pi_T, \mathcal{A}_T, \mathcal{B}_T)$ with n nodes.[3] In order to explain each step, we will also refer to the toy example query Q and target T of Figure 2.5 on page 42.

[3] V_Q and V_T represent the sets of nodes, E_Q and E_T are the sets of edges, $\mathcal{L}_Q$ and $\mathcal{L}_T$ denote the sets of node labels, σ_Q and σ_T are node label functions, $\mathcal{T}_Q$ and $\mathcal{T}_T$ are the sets of edge types, π_Q and π_T are edge type functions, $\mathcal{A}_Q$ and $\mathcal{A}_T$ are node property functions, and $\mathcal{B}_Q$ and $\mathcal{B}_T$ are edge property functions, respectively, for the query and target graphs.

4.2.1 Indexing of target network

As a preprocessing step, the target is read into main memory and several data structures are created to efficiently index it and speed up the next phases of the algorithm, in particular the computation of compatibility domains and the matching process.

The four data structures created are:

a) Node labels graph;

b) Edge types map;

c) Edge properties table;

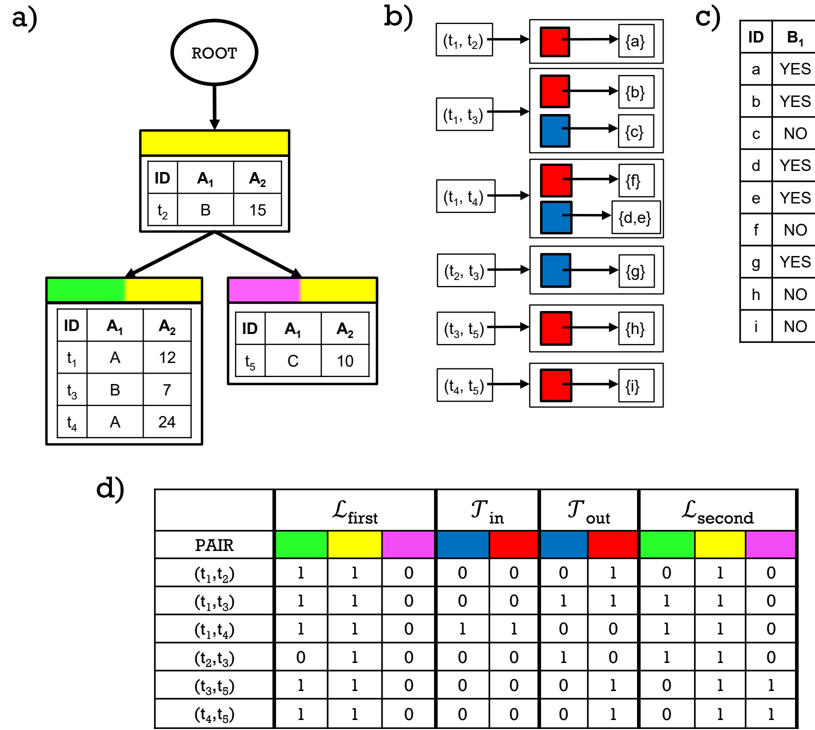
d) Bit signature matrix.

An example of computation of these indexing data structures is shown in Figure 4.1 on the next page for the target T of Figure 2.5 on page 42.

The node labels graph (Figure 4.1a) is a labeled graph where each vertex[4] contains a table with the ids of nodes having a specific set of labels and their relative properties (in the example of Figure 4.1 the target graph has three different labels). A directed edge connects vertex V_1 to vertex V_2 iff the set of labels of nodes in V_1 is a subset of the set of labels of nodes in V_2. In the worst case, the node label graph has a number of vertices K which is exponential in the number of distinct node labels in the target (i.e. the power set of all labels). However, K is upper bounded by the number of target nodes.

The edge types map (Figure 4.1b) is a two-level hash map. The outer level associates an ordered pair of nodes (t_i, t_j) to an inner local hash map where each key is an edge type X and the corresponding value is the list of all edges linking t_i to t_j and having type X.

[4] To avoid confusion, we use here the term "vertex" to denote a node of the node labels graph and the term "node" to indicate a query node or a target node.



ID	B_1
a	YES
b	YES
c	NO
d	YES
e	YES
f	NO
g	YES
h	NO
i	NO

	$\mathcal{L}_{first}$			$\mathcal{T}_{in}$		$\mathcal{T}_{out}$		$\mathcal{L}_{second}$		
PAIR										
(t_1,t_2)	1	1	0	0	0	0	1	0	1	0
(t_1,t_3)	1	1	0	0	0	1	1	1	1	0
(t_1,t_4)	1	1	0	1	1	0	0	1	1	0
(t_2,t_3)	0	1	0	0	0	1	0	1	1	0
(t_3,t_5)	1	1	0	0	0	0	1	0	1	1
(t_4,t_5)	1	1	0	0	0	0	1	0	1	1

Figure 4.1: Indexing data structures associated with the target T in Figure 2.5 on page 42: a) *Node labels graph* where each vertex contains a table with node IDs having a specific set of labels and properties. Directed edges represent subset relationships between label sets of connected nodes; b) *Edge types map*, a two-level hash map where the outer map associates an ordered pair of nodes to an inner map with edge types as keys and lists of edges as values; c) *Edge properties table*, which lists each edge along with its properties; d) *Bit signature matrix*, where each row represents a pair of connected nodes in the target, and their bit signature is a concatenation of substrings representing the labels of both nodes and the types of edges connecting them.

The edge properties table (Figure 4.1c) is a table listing for each edge its properties.

The bit signature matrix (Figure 4.1d) is a matrix of bits where each row R contains, for a given pair of connected nodes (t_i, t_j) in the target (with $id(t_i) < id(t_j)$), a string of bits, called the *signature* of row R and denoted as $Sign(R)$. The signature is a concatenation of four substrings of bits indicating the labels of t_i and t_j and the types of all edges connecting the two nodes in both directions.

Formally, suppose that there are h node labels and p edge labels in the target graph and both target's labels and types are arbitrarily ordered. The bit signature of row R associated to the pair (t_i, t_j) is a string of bits $\mathcal{L}_{first} \cdot \mathcal{T}_{in} \cdot \mathcal{T}_{out} \cdot \mathcal{L}_{second}$, where $\cdot$ is the string concatenation operator and:

- $\mathcal{L}_{first}$ is a substring of bits $b_{11}b_{12} \cdots b_{1h}$, where $b_{1k} = 1$ iff node t_i contains the k-th label in the ordered set of node labels and $b_{1k} = 0$ otherwise;

- $\mathcal{T}_{in}$ is a substring of bits $b_{21}b_{22} \cdots b_{2p}$, where $b_{2k} = 1$ iff there is at least one directed edge (t_j, t_i) having the k-th type in the ordered set of edge types and $b_{2k} = 0$ otherwise;

- $\mathcal{T}_{out}$ is a substring of bits $b_{31}b_{32} \cdots b_{3p}$, where $b_{3k} = 1$ iff there is at least one directed edge (t_i, t_j) having the k-th type in the ordered set of edge types and $b_{3k} = 0$ otherwise;

- $\mathcal{L}_{second}$ is a substring of bits $b_{41}b_{42} \cdots b_{4h}$, where $b_{4k} = 1$ iff node t_j contains the k-th label in the ordered set of node labels and $b_{4k} = 0$ otherwise;

Intuitively, the bit signature matrix is a sort of one-hot encoding scheme of the node labels and edge types associated to each pair of connected nodes in the target. Note that the bit signature matrix records only the presence

or absence of edges with a specific type and direction linking two nodes and not the number of such edges. However, the bit signature matrix's purpose is to filter target candidate nodes for matching during the computation of compatibility domains in the next steps, so this data structure represents a good compromise between memory used and filtering capabilities.

4.2.2 Computation of symmetry breaking conditions

Several mappings in the SMM problem may in fact correspond to the same occurrence in the target.

Figure 4.2 depicts two toy examples with the same target T of Figure 2.5 on page 42 and two queries Q' and Q''. In the query Q' (Figure 4.2a), nodes q_2 and q_3 have the same sets of labels and are both connected to q_1 with an edge having the same direction and the same type. They can be considered equivalent and either can be mapped to target node t_2 and the other to target node t_3. This results in two distinct mappings corresponding to the same occurrence in T. Therefore, one of them should be excluded from the set of solutions. Likewise, in the query Q'' (Figure 4.2b), edges x and y have the same type, the same direction and connect the same pair of nodes. So, they can be indifferently mapped to target edges d and e. The resulting mappings correspond to the same occurrence and one of them can be safely discarded.

To avoid redundancies, MultiGraphMatch computes *symmetry breaking conditions* on query nodes and edges. In the literature, breaking conditions have been mainly used in motif search [58, 121] and subgraph matching [63, 83] on simple graphs. Our novel contribution is to extend the concept of breaking conditions to the SMM problem.

As a short review, symmetry breaking conditions are related to the concepts of automorphisms and orbits of a multigraph.

An automorphism is a permutation of nodes and edges of a multigraph

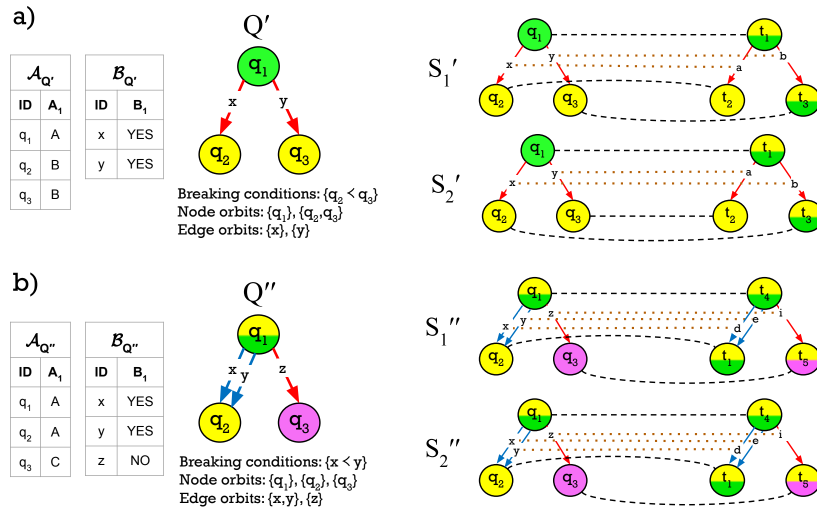


Figure 4.2: Example of symmetry breaking condition on nodes and edges and application of conditions on matching queries Q' and Q'' with the target multigraph T of Figure 2.5 on page 42. a) In query Q' nodes q_2 and q_3 are in the same orbit, thus yielding the breaking condition $q_2 \prec q_3$. Assuming that $t_2 < t_3$ in T, after the application of such condition on T, solution S_2' is discarded. b) In query Q'' edges x and y are in the same orbit, thus yielding the breaking condition $x \prec y$. Assuming that $d < e$ in T, after the application of such condition on T, solution S_2'' is discarded.

that preserves the structure of the graph, the labels and the properties of its nodes and the types and the properties of its edges. Automorphism is an equivalence relation that induces a partition of the sets of multigraph's nodes and edges, respectively, into equivalence classes, called *orbits*. In other words, nodes (or edges) in the same orbit can be permuted without affecting the structure or the semantics of the multigraph. Symmetry breaking conditions are inequalities of the form $a \prec b$ that impose a relative order between two query nodes (or two query edges) belonging to the same orbit.

In the query graph Q' of Figure 4.2a) nodes q_2 and q_3 are in the same orbit, therefore we can define a symmetry breaking condition $q_2 \prec q_3$. Similarly, in Figure 4.2b) edges x and y belong to the same orbit, so we can define a breaking condition $x \prec y$.

Computation of symmetry breaking conditions is done by extending the method presented in [121] for simple unlabeled graphs. First, a query's automorphisms and orbits are computed using the NAUTY algorithm [105]. Next, breaking conditions are extracted from equivalence classes with at least two nodes or edges, always starting from the element with smallest id.

Symmetry breaking conditions are applied to target nodes and edges during the matching process to discard redundant occurrences. Specifically, let q and q' be two query nodes, t and t' be two target nodes and f be a node mapping function which is a solution to the SMM problem, with $f(q) = t$ and $f(q') = t'$. If $q \prec q'$, then q and q' are in the same orbit, so either can be mapped to t and the other to t'. As a result, there must exist another node mapping function f' that yields a second solution to the SMM problem, with $f'(q) = t'$ and $f'(q') = t$. The two solutions produce the same occurrence and in one of them $id(t) < id(t')$, while in the other one $id(t) > id(t')$. To discard one of the two redundant solutions, we simply require that $id(t) < id(t')$. Similar reasoning applies to edge breaking conditions.

In Figure 4.2a, assuming $t_2 < t_3$, the application of breaking condition $q_2 \prec q_3$ results in including solution S_1' and discarding solution S_2'. In Figure 4.2b, assuming $d < e$, the application of breaking condition $x \prec y$ results in including solution S_1'' and discarding solution S_2''.

4.2.3 Computation of compatibility domains

Before matching the query with the target, MultiGraphMatch builds a set of matchable target edges based on compatibility domains for all pairs of connected query nodes. Only matchable edges will be considered as candidates for a match during the search.

Given a pair of connected query nodes (q_i, q_j), with $id(q_i) < id(q_j)$, the *compatibility domain* of (q_i, q_j), $Dom(q_i, q_j)$, is a set of pairs of connected target nodes (t_i, t_j) which are selected according to a set of conditions involving: i) the direction and the type of all the edges connecting q_i to q_j; ii) the labels of q_i and q_j; iii) the type-dependent in-degrees and out-degrees of q_i and q_j. If all conditions are satisfied, (t_i, t_j) is said to be *compatible* to (q_i, q_j) and the pair (t_i, t_j) is added to $Dom(q_i, q_j)$. Node and edge properties are not used to build compatibility domains.

Conditions based on the direction and types of query edges and the labels of query nodes are checked by using the bit signature matrix data structure introduced in Subsection 4.2.1. Specifically, the algorithm computes the bit signature matrix of query Q, $\mathcal{BS}(Q)$, and then matches the rows of $\mathcal{BS}(\mathcal{Q})$ with the rows of the bit signature matrix of the target T, $\mathcal{BS}(T)$.

Each row of $\mathcal{BS}(Q)$ contains the bit signature of a specific pair of connected query nodes. The structure of $\mathcal{BS}(Q)$ is the same as the bit signature matrix of the target (see Subsection 4.2.1), with node labels and edge types following the same order chosen to build $\mathcal{BS}(T)$. The only difference is that now for the query graph, the bit signature matrix includes the edges when considering q_i as source and q_j as destination as well as vice versa. This is

done to speed up the computation of domains and to ensure that all compatible pairs are discovered. Figure 4.3 on the following page shows the bit signature matrices $\mathcal{BS}(Q)$ and $\mathcal{BS}(T)$ for the query Q and target T of Figure 2.5 on page 42, respectively.

In order to find all pairs of connected target nodes that match the labels of q_i and q_j and the types and directions of all edges linking q_i and q_j, MultiGraphMatch scans the bit signature matrix of target $\mathcal{BS}(T)$ to look for all rows which contain all the entries of the rows of either pair (q_i, q_j) or pair (q_j, q_i). Changing the order of target and query nodes in their respective pairs corresponds to simply swapping node labels and the direction of the edges connecting them both in the query and in the target, preserving their match. Therefore, if the row of (t_i, t_j) contains all the entries of the row of (q_j, q_i), this is equivalent to saying that (t_j, t_i) matches (q_i, q_j). Mathematically, checking if a row R contains all the entries of a row R' corresponds to verifying if $Sign(R) \wedge Sign(R') = Sign(R)$, where $\wedge$ is the bitwise operator.

If the row of a target pair (t_i, t_j) matches either the row of (q_i, q_j) or the row of (q_j, q_i), type-dependent in-degrees and out-degrees of corresponding query and target nodes are compared. In particular, suppose that $q_i < q_j$ and the row of (t_i, t_j) matches the row of (q_i, q_j). Then, (t_i, t_j) is added to $Dom(q_i, q_j)$ iff $\forall t \in \mathcal{T}$:

1. $t - outDeg(q_i) \leq t - outDeg(t_i)$;
2. $t - inDeg(q_i) \leq t - inDeg(t_i)$;
3. $t - outDeg(q_j) \leq t - outDeg(t_j)$;
4. $t - inDeg(q_j) \leq t - inDeg(t_j)$;

In Figure 4.3 the row corresponding to pair (t_1, t_4) contains all the entries of the row of (q_1, q_2), therefore the two rows match. The row of (t_1, t_3) would also match the row of (q_1, q_2) if t_1 and t_3 were swapped. So, the row of (t_1, t_3)

$\mathcal{BS}$(Q)

	$\mathcal{L}_{first}$			$\mathcal{T}_{in}$		$\mathcal{T}_{out}$		$\mathcal{L}_{second}$		
PAIR										
(q_1,q_2)	1	0	0	1	1	0	0	1	0	0
(q_2,q_1)	1	0	0	0	0	1	1	1	0	0
(q_1,q_3)	1	0	0	0	0	0	1	0	1	0
(q_3,q_1)	0	1	0	0	1	0	0	1	0	0
(q_3,q_4)	0	1	0	0	0	0	1	0	1	0
(q_4,q_3)	0	1	0	0	1	0	0	0	1	0

$\mathcal{BS}$(T)

	$\mathcal{L}_{first}$			$\mathcal{T}_{in}$		$\mathcal{T}_{out}$		$\mathcal{L}_{second}$		
PAIR										
(t_1,t_2)	1	1	0	0	0	0	1	0	1	0
(t_1,t_3)	1	1	0	0	0	1	1	1	1	0
(t_1,t_4)	1	1	0	1	1	0	0	1	1	0
(t_2,t_3)	0	1	0	0	0	1	0	1	1	0
(t_3,t_5)	1	1	0	0	0	0	1	0	1	1
(t_4,t_5)	1	1	0	0	0	0	1	0	1	1

Figure 4.3: Example of computation of compatibility domains for the query Q and target T of Figure 2.5 on page 42. $\mathcal{BS}(Q)$ and $\mathcal{BS}(T)$ are the bit signature matrices of Q and T, respectively. The row of pair (t_1, t_4) contains all the features of the row associated to (q_1, q_2), so (t_1, t_4) matches (q_1, q_2). Target pair (t_1, t_3) does not match (q_1, q_2) but matches (q_2, q_1). This is equivalent to say that (t_3, t_1) matches (q_1, q_2). In addition, target nodes of both pairs (t_1, t_4) and (t_3, t_1) have at least the same number of type-dependent in-degress and out-degrees of corresponding query nodes q_1 and q_2. No more pairs of target nodes match either (q_1, q_2) or (q_2, q_1), because they miss a pair of blue and red edges in either ingoing or outgoing direction. In the case that $id(q_1) < id(q_2)$, we can build the compatibility domain of (q_1, q_2), given by $Dom(q_1, q_2) = \{(t_1, t_4), (t_3, t_1)\}$.

matches the row of (q_2, q_1). Since both t_1 and q_1 have an outgoing red edge and t_4 have at least one red outgoing edge and one blue outgoing edge as does query node q_2, we can conclude that (t_1, t_3) is compatible to (q_1, q_2). By comparing the type-dependent in-degrees and out-degrees of nodes q_2 and t_1 and nodes q_1 and t_3, we can state that (t_3, t_1) is compatible to (q_1, q_2) as well. There are no other pairs of target nodes that match either (q_1, q_2) or (q_2, q_1). Provided that $id(q_1) < id(q_2)$, we can build a compatibility domain for the pair (q_1, q_2), given by $Dom(q_1, q_2) = \{(t_1, t_4), (t_3, t_1)\}$.

4.2.4 Determining the processing order of query edges

To establish the order in which query edges have to be processed during the matching, MultiGraphMatch exploits information about compatibility domains. As shown in [20], the processing order of query nodes and edges can have a large impact on the running time of subgraph matching algorithms. The idea is to start by matching the densest parts of the query graph, while also taking into account the cardinalities of compatibility domains and the number of matching constraint based on the pairs that are already in the ordering.

Starting from an empty *sequential ordering* $\mathcal{O}$, MultiGraphMatch iteratively chooses a new pair of connected query nodes (q_i, q_j) (with $id(q_i) < id(q_j)$) and add all edges linking q_i and q_j to $\mathcal{O}$ in arbitrary order. The process ends when all query edges have been added to $\mathcal{O}$.

At each step of the ordering, the pair of query nodes is chosen according to a priority score. Given a pair (q_i, q_j) with $id(q_i) < id(q_j)$ the priority of (q_i, q_j) is a pair (CF, Sc), where:

- CF is the constraint factor, defined as the number of nodes in (q_i, q_j) that belong to pairs already present in the current ordering $\mathcal{O}$;
- Sc is a score which depends on the the density of the neighborhood of

(q_i, q_j) and the size of its compatibility domain.

CF can take values 0, 1 or 2. In the case of $CF = 1$, we call the node that does not yet belong to any pair already in the ordering *free node*. The score Sc is defined differently for different values of CF:

$$Sc(q_i, q_j) = \begin{cases} \frac{totDeg(q_i) \times totDeg(q_j) \times Jacc(q_i, q_j)}{|Dom(q_i, q_j)|} & \text{if } CF(q_i, q_j) = 0 \\ \frac{totDeg(freenode(q_i, q_j)) \times Jacc(q_i, q_j)}{|Dom(q_i, q_j)|} & \text{if } CF(q_i, q_j) = 1 \\ \frac{1}{|Dom(q_i, q_j)|} & \text{if } CF(q_i, q_j) = 2 \end{cases} \tag{4.1}$$

where $totDeg(q_i)$ is the total degree (i.e. out-degree + in-degree) of node q_i, $freenode(q_i, q_j)$ is the free node of pair (q_i, q_j) and $Jacc(q_i, q_j)$ is the neighbor Jaccard similarity between q_i and q_j, i.e. the fraction of neighbors in common between q_i and q_j:

$$Jacc(q_i, q_j) = \frac{|Neigh(q_i) \cap Neigh(q_j)|}{|Neigh(q_i) \cup Neigh(q_j)|} \tag{4.2}$$

Note that, except for the case $CF = 2$, Sc combines the local density of a pair's neighborhood and the pair's domain size. The score expression reflects a tradeoff between the two. For example, it may be better to start from edges with very small domains even though they are not in a dense core region of the query. Thus, Sc represents a trade-off between the two measures.

To choose the next pair to be added to the current ordering, MultiGraphMatch first computes the priority of all pairs that are not yet in the current ordering $\mathcal{O}$. Then, candidate pairs are compared lexicographically: a pair (u, v) has higher priority than pair (u', v') iff either $CF(u, v) > CF(u', v')$ or $CF(u, v) = CF(u', v')$ and $Sc(u, v) > Sc(u', v')$. The pair with highest priority is added to the current ordering $\mathcal{O}$ and possible ties are solved arbitrarily.

Thus, the algorithm gives higher priority to pairs having $CF = 2$. This is done to extend a partial match from edges having the maximum number

of constraints as soon as possible. In fact, if both endpoints of a query edge e have already been matched, there are likely very few candidate target edges to scan while matching e.

4.2.5 Matching process

Following the previously defined ordering of query edges, MultiGraphMatch performs matching to find occurrences of the query within the target. The matching process is outlined in Algorithm 9.

Matching is done by building a node mapping function $f : V_Q \rightarrow V_T$ and an edge mapping function $g : E_Q \rightarrow E_T$. The list of candidates to scan during the search is stored in variable $Cand$.

The matching process starts with the first edge $e_Q = (q, q')$ in the ordering (line 1). An initial list of candidates $Cand(e)$ is calculated (line 2) and the list is scanned starting from the first element. A given candidate $e_T = (t, t')$ can be matched with e_Q iff e_T has not been mapped yet (line 11). Whenever a new match is found, the mapping functions f and g are updated (lines 12-16). If e_Q was the last query edge to match, the occurrence Occ resulting from mappings f and g is built and added to the list of matches found (lines 18-19). Then, the search goes on with the next candidate (line 20). If e_Q was not the last query edge to match, we continue the search with the next query edge to match (lines 22-23) and find the set of candidates for such edge (line 24). If candidate e_T does not match e_Q, the algorithm just skips to the next candidate (line 27). When all candidates for e_Q have been examined (line 5), MultiGraphMatch performs backtracking, i.e. restores mappings to the previous values (line 6) and goes back to the previous query edge (lines 7-8). Backtracking implies removing the mapping for the last matched query edge e_q and optionally the mapping for one or both nodes of e_q. To guarantee that every time we do backtracking the search continues from last examined candidate for the previous query edge, MultiGraphMatch uses a set of list

Algorithm 9 SubgraphMatching($Q, T, \mathcal{BC}_N, \mathcal{BC}_E, Dom, \mathcal{O}$)

1: **procedure** SubgraphMatching($Q, T, \mathcal{BC}_N, \mathcal{BC}_E, Dom, \mathcal{O}$)
2: $f(u) := undefined$ for all $u \in V_Q$ ▷ Current node mapping
3: $g(e) := undefined$ for all $e \in E_Q$ ▷ Current edge mapping
4: $Cand(e) := undefined$ for all $e \in E_Q$ ▷ Lists of candidate target edges
5: $CandIndex(e) := 1$ for all $e \in E_Q$ ▷ Iterators for candidate lists
6: $e_Q = (q, q') := \mathcal{O}[1]$
7: $Cand(e_Q) :=$ FindCandidates($Q, e_Q, T, \mathcal{BC}_N, \mathcal{BC}_E, Dom, f, g$)
8: $i := 1$
9: **while** $i > 0$ **do**
10: **if** $CandIndex(e_Q) > |Cand(e_Q)|$ **then**
11: RestoreInfo(f, g)
12: $i := i - 1$
13: $e := \mathcal{O}[i]$
14: **else**
15: $e_T = (t, t') := Cand(e_Q)[CandIndex(e_Q)]$
16: **if** $\forall e_Q \in E_Q : g(e_Q) \neq e_T$ **then**
17: **if** $f(q) = undefined$ **then**
18: $f(q) := t$
19: **end if**
20: **if** $f(q') = undefined$ **then**
21: $f(q') := t'$
22: **end if**
23: $g(e_Q) := e_T$
24: **if** $i = |E_Q|$ **then**
25: $Occ :=$ BuildOccurrence(f, g)
26: $Occs := Occs \cup \{Occ\}$
27: $CandIndex(e) := CandIndex(e) + 1$
28: **else**
29: $i := i + 1$
30: $e_Q = (q, q') := \mathcal{O}[i]$
31: $Cand(e_Q) :=$ FindCandidates($Q, e_Q, T, \mathcal{BC}_N, \mathcal{BC}_E, Dom, f, g$)
32: $CandIndex(e_Q) := 1$
33: **end if**
34: **else**
35: $CandIndex(e_Q) := CandIndex(e_Q) + 1$
36: **end if**
37: **end if**
38: **end while**
39: **return** $Occs$
40: **end procedure**

iterators $CandIndex$, one for each query edge. $CandIndex(e_Q)$ contains the position of the last examined candidate in $Cand(e_Q)$. Every time we pass to the next candidate, the corresponding iterator is incremented (lines 20 and 27). The search ends when no more candidates are available for the first query edge. At the end of the process, MultiGraphMatch returns the list of all occurrences found (line 28).

The procedure used to find the set of candidates $Cand(e_Q)$ for a query edge $e_Q = (q, q')$ is detailed in Algorithm 10.

The content of $Cand(e_Q)$ depends on whether q and/or q' have already been mapped or not. This leads to the four cases illustrated in the pseudocode of Algorithm 10. In all these cases, before adding a target edge e_T to $Cand(e_Q)$, we must ensure that e_T is in the compatibility domain of pair (q, q') and matches the type and the properties of e_Q. If one or both endpoints of e_T are still unmapped (lines 1-4, lines 11-16 and lines 18-22) we also need to check that such nodes match the labels and the properties of corresponding e_Q's endpoints. Moreover, except for the matching of the first target edge (lines 1-4), the algorithm also verifies if adding e_T to the match violates one or more symmetry breaking conditions involving e_T or one of its endpoints not yet mapped. For example, if only q has been mapped (lines 11-16), then $Cand(e_Q)$ is the set of all possible target edges $e_T = (f(q), t')$ in the compatibility domain of pair (q, q') such that: i) e_T has the same type of e_Q and contains all its properties, ii) t' contains all the properties of q' and iii) symmetry breaking conditions on t' are satisfied.

Checking breaking conditions for a target node t candidate to be mapped with a query node q (lines 14 and 20) involves verifying if $id(f(q')) < id(t)$ for all query nodes q' such that i) there exists a breaking condition $q' \prec q$ and ii) q' has been already mapped. Likewise, checking breaking conditions for a target edge e_T candidate to be mapped with a query edge e_Q (line 8) implies verifying if $id(g(e'_Q)) < id(e_T)$ for all query edges e'_Q such that i) there exists

Algorithm 10 FindCandidates$(Q, e, T, \mathcal{BC}_N, \mathcal{BC}_E, Dom, f, g)$

```
1: procedure FindCandidates(Q, e, T, BC_N, BC_E, Dom, f, g)
2:     Input: Q = (V_Q, E_Q, L_Q, σ_Q, T_Q, π_Q, A_Q, B_Q): query, e_Q = (q, q'): query edge
3:            T = (V_T, E_T, L_T, σ_T, T_T, π_T, A_T, B_T): target
4:            BC_N: set of breaking conditions on nodes, BC_E: set of breaking conditions on edges
5:            Dom: compatibility domains, f: current node mapping, g: current edge mapping
6:     Output: Cand: list of candidates
7:     if f(q) = undefined and f(q') = undefined then
8:         for all e_T = (t, t') ∈ Dom(q, q') do
9:             if π_T(e_T) = π_Q(e_Q) and A_Q(q) ⊆ A_T(t) and A_Q(q') ⊆ A_T(t') and B_Q(e_Q) ⊆ B_T(e_T)
   then
10:                Cand(e) := Cand(e) ∪ {e_T}
11:            end if
12:        end for
13:    else if f(q) ≠ undefined and f(q') ≠ undefined then
14:        for all e_T = (f(q), f(q')) ∈ E_T do
15:            if π_T(e_T) = π_Q(e_Q) and B_Q(e_Q) ⊆ B_T(e_T) then
16:                condCheck := CheckEdgeBreakCond(BC_E, e_Q, e_T, g)
17:                if condCheck = true then
18:                    Cand(e) := Cand(e) ∪ {e_T}
19:                end if
20:            end if
21:        end for
22:    else if f(q) ≠ undefined then
23:        for all e_T = (f(q), t') ∈ Dom(q, q') do
24:            if π_T(e_T) = π_Q(e_Q) and A_Q(q') ⊆ A_T(t') and B_Q(e_Q) ⊆ B_T(e_T) then
25:                condCheck := CheckNodeBreakCond(BC_N, q', t', f)
26:                if condCheck = true then
27:                    Cand(e) := Cand(e) ∪ {e_T}
28:                end if
29:            end if
30:        end for
31:    else
32:        for all e_T = (t, f(q')) ∈ Dom(q, q') do
33:            if π_T(e_T) = π_Q(e_Q) and A_Q(q) ⊆ A_T(t) and B_Q(e_Q) ⊆ B_T(e_T) then
34:                condCheck := CheckNodeBreakCond(BC_N, q, t, f)
35:                if condCheck = true then
36:                    Cand(e) := Cand(e) ∪ {e_T}
37:                end if
38:            end if
39:        end for
40:    end if
41:    return Cand
42: end procedure
```

a breaking condition $e'_Q \prec e_Q$ and ii) e'_Q has been already mapped.

4.2.6 Handling queries with WHERE clause

MultiGraphMatch can be extended to handle conditions on labels, types and properties expressed by the WHERE clause in CYPHER language.

Before processing a query with a WHERE clause, the logical proposition P expressed by the clause is first transformed into Disjunctive Normal Form (DNF), so that P will have the following standard representation:

$$P = (C_{11} \wedge C_{12} \cdots C_{1k_1}) \vee (C_{21} \wedge C_{22} \cdots C_{2k_2}) \vee \cdots \vee (C_{m1} \wedge C_{m2} \cdots C_{mk_m})$$

where C_{ij} are boolean conditions involving node labels, edge types, node properties or edge properties, optionally preceded by the NOT operator that, if present, negates the value of the condition.

Conditions of the logical proposition P in DNF form are then processed by MultiGraphMatch depending on the structure of P.

Let us first consider the case that P does not contain OR operators. This means that P is formed by a chain of boolean conditions linked by an AND operator, so they must all be satisfied in a matching subgraph.

A boolean condition of the form $A.x\ op\ c$, where A is a node or an edge, x is a label, a type or a property, op is a relational operator and c is a constant can be used as additional constraints to build compatibility domains of query nodes. The net result will be to allow only target edges that satisfy the conditions.

By contrast, conditions of the form $A.x\ op\ B.y$, where B is a second node or edge and y is a label, a type or a property can be verified only at matching time, during the search for candidates (Algorithm 10). In particular, the condition holds when either A or B (or both) are matched. Verification consists of checking whether the type or the properties of candidate edges

(and optionally their endpoints) match the type and the properties of the matching query edge (lines 3, 7, 13, 19).

If the logical proposition P contains OR operators, MultiGraphMatch first splits the query into several queries having the same MATCH clause and different WHERE clauses. More formally, let Q a query and $P = P_1 \vee P_2 \vee \cdots \vee P_k$ the logical proposition expressed by the WHERE clause of Q, where $P_1, P_2, ..., P_k$ are propositions containing only AND operators. MultiGraphMatch creates k different queries $Q_1, Q_2, ..., Q_k$, where query Q_i (with $1 \leq i \leq k$) has the same MATCH clause as Q and the WHERE clause of Q_i has the proposition P_i. Each query Q_i contains only propositions linked by AND operators, so it can be solved as in the case where there are no OR operators. Finally, the union of the sets of solutions returned by each query is computed to find set of solutions for Q.

4.2.7 Complexity analysis

In this section we analyze the time and space complexity of MultiGraphMatch.

We start by evaluating the time complexity of each step of the algorithm. Let $Q = (V_Q, E_Q, \mathcal{L}_Q, \sigma_Q, \mathcal{T}_Q, \pi_Q, \mathcal{A}_Q, \mathcal{B}_Q)$ the query and $T = (V_T, E_T, \mathcal{L}_T, \sigma_T, \mathcal{T}_T, \pi_T, \mathcal{A}_T, \mathcal{B}_T)$ the target.

Indexing requires scanning all nodes and edges of the target multigraph. To build the node label graph, nodes are read one at the time and inserted into the graph, following a path from the root to one of the graph vertices. If $|\mathcal{L}_T| = l_T$, in the worst case, the node to insert has l_T labels and the length of the traversed path is l_T. Therefore, the time required to build the node label graph on the target graph is $O(n_T \times l_T)$, where $n_T = |V_T|$. The edge types map and the edge properties table are built by reading edges and associated type and properties one at the time. If the two data structures are implemented using hash maps and hash table, accessing and updating

them can be done in constant time, so building the edge types map and the edge properties table requires time $O(e_T)$, where $e_T = |E_T|$. To create the bit matrix we need to scan all target nodes and edges, with relative labels and types, so the time required to build the matrix is $O(n_T + e_T)$.

Computation of symmetry breaking conditions is related to the calculation of query's automorphisms, which is at least as difficult as solving graph isomorphism [102], and requires $O(2^{n_Q})$ time, where $n_Q = |V_Q|$, the number of nodes in the query graph.

Compatibility domains are computed by matching the rows of the bit matrices associated to the target and the query and comparing the type-dependent in- and out-degrees of endpoints of the matched pairs of nodes. Building the bit matrix for the query requires $O(e_Q)$, where $e_Q = |E_Q|$. The worst case holds when there is at most one edge between pairs of nodes in both the query and the target. In that case, the bit matrices of the query and the target have e_Q and e_T rows, respectively. Therefore, matching the rows of the two bit matrices costs $O(e_Q \times e_T)$. For each pair of matched rows, checking type-dependent degrees costs $O(t_T)$, where $t_T = |\mathcal{T}_T|$. So, the total time required to compute compatibility domains is $O(e_Q \times e_T \times t_T)$.

Calculating the processing order of query edges for the match requires computing at each step the CF and the Sc scores for every pair of connected query nodes that have not been added yet to the partial ordering. The number of connected pairs is $O(e_Q)$, so the two scores are computed $O(e_Q^2)$ times. In the worst case, calculation of the Sc score for a pair (q_i, q_j) also requires the Jaccard similarity between the neighborhoods of q_i and q_j. Since a query node can have at most $n_Q - 1$ neighbors, calculating Sc costs $O(n_Q)$. Therefore, the total time required to compute the processing order is $O(e_Q^2 \times n_Q)$.

The computational complexity of the matching step mainly depends on the total number of candidate pairs of query and target edges that are exam-

ined during the process. In the worst case, the compatibility domain of each query edge has e_T elements. Therefore, the set of candidate nodes for the initial query edge has at most e_T edges. Once a new pair of edges has been matched, the set of candidate target edges for the next query edge in the ordering has at most $d-1$ edges, where d is the maximum target node degree. In fact, except for the initial set of candidates, the next sets of candidates contain target edges having at least one node in common with already matched edges. By summing up, the total number of examined candidate pairs is at most $e_T + e_T(d-1) + e_T(d-1)^2 + ... + e_T(d-1)^{e_Q}$ which is bounded by $O(e_T \times d^{e_Q})$. For a given candidate pair of edges, breaking conditions are checked in $O(n_Q)$ time, while comparing node and edge properties of the two edges costs $O(p_{n_T} + p_{e_T})$, where $p_{n_T} = |\mathcal{A}_T|$ and $p_{e_T} = |\mathcal{B}_T|$. If the algorithm finds a match with a candidate we need to update the current partial match and the latter operation requires constant time. On the other hand performing a step of backtracking requires also constant time operations. So, the total time required to perform matching is $O(e_T \times d^{e_Q} \times (n_Q + pn_T + pe_T))$, which is bounded by $O(n_Q e_T d^{e_Q})$. By summing the time complexities of each step of MultiGraphMatch, we get $O(2^{n_Q} + e_Q e_T t_T + {e_Q}^2 n_Q + n_Q e_T d^{e_Q})$. In practice, the query is much smaller than the target, in which case the worst case time complexity of our algorithm is $O(n_Q e_T d^{e_Q})$.

Regarding the spatial complexity of MultiGraphMatch, most of the memory is used to store the four indexing data structures on the target. In the node label graph each node with associated properties is stored only once, so this data structure costs $O(n_T \times pn_T)$. In the edge type map each edge is stored in only one of the inner hash maps, so this index requires $O(e_T)$ space. The edge properties table costs $O(e_T \times pe_T)$ space. The bit matrix of the target contains a row for each ordered pair of connected nodes. The number of such pairs is at most $O(e_T)$, while the number of columns of the bit matrix is $2l_T + 2$. So, the bit matrix costs $O(e_T \times l_T)$ space. Summing

up, the total space required to store the indexing data structures for the target is $O(n_T \times pn_T + e_T \times (pe_T + l_T))$. In a large network, we can reasonably assume that the number of labels and properties is negligible compared with the number of nodes and edges of the target, so the space required is $O(e_T)$. Additional data structures used by the algorithm in the next steps include the set of symmetry breaking conditions for the query ($O(n_Q^2 + e_Q^2)$ space), the bit matrix for the query ($O(e_Q l_Q)$ space, where $l_Q = |\mathcal{L}_Q|$), the set of compatibility domains for each query edge ($O(e_Q e_T)$ space in the worst case), the node and edge mappings ($O(n_Q)$ and $O(e_Q)$ space, respectively) and the set of candidate target edges for the matching for each query edge (which costs $O(e_Q e_T)$ in the worst case). Therefore, the total spatial complexity of MultiGraphMatch is $O(e_Q \times e_T)$.

4.3 Experimental Results

We performed a series of experiments on both synthetic and real networks to compare the performance of the MultiGraphMatch algorithm with state-of-the-art methods:

Neo4J (version 4.2) (`https://neo4j.com/`), Memgraph (`https://memgraph.com/`) and SumGra [73].

According to recent work [110], Neo4J is the most efficient graph database system. Further Neo4J provides full support for queries in multigraphs, unlike many other systems. In our experiments, we also included Memgraph, because it compared favorably with Neo4J (version 3). Moorman's algorithm [111] was excluded from the comparison because of its high memory requirements even for networks of medium size in our tests. We also excluded FGq_t-Match [137] from our analysis, because we could not obtain the software from the authors.

We performed additional experiments to evaluate i) the impact of newly

introduced features, such as bit matrix and processing ordering of query edges, on the performance of MultiGraphMatch and ii) the scalability of our algorithm.

All the experiments were performed on an Intel(R) Xeon(R) Gold 6132 CPU 2.60GHz with 400 GB of RAM. MultiGraphMatch has been implemented in Java and is available on GitHub[5], together with the datasets used for the experiments.

4.3.1 Experiments on synthetic networks

Data description

We built a dataset of labeled synthetic networks, generated according to the Barabasi-Albert model [10], implemented within the snap tool [90]. All networks contain 10,000 nodes and 1 million edges. Each node of the networks is associated with a single label, randomly sampled from a set of possible labels according to either a power law or a uniform distribution. An analogous algorithm is used to label each edge of the networks. The number of distinct node labels and the number of distinct edge labels in the synthetic networks is either 2 or 10.

This results in a final set of 8 different synthetic networks:

- UNIFORM(2,2): a network with 2 distinct node labels and 2 distinct edge labels sampled from a uniform distribution;
- UNIFORM(2,10): a network with 2 distinct node labels and 10 distinct edge labels sampled from a uniform distribution;
- UNIFORM(10,2): a network with 10 distinct node labels and 2 distinct edge labels sampled from a uniform distribution;

[5] https://github.com/Anto188bas/MultiGraphMatch.git

- UNIFORM(10,10): a network with 10 distinct node labels and 10 distinct edge labels sampled from a uniform distribution;
- POWERLAW(2,2): a network with 2 distinct node labels and 2 distinct edge labels sampled from a power-law distribution;
- POWERLAW(2,10): a network with 2 distinct node labels and 10 distinct edge labels sampled from a power-law distribution;
- POWERLAW(10,2): a network with 10 distinct node labels and 2 distinct edge labels sampled from a power-law distribution;
- POWERLAW(10,10): a network with 10 distinct node labels and 10 distinct edge labels sampled from a power-law distribution;

Performance comparison

We first compared MultiGraphMatch, SumGra, Memgraph and Neo4j on the dataset of synthetic networks.

To this end, from each network, we randomly extracted 600 query graphs having different number of nodes (from 3 to 8) and densities (from 0.25 to 1). For each algorithm, we set a timeout of 30 minutes for the execution of any single query.

Experimental results are summarized in Table 4.1. For each synthetic network and for each algorithm, the table reports the number of queries completed before the timeout, the mean running time and the 90% confidence interval of the mean time. Reported results refer only to queries completed by at least one algorithm before the timeout, with a running time of 1,800 seconds assigned for each uncompleted query.

Figures 4.4 and 4.5 on page 136 depict boxplots of the distribution of the query-by-query running time differences between MultiGraphMatch and the other compared algorithms in each synthetic network. To build these plots

Network	Algorithm	Completed queries	Mean time	Confidence interval
Uniform(2,2)	MGM	340	**223.83**	[192.07, 259.50]
	Neo4J	267	640.71	[565.50, 721.99]
	Sumgra	338	292.94	[252.03, 337.70]
	Mem	145	1188.98	[1104.87, 1271.69]
Uniform(2,10)	MGM	556	17.62	[7.74, 32.55]
	Neo4J	555	90.86	[69.96, 115.34]
	Sumgra	559	**16.06**	[8.42, 27.05]
	Mem	316	912.05	[843.93, 981.78]
Uniform(10,2)	MGM	583	**21.04**	[11.85, 33.63]
	Neo4J	571	74.79	[52.14, 101.08]
	Sumgra	575	66.85	[46.52, 91.03]
	Mem	313	909.59	[841.84, 978.93]
Uniform(10,10)	MGM	600	**0.03**	[0.02, 0.04]
	Neo4J	600	0.73	[0.64, 0.85]
	Sumgra	600	0.15	[0.09, 0.23]
	Mem	600	68.95	[49.79, 90.97]
PowerLaw(2,2)	MGM	308	**287.76**	[240.48, 336.69]
	Neo4J	206	759.19	[671.98, 851.37]
	Sumgra	293	406.69	[344.86, 468.07]
	Mem	151	1075.21	[984.56, 1166.70]
PowerLaw(2,10)	MGM	564	**38.15**	[23.84, 56.45]
	Neo4J	525	199.42	[160.47, 240.92]
	Sumgra	557	53.14	[35.05, 75.89]
	Mem	280	1001.44	[930.62, 1070.48]
PowerLaw(10,2)	MGM	568	**46.40**	[32.51, 63.98]
	Neo4J	537	172.15	[136.75, 212.65]
	Sumgra	549	124.13	[94.91, 156.71]
	Mem	299	966.42	[896.23, 1035.33]
PowerLaw(10,10)	MGM	600	**1.92**	[0.41, 5.22]
	Neo4J	600	11.17	[4.20, 22.05]
	Sumgra	600	9.63	[2.67, 20.81]
	Mem	600	219.12	[178.57, 261.53]

Table 4.1: Performance of MultiGraphMatch (MGM), Sumgra, Neo4J and Memgraph (Mem) on a dataset of queries run on synthetic networks. The table reports, for each synthetic network and for each algorithm, the number of queries completed before the timeout (30 minutes), the mean running time and the 90% confidence interval. Uncompleted queries are considered to take 30 minutes. MultiGraphMatch is significantly better for nearly all synthetic networks except Uniform(2,10) where Sumgra is slightly faster.

we considered only queries completed by at least one tool before the timeout and we assigned a running time of 1,800 seconds for each uncompleted query. In Table 4.2 on page 137 we also list the median, the 90% confidence interval and the p-value of significance of the running time differences between MultiGraphMatch and the other algorithms. The p-values and confidence intervals were computed non-parametrically [78].

Results clearly show that MultiGraphMatch is the fastest algorithm (with p-value < 0.05) except for the network Uniform(2,10), where Sumgra is slightly faster but with a non-significant p-value (0.35). The advantage of MultiGraphMatch is overall higher in fractal networks. Moreover, except for the network Uniform(2,10), MultiGraphMatch completed the most queries before the timeout.

Ablation Tests

MultiGraphMatch introduces (i) a bit matrix data structure (see Section 4.2.1) for SubMultigraph Matching as well as (ii) a novel ordering scheme for processing query edges (see Section 4.2.4). We ran a series of ablation tests on a subset of synthetic networks to evaluate the impact of each of these features on the performance of our algorithm.

We compared the full MultiGraphMatch (abbreviated as MGM) with three simplified versions of the algorithm: MultiGraphMatch with no Bit Matrix (MGM-NoBM), MultiGraphMatch with Random Ordering (MGM-RO) and MultiGraphMatch with Random Ordering and no Bit Matrix (MGM-NoBM-RO). The comparison was performed on the complete set of queries extracted from the following synthetic networks: Uniform(2,10), Uniform(10,2), PowerLaw(2,10) and PowerLaw(10,2). Also in this case, a timeout of 30 minutes was set for the execution of each query.

Figure 4.6 on page 138 shows boxplots of the running times of the compared versions of MultiGraphMatch on each tested synthetic network.

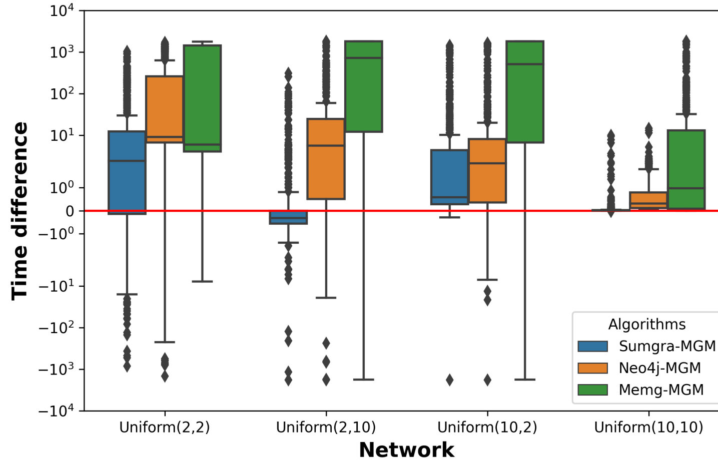


Figure 4.4: Boxplots of the query-by-query running time differences between MultiGraphMatch (MGM) and the other compared tools Sumgra, Neo4J and Memgraph (Mem) for the dataset of queries run on the uniform synthetic networks. If the mean value of another system S lies on the zero line, then its mean is the same as MGM. If above, then S is slower (in the mean) than MGM. The boxplots show that the mean value of MultiGraphMatch is the lowest in every case except for UNIFORM(2,10) where Sumgra enjoys a small but not significant advantage.

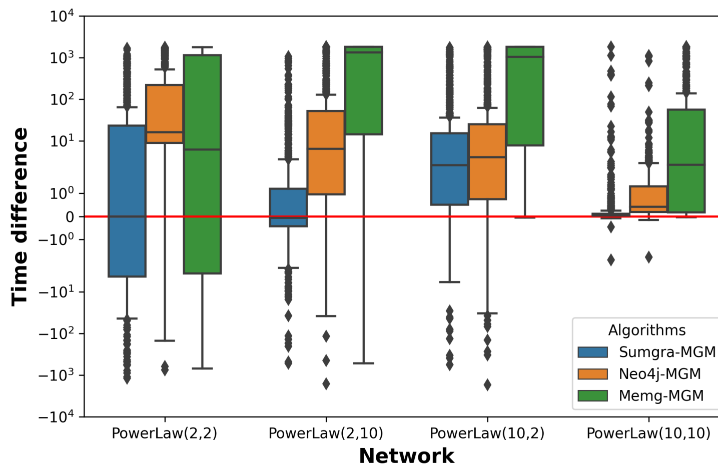


Figure 4.5: Boxplots of the query-by-query running time differences between MultiGraphMatch (MGM) and the other compared tools Sumgra, Neo4J and Memgraph (Mem) for the dataset of queries run on power-law synthetic networks. MultiGraphMatch is as fast as or significantly faster than the other methods.

Network	Comparison	Median time difference	P-value	Confidence interval
UNIFORM(2,2)	(MGM, Neo4J)	13.71	< 0.0001	$[12.02, 18.93]$
	(MGM, Sumgra)	2.87	< 0.0001	$[2.46, 4.36]$
	(MGM, Mem)	1176.85	< 0.0001	$[1041.50, 1305.88]$
UNIFORM(2,10)	(MGM, Neo4J)	5.62	< 0.0001	$[4.60, 6.67]$
	(MGM, Sumgra)	-0.33	0.3351	$[-0.36, -0.31]$
	(MGM, Mem)	736.76	< 0.0001	$[566.26, 989.78]$
UNIFORM(10,2)	(MGM, Neo4J)	2.13	< 0.0001	$[1.66, 2.72]$
	(MGM, Sumgra)	0.58	< 0.0001	$[0.54, 0.65]$
	(MGM, Mem)	514.54	< 0.0001	$[460.38, 1063.16]$
UNIFORM(10,10)	(MGM, Neo4J)	0.31	< 0.0001	$[0.27, 0.35]$
	(MGM, Sumgra)	0.01	< 0.0001	$[0.01, 0.02]$
	(MGM, Mem)	0.97	< 0.0001	$[0.68, 1.19]$
POWERLAW(2,2)	(MGM, Neo4J)	67.29	< 0.0001	$[39.02, 96.35]$
	(MGM, Sumgra)	6.34	< 0.0001	$[4.89, 11.05]$
	(MGM, Mem)	669.46	< 0.0001	$[399.46, 928.76]$
POWERLAW(2,10)	(MGM, Neo4J)	6.42	< 0.0001	$[4.74, 8.96]$
	(MGM, Sumgra)	-0.06	< 0.0001	$[-0.12, -0.01]$
	(MGM, Mem)	1336.38	< 0.0001	$[889.53, 1752.19]$
POWERLAW(10,2)	(MGM, Neo4J)	4.08	< 0.0001	$[3.13, 5.20]$
	(MGM, Sumgra)	2.63	< 0.0001	$[2.14, 3.23]$
	(MGM, Mem)	1056.51	< 0.0001	$[638.95, 1351.53]$
POWERLAW(10,10)	(MGM, Neo4J)	0.41	< 0.0001	$[0.36, 0.44]$
	(MGM, Sumgra)	0.04	< 0.0001	$[0.03, 0.04]$
	(MGM, Mem)	2.66	< 0.0001	$[1.85, 3.81]$

Table 4.2: Median, p-value of significance and 90% confidence interval of the query-by-query running time difference (in seconds) between MultiGraphMatch (MGM) and the other tools Sumgra, Neo4J and Memgraph (Mem) on a dataset of queries run on synthetic networks. Uncompleted queries before the timeout were assigned a running time of 1,800 seconds.

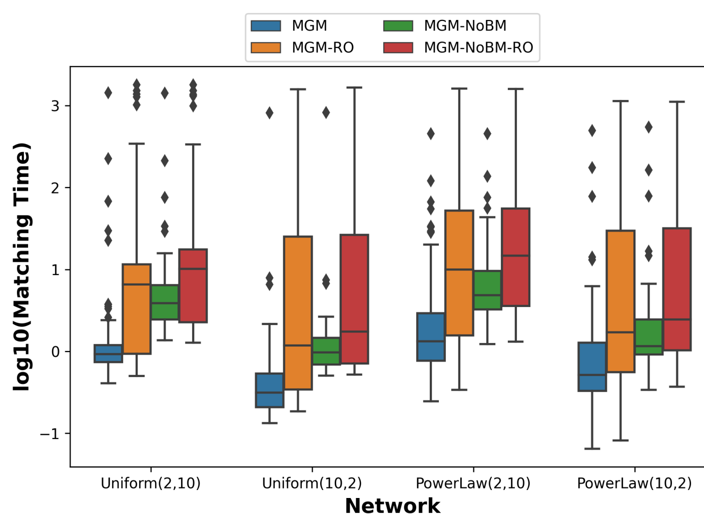


Figure 4.6: Boxplots of running times of MultiGraphMatch (MGM), MultiGraphMatch with no Bit Matrix (MGM-NoBM), MultiGraphMatch with Random Ordering (MGM-RO) and MultiGraphMatch with no Bit Matrix and Random Ordering (MGM-NoBM-RO) on the following synthetic networks: Uniform(2,10), Uniform(10,2), PowerLaw(2,10) and PowerLaw(10,2). Ordering and the bit matrix significantly improve performance, both individually and collectively. All values are relative to a 0 point which is the mean of the matching time of MGM on Uniform(2,10). Lower is better.

Results show that the full algorithm is faster than its simplified versions, implying that both features are important for the performance of MultiGraphMatch. In particular, the bit matrix has a strong positive impact on candidate detection and computation of compatibility domains thanks to the bit signature representation of the types of all edges connecting two nodes and the labels of the source and destination nodes of such edges. This representation is more memory-efficient and less time consuming (high performance of bitwise operations) than a solution which uses a vector of vectors or a map of maps to identify compatible edges. Ordering query edges greatly speeds up matching. An incorrect ordering of query edges may increase the number of backtracking operations and, consequently, the matching time.

4.3.2 Experiments on real networks

Data description

The performance of the various matching algorithms were also evaluated on three real networks of different sizes: IMDB, PANAMA and VECTOR. Tab. 4.3 on page 143 summarizes their main features.

IMDB is a network extracted from the Internet Movie Database (IMDB)[6] that provides information about millions of films, television, home videos, video games, and streaming content online programs including cast and crew. The nodes of IMDB are people working in the show business (actors, producers, composers, writers, etc.) and movies, while edges denote different types of relationships between actors and movies (e.g. an actor who played in a movie or a director who directed a movie). Person nodes are annotated with labels denoting the main professions of a person. Movie nodes have labels denoting the genre of a movie. Properties associated to people are the name of a person, the year of birth and the year of death (if not alive), while properties linked to a movie are its original title and the year of production.

[6]https://www.imdb.com/

The IMDB network was built starting from information stored in flat text files publicly available at `https://imdb-api.com/`.

PANAMA is a multigraph extracted from the Panama Papers, a set of 11.5 million leaked documents containing detailed financial information of more than 200,000 offshore entities and their clients from its foundation in 1977 to April 2015. The documents originally belonged to the Panamanian law firm and corporate service provider Mossack Fonseca. They were leaked in 2015 by an anonymous source. The documents reveal that some of the Mossack Fonseca shell corporations were used for illegal purposes, including fraud, tax evasion, and evading international sanctions. Intermediaries, e.g. law firms, banks or trust companies with external advisors who connect the client to offshore service providers (e.g. Mossack Fonseca), support these activities. Recently, the International Consortium of Investigative Journalists (ICIJ)[7] collected all this information into a freely accessible database called Panama[8]. From this database, we extracted the directed relationship network between offshore entities, intermediaries and officers (i.e. people or companies playing a role in an offshore entity). Nodes are labeled according to their type ('intermediary', 'entity', 'officer' or 'address' of offshore companies). Edges denote the kind of relationship (e.g. 'intermediary of' links an intermediary to an entity, while 'officer of' links an officer to an entity). Experiments consist in running queries consisting of subgraphs randomly extracted from IMDB and PANAMA. The number of nodes of each query varies between 3 and 5 and the density values vary between 0.25 and 1.

VECTOR [11] is a Uveal Melanoma (a type of eye cancer) knowledge network containing:

- the expression of mRNA, transcription factors (TFs) and non-coding RNAs (e.g. miRNAs, lncRNAs) in several human normal tissues and

[7]https://www.icij.org
[8]https://offshoreleaks.icij.org

cancer samples;

- the correlation between the expression profiles of mRNAs, TFs and non-coding RNAs in normal tissues and cancer samples.

  Expression data are taken from The Cancer Genome Atlas (TCGA)[9], a cancer genomics program that collects molecular data concerning about 20,000 primary cancer and matched normal samples spanning 33 different cancer types. Correlations based on Pearson calculation were computed starting from the expression matrices: (a) miRNAs-mRNAs, (b) miRNAs-lncRNAs, and (c) lncRNAs-mRNAs, extracted from the UM TCGA dataset.

Performance comparison

Performance on real networks were evaluated considering two sets of queries: queries having only specified values on node labels and edge types (i.e. without WHERE clauses) and queries also having conditions on node and edge properties (i.e. with WHERE clauses). Queries with WHERE clauses were performed on all real networks, while queries without WHERE clause were tested only on IMDB and PANAMA, because VECTOR is quite dense, therefore queries were not very selective, resulting in many timeouts. In addition, SumGra cannot handle multigraphs with properties, for this reason SumGra was run only on queries without WHERE clause.

We randomly extracted from each real network 300 queries with WHERE clause and 400 queries without WHERE clause having different number of nodes (from 3 to 5) and density values (from 0.25 to 1). WHERE clauses were built by randomly imposing equality constraints (from 1 to 3) on node and edge properties.

[9]https://www.cancer.gov/about-nci/organization/ccg/research/structural-genomics/tcga

Experimental results obtained on queries without and with WHERE clauses are summarized in Tables 4.4 on the following page and 4.5 on page 144, respectively. The two tables report, for each network and for each algorithm, the number of queries completed before the timeout, the mean running time and the relative 90% confidence interval of the mean time. Reported results refer only to queries completed by at least one algorithm before the timeout, with a running time of 1,800 seconds assigned for each uncompleted query.

In Figures 4.7 on page 145 and 4.8 on page 147 we depict boxplots of the query-by-query running time differences between MultiGraphMatch and the other compared tools for the queries without and with WHERE clause, respectively. Boxplots were built by considering only queries completed by at least one algorithm before the timeout and assigning a running time of 1,800 seconds for each uncompleted query. In Tables 4.6 on page 144 and 4.7 on page 145 we list the median, the 90% confidence interval and the p-value of significance of the running time differences between MultiGraph-Match and the other algorithms for queries without and with WHERE clause, respectively. We computed the p-values and confidence intervals using non-parametric methods.[78].

Results show that MultiGraphMatch significantly outperforms the other tools for queries in the absence of WHERE clauses (p-value < 0.05). For queries with WHERE clauses, MultiGraphMatch has a smaller advantage, but it is still the fastest tool by a significant margin. Moreover, MultiGraph-Match completed the most queries before the timeout in all scenarios.

4.3.3 Scalability tests

Data description

To evaluate the scalability of MultiGraphMatch, we used four large networks of different sizes built using the LDBC social network data generator [4]

Network	$\lvert V \rvert$	$\lvert E \rvert$	$\lvert \mathcal{L} \rvert$	$\lvert \mathcal{T} \rvert$
IMDB	9,930,907	37,874,660	67	10
PANAMA	1,908,466	3,142,523	5	13
VECTOR	19,568	16,027,761	6	2

Table 4.3: Main features of the three real networks used in the experiments. For each network we report the number of nodes $|V|$, the number of edges $|E|$, the number of distinct node labels $|\mathcal{L}|$ and the number of distinct edge types $|\mathcal{T}|$.

Network	**Algorithm**	**Completed queries**	**Mean time**	**Confidence interval**
IMDB	MGM	358	**256.50**	$[163.09, 391.47]$
	Neo4J	304	386.12	$[317.88, 454.47]$
	Sumgra	315	344.86	$[283.78, 412.53]$
	Mem	315	286.21	$[225.45, 349.13]$
PANAMA	MGM	162	**31.01**	$[19.97, 46.76]$
	Neo4J	149	231.06	$[157.71, 320.34]$
	Sumgra	161	115.75	$[69.13, 174.04]$
	Mem	128	481.38	$[374.82, 596.47]$

Table 4.4: Performance of MultiGraphMatch (MGM), Sumgra, Neo4J and Memgraph (Mem) on a dataset of queries without WHERE clause extracted from the IMDB and PANAMA networks. The table reports, for each synthetic network and for each algorithm, the number of queries completed before the timeout (30 minutes), the mean running time and the relative 90% confidence interval. Uncompleted queries are considered to take 30 minutes. MultiGraphMatch is significantly better on the two networks compared to Neo4j, Sumgra, and Memgraph.

Network	Algorithm	Completed queries	Mean time	Confidence interval
IMDB	MGM	158	**3.90**	$[1.52, 8.23]$
	Neo4J	156	32.87	$[4.29, 100.50]$
	Mem	158	25.95	$[2.36, 91.91]$
VECTOR	MGM	300	**0.05**	$[0.03, 0.07]$
	Neo4J	300	0.25	$[0.18, 0.42]$
	Mem	300	83.52	$[72.65, 94.87]$
PANAMA	MGM	300	**0.67**	$[0.46, 1.04]$
	Neo4J	300	16.52	$[5.61, 31.42]$
	Mem	250	384.03	$[310.22, 463.60]$

Table 4.5: Performance of MultiGraphMatch (MGM), Neo4J and Memgraph (Mem) on a dataset of queries extracted from the IMDB, VECTOR and PANAMA networks having WHERE clauses. The table reports, for each synthetic network and for each algorithm, the number of queries completed before the timeout (30 minutes), the mean running time and the relative 90% confidence interval. Uncompleted queries are considered to take 30 minutes. MultiGraphMatch is by far the best method.

Network	Comparison	Median time difference	P-value	Confidence interval
IMDB	(MGM, Neo4J)	5.56	0.0060	$[2.77, 11.26]$
	(MGM, Sumgra)	4.75	0.0479	$[3.19, 6.84]$
	(MGM, Mem)	0.80	0.2914	$[0.53, 2.36]$
PANAMA	(MGM, Neo4J)	0.18	< 0.0001	$[-0.45, 1.30]$
	(MGM, Sumgra)	-0.82	< 0.0001	$[-1.24, -0.66]$
	(MGM, Mem)	6.85	< 0.0001	$[0.95, 23.58]$

Table 4.6: Median, p-value of significance and 90% confidence interval of the query-by-query running time difference (in seconds) between MultiGraphMatch (MGM) and the other tools Sumgra, Neo4J and Memgraph (Mem) on a dataset of queries without WHERE clause extracted from IMDB and PANAMA networks. Uncompleted queries before the timeout were assigned a running time of 1,800 seconds.

Network	Comparison	Median time difference	P-value	Confidence interval
IMDB	(MGM, Neo4J)	0.02	0.0293	$[0.01, 0.05]$
	(MGM, Mem)	-0.06	0.0875	$[-0.08, -0.04]$
VECTOR	(MGM, Neo4J)	0.06	< 0.0001	$[0.06, 0.08]$
	(MGM, Mem)	1.74	< 0.0001	$[0.01, 75.24]$
PANAMA	(MGM, Neo4J)	0.10	< 0.0001	$[0.08, 0.11]$
	(MGM, Mem)	6.85	< 0.0001	$[1.94, 17.54]$

Table 4.7: Median, p-value of significance and 90% confidence interval of the query-by-query running time difference (in seconds) between MultiGraphMatch (MGM) and the other tools Sumgra, Neo4J and Memgraph (Mem) on a dataset of queries with WHERE clause extracted from IMDB, VECTOR and PANAMA networks. Uncompleted queries before the timeout were assigned a running time of 1,800 seconds.

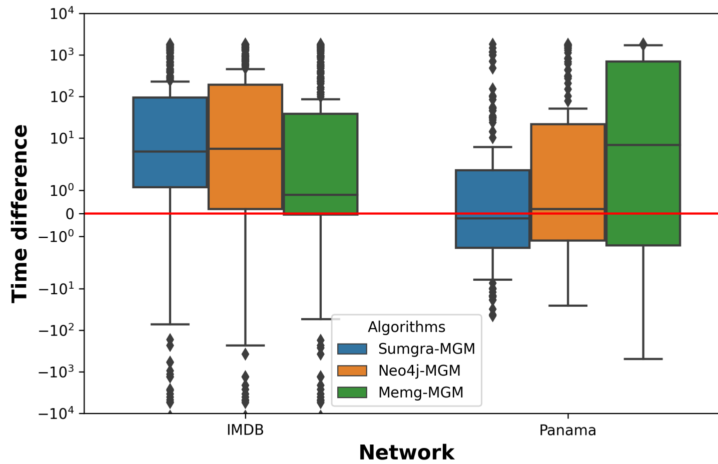


Figure 4.7: Boxplots of the query-by-query running time differences between MultiGraphMatch (MGM) and the state-of-the-art algorithms Sumgra, Neo4J and Memgraph (Mem) for a dataset of queries without WHERE clause extracted from the IMDB and PANAMA networks.

and six queries defined upon these networks, extracted from the Labelled Subgraph Query Benchmark (LSQB)[10] [106].

The LDBC data generator produces artificial social networks modeling the activity of a social network at different periods of time. Networks are formed by people who live in cities and countries and interact each other by establishing friendship relations and posting or replying to messages and comments in forums. People can also form groups to talk about specific topics, represented as tags. The LDBC generator creates networks that mimic the node degree distributions and attribute correlations observed in real social networks. For instance, people are more likely to travel to neighbouring countries and post messages there. The size of generated networks can be controlled through a parameter called scale factor, related to the size of produced output files. For our experiments we used four networks with scale factors 0.1, 0.3, 1 and 3, respectively, downloaded from `https://github.com/ldbc/data-sets-surf-repository`. The main features of these networks are summarized in Table 4.8.

Queries taken from LSQB represent patterns of different sizes and complexity typically observed in the LDBC generated networks. Queries are specifically designed to evaluate the join performance of database management systems. Figure 4.9 on page 150 illustrates the six LSQB queries used for our scalability tests, named as Q1, Q2, Q3, Q4, Q5 and Q6.

Experimental results

In Figure 4.10 on page 151 we depict the running times of MultiGraphMatch for each query of Figure 4.9 on page 150 and for each social network listed in Table 4.8.

We further investigated the relationship between the scale factor of the target s and the running time T and found that:

[10] https://github.com/ldbc/lsqb

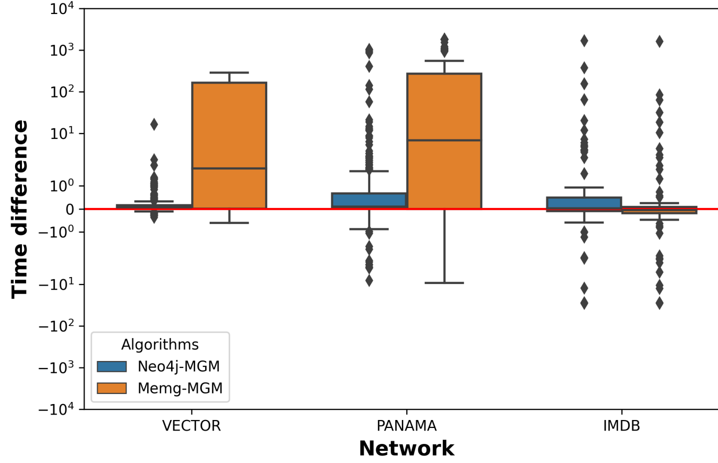


Figure 4.8: Boxplots of the query-by-query running time differences between MultiGraphMatch (MGM) and the state-of-the-art tools Neo4J and Memgraph (Mem) for the dataset of queries having WHERE clauses extracted from the IMDB, PANAMA and VECTOR networks.

Network	$\|V\|$	$\|E\|$	$\|\mathcal{L}\|$	$\|\mathcal{T}\|$
LDBC_0.1	432,235	1,806,538	12	15
LDBC_0.3	1,179,535	5,363,741	12	15
LDBC_1	3,955,790	18,928,261	12	15
LDBC_3	11,257,915	55,790,601	12	15

Table 4.8: Main features of the 4 LDBC networks used in scalability tests. For each network we report the number of nodes, the number of edges, the number of distinct node labels and the number of distinct edge types.

Network	Network reading	Node indexing	Edge indexing	Bit matrix computation
LDBC_0.1	3.852	1.014	5.438	4.394
LDBC_0.3	8.428	2.415	19.856	16.297
LDBC_1	29.413	6.287	63.687	52.015
LDBC_3	81.510	13.815	195.217	167.249

Table 4.9: Times (in seconds) required to read the 4 LDBC networks used for scalability tests, index their node and edges and build their bit matrices.

Network	Query	Ordering	Breaking conditions	Compatibility domains	Matching	Total time
LDBC_0.1	Q1	0.006	0.003	0.765	0.317	1.091
	Q2	0.008	0.004	0.699	0.548	1.259
	Q3	0.008	0.003	0.642	2.287	2.940
	Q4	0.008	0.003	0.101	2.696	2.808
	Q5	0.007	0.004	0.056	5.510	5.577
	Q6	0.007	0.003	0.723	11.618	12.351
LDBC_0.3	Q1	0.007	0.002	1.329	1.522	2.860
	Q2	0.006	0.003	0.156	3.824	3.989
	Q3	0.007	0.004	0.092	11.176	11.279
	Q4	0.007	0.003	1.911	10.483	12.404
	Q5	0.007	0.003	1.848	26.352	28.211
	Q6	0.007	0.002	2.278	53.567	55.854
LDBC_1	Q1	0.007	0.002	5.979	6.874	12.861
	Q2	0.007	0.002	6.478	10.627	17.114
	Q3	0.006	0.003	5.112	55.436	60.557
	Q4	0.007	0.004	1.905	70.560	72.476
	Q5	0.007	0.004	0.451	186.318	186.780
	Q6	0.007	0.004	7.311	294.302	301.624
LDBC_3	Q1	0.007	0.002	20.775	18.418	39.202
	Q2	0.009	0.003	24.743	19.919	44.674
	Q3	0.006	0.003	23.150	192.921	216.080
	Q4	0.007	0.004	3.550	267.271	270.832
	Q5	0.007	0.004	0.894	1219.036	1219.941
	Q6	0.007	0.004	32.141	1292.274	1324.426

Table 4.10: Times (in seconds) required to compute processing order of query edges, symmetry breaking conditions, compatibility domains and matching for each query depicted in Figure 4.9 on page 150 in each target network listed in Table 4.3 on page 143.

- For queries Q1 and Q2, $T \sim 15s^1$;
- For queries Q3 and Q4, $T \sim 60s^{1.25}$;
- For queries Q5 and Q6, $T \sim 240s^{1.5}$.

We can conclude that, the time rises super-linearly according to a power law relationship of the form $T = as^k$ between s and T, with larger k values as the query complexity increases, but the k value is always modest.

In Table 4.9 we report the time required to index target data structures, whereas Table 4.10 contains, for each experiment, the time required to compute symmetry breaking conditions, processing order of query edges, compatibility domains and matching. The indexing time increases linearly with the size of target graph, while the time to build domains rises linearly with the query size. Computation of symmetry breaking conditions and ordering of query edges always require negligible time.

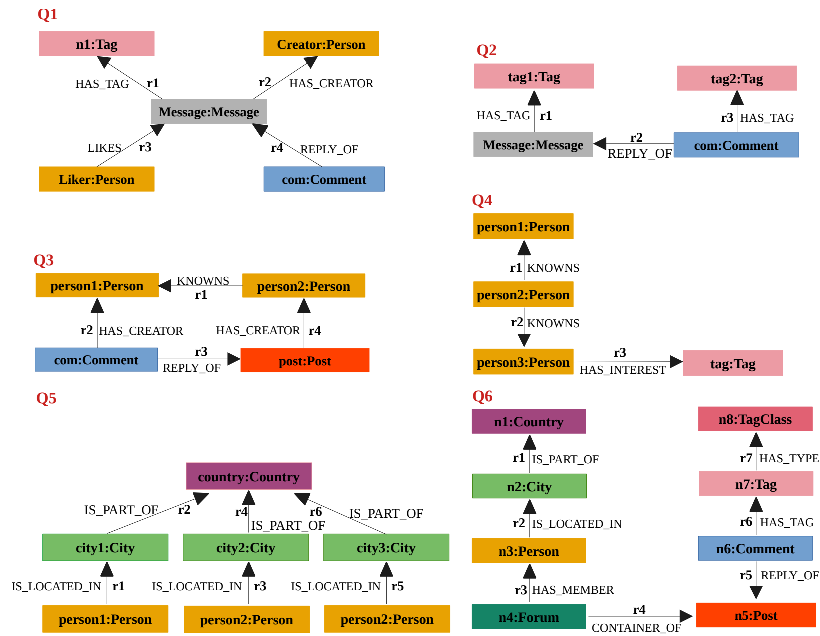


Figure 4.9: Queries of the LSQB benchmark used for scalability experiments.

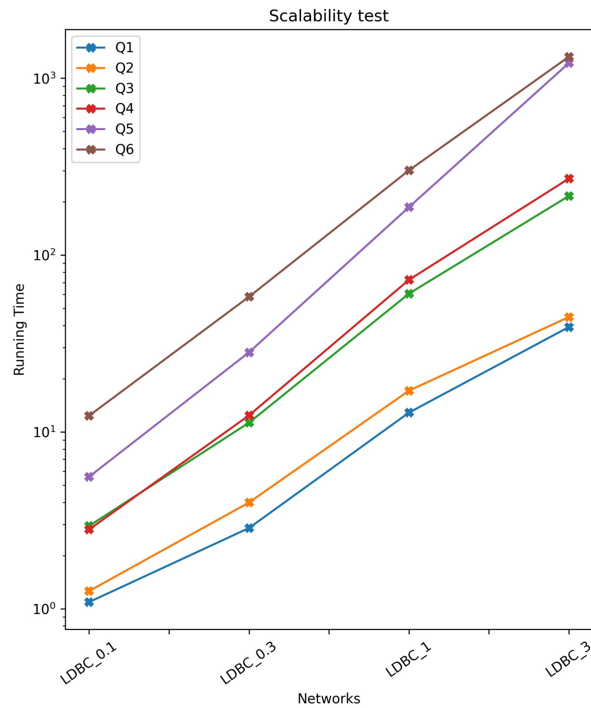


Figure 4.10: Running times of MultiGraphMatch for the networks listed in Table 4.3 on page 143 and the queries depicted in Figure 4.9. For the simplest queries Q1 and Q2, the time scales linearly with network size. For the other queries, there is a power law relationship but with an exponent of 1.5 or less. Please see text for details.

4.4 Conclusion

We have proposed both algorithm and a system to match subgraphs to graphs where both the subgraphs and graphs may have multiple labels on nodes and multiple labeled edges between pairs of nodes [107]. The main algorithmic novelties are:

- We use bit matrices on edges to accelerate the construction of compatibility domains, which are sets of target edges that can be matched with query edges.
- We order the query edges according to a new scoring scheme that exploits the cardinalities of the compatibility domains, which reduces the search space of possible solutions.
- We adapt the breaking conditions to the multigraph setting.

In Section 4.3.1, we experimentally demonstrate the benefits of these contributions.

Experiments performed on both synthetic and real networks show that we are generally faster by a factor ranging from 2 to 10 than state-of-the-art systems.

Our improvement is statistically significant (p-value < 0.05) in almost all our experiments. Scalability tests show that MultiGraphMatch scales subquadratically with the size of the target, whereas the indexing time and the compatibility domain time rise linearly with the sizes of the target and the query, respectively.

Future work includes:

- Provide our software to the community version of Neo4j;
- Maintain our data structures as the target graph changes;
- Further optimizations to filtering.

5 MODIT
MOtif DIscovery in Temporal networks

In this chapter, we present a new motif search algorithm, called MODIT (MOtif DIscovery in Temporal networks). Our algorithm overcomes many of the limitations imposed by other motif search methods. In fact, MODIT is general and can search for motifs of any size. It has no consecutive edge restriction and allows edges with equal timestamps, provided that they do not link the same pair of nodes [122].

The rest of the chapter is organized as follows. In Section 5.1 we illustrate MODIT and evaluate its computational complexity. In Section 5.2 we assess the performance of MODIT on a dataset of real networks and compare it with the algorithm presented in [115]. Finally, Section 5.3 ends the chapter.

5.1 Methods

5.1.1 The MODIT algorithm

In what follows we introduce a new algorithm for solving the TMS problem, called MODIT (MOtif DIscovery in Temporal networks) [122].

Given three parameters k, l and Δ, MODIT scans a temporal graph T to retrieve all temporal motifs with at most k nodes and l edges which Δ-occur

in T and counts the number of Δ-occurrences of each motif in T.

For each newly identified occurrence, the algorithm performs the following steps:

1. Standardization of edge timestamps;

2. Construction of the canonical form and identification of the corresponding temporal motif;

3. Update of the count of the number of motif occurrences.

The search starts from the smallest motif occurrences formed by single edges. We call these edges seeds. Each of these occurrences is then recursively extended by adding one edge at the time until the specified maximum number of nodes and edges is reached.

MODIT can work on both undirected and directed graphs. For ease of presentation, we illustrate the functioning of the algorithm for undirected networks. However, all the procedures presented here can be easily adapted to directed networks.

The pseudocode of MODIT is reported in Algorithm 11 on page 157. All motifs retrieved by the algorithm, together with the number of their occurrences, are stored in a hash map $motifMap$, empty at the beginning (line 1). Each motif in the map is uniquely represented by a string, called canonical form. In addition, since the same occurrence of a motif may be examined multiple times, we also need to store all the distinct retrieved subgraph occurrences in a hash set *minedSubgraphs* (line 2). For each edge $e = (u, v, t)$ of the target graph, MODIT performs the following steps. First, e and its nodes are embedded in a new occurrence graph S, which is added to the set *minedSubgraphs* (lines 4-7). The timestamp of the seed edge e is stored in a variable $minTime$, which will be used throughout the search to avoid scanning some subgraphs of T multiple times (line 8). Each edge that will be added to the subgraph must have a timestamp greater than or equal to

minTime. To ensure that each subgraph obtained by expanding S does not violate the Δ temporal constraint, MODIT also uses three auxiliary variables: *timestampSet*, *maxTime* and λ (lines 9-12). Variable *timestampSet* is a multi-set containing the timestamps of the currently examined subgraph S (at the beginning just t). *maxTime* will store at each step the maximum timestamp of edges in S. λ represents how much we can extend the time window covered by edges in S (i.e. the difference between the maximum and the minimum timestamps), without exceeding Δ. *maxTime* and λ are initialized and kept updated during the search using Algorithm 12 on page 157 (line 13).

Next, the timestamps of the current subgraph S are standardized, i.e they are modified in order to transform S into a temporal motif M (line 14). Standardization of timestamps aims at identifying the motif M of which S is an occurrence and works as follows. First, the list of edge timestamps in S (without duplicates) is sorted in ascending order. Then, each edge of S is assigned the rank of the corresponding timestamp in the sorted list.

Standardization alone may produce distinct motifs that are actually structurally equivalent and in which equivalent edges have the same timestamps. To avoid this, a canonical form C is extracted from M (line 15). The canonical form is a string that uniquely represents a motif, so that two motifs that are equivalent have exactly the same canonical forms. C is obtained by concatenating motif edges based on a given order of motif nodes. Nodes are first ordered according to their degree. Possible ties are solved comparing their sorted adjacency lists, in which edges are ordered by timestamp and, in case of ties, by destination node. Following the calculated node ordering, sorted adjacency lists of nodes are concatenated to yield the canonical form. Figure 5.1 on page 157 shows an example of computation of canonical form. In the depicted motif, node a is the first node in the ordering, since it has the maximum degree (2). Nodes b and c have both degree 1. If the adjacency lists of b

and c are sorted by timestamps, b comes before c because the first edge in the sorted adjacency list of b has timestamp 1, while the first edge in the sorted adjacency list of c has timestamp 2. Following the node ordering $\{a, b, c\}$, sorted adjacency lists of a, b and c are appended to yield the final canonical form of the motif, i.e. the string $C = \{(a, 1, b), (a, 2, c), (b, 1, a), (c, 2, a)\}$.

After constructing the canonical form, the number of occurrences of M is incremented in $motifMap$ (line 17).

Next, MODIT continues the search of new occurrences by extending S edge by edge, starting from an anchor node. This is done using the recursive procedure RecursiveSearch described in Algorithm 13 on page 159 (lines 17-18). The first two calls to the procedure will extend the seed using both endpoints as anchors. In general, S will be extended by adding an edge which is not already present in S and is incident to an anchor node already present in S.

The structure of RecursiveSearch procedure is very similar to Algorithm 11. First, we check if the currently examined subgraph has reached the maximum allowed number of edges (lines 1-2). If so, the recursive algorithm stops, otherwise the search goes on, considering all possible edges $e = (u, a, t)$ not already present in S and incident to the anchor node a (line 3). For each such edge, MODIT ensures that by adding e to S, the Δ temporal constrain is not violated (line 4). Based on the current values of $minTime$, $maxTime$ and λ, we can add e to S without violating the Δ constraint iff $minTime \leq t \leq maxTime + \lambda$. We impose that t is no lower than $minTime$, i.e. the timestamp of the seed edge, to reduce the number of redundant candidates generated.

If e does not violate the Δ constraint, before adding it to S (line 11), we do the following steps. First, we check if e does not connect two nodes already present in S (line 5). If so, we verify if the currently examined subgraph has not reached the maximum allowed number of nodes (line 6). In this case,

Algorithm 11 MODIT(G, Δ, k, l)

1: Let motifMap be an empty hash map.
2: $minedSubgraphs = \emptyset$
3: **for each** $e = (u, v, t) \in E(G)$ **do**
4: Let S be an empty graph.
5: $V(S) = V(S) \cup \{v\}$
6: $E(S) = E(S) \cup \{e\}$
7: $minedSubgraphs = minedSubgraphs \cup S$
8: $minTime = t$
9: $maxTime = 0$
10: $\lambda = +\infty$
11: Let timestampSet be an empty multiset
12: $timestampSet = timestampSet \cup \{t\}$
13: UpdateBounds($timestampSet, \Delta, minTime, maxTime, \lambda$)
14: M = StandardizeTimestamps(S)
15: C = ComputeCanonization(M)
16: UpdateOccurrences($motifMap, C$)
17: RecursiveSearch($S, u, \Delta, timestampSet, minTime, maxTime, \lambda, motifMap, k, l, minedSubgraphs$)
18: RecursiveSearch($S, v, \Delta, timestampSet, minTime, maxTime, \lambda, motifMap, k, l, minedSubgraphs$)
19: **end for**
20: **return** motifMap

Algorithm 12 UpdateBounds($timestampSet$, Δ, $minTime$, $maxTime$, λ)

1: $maxTime$ = Max($timestampSet$)
2: $\lambda = \Delta - maxTime + minTime$
3: **return**

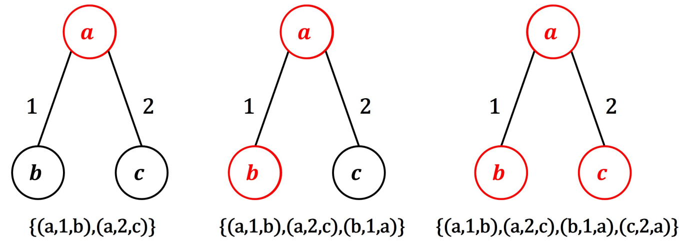


Figure 5.1: Example of construction of the canonical form. Node a has the highest degree (2), so a is the first node of the ordering. Nodes b and c have the same degree, so we need to examine the adjacency lists of b and c sorted by timestamp and destination node. The first edge in the sorted list of b has timestamp 1, while the first edge in the sorted list of c has timestamp 2, so the second node of the ordering is b and the third one is c. Following such node ordering, the canonical form of the graph in the figure is $(a, 1, b), (a, 2, c), (b, 1, a), (c, 2, a)$.

node u is added to S (line 7). Otherwise, we proceed with the next edge (line 10).

Boolean conditions expressed in line 4 does not prevent examining some subgraphs multiple times. Therefore, before going on with the search, we need to check if S has not been already examined before (line 12). This is done by simply comparing the list of edge ids of S and each subgraph of the *minedSubgraphs* set.

If S is new, we follow the same steps performed in algorithm 11. First, we include t in *timestampSet* and add S to *minedSubgraph* (lines 13-14). We update temporal auxiliary variables $minTime$, $maxTime$ and λ (line 15). Then, edge timestamps are standardized to obtain a temporal motif M (line 16). From M we extract the canonical form C (line 17) and increase the number of its occurrences (line 18).

Next, Algorithm 13 is called recursively twice using the endpoints of u as anchor nodes (lines 19-20). After returning from the recursive calls, backtracking is performed (lines 21-26). Backtracking implies: i) removing from S the last added edge e, ii) removing from S the last added node, iii) removing the timestamp of e from *timestampSet*, iv)updating auxiliary variables $maxTime$ and λ.

5.1.2 MODIT complexity analysis

In this subsection we analyze the complexity of MODIT. The search starts from the smallest motif occurrences formed by single edges. Therefore, the for-loop in Algorithm 11 is performed $|E(G)|$ times, where G is the target graph. Inside the loop, MODIT tries to expand each occurrence as long as possible. Lines 4-7 of Algorithm 11 require constant time because they are applied to a subgraph formed by only one edge.

Now let's analyze the complexity of Algorithm 13. Let d_{max} the maximum node degree of G. The for-loop in Algorithm 13 is performed, in the

Algorithm 13 RECURSIVESEARCH(S, a, Δ, $timestampSet$, $minTime$, $maxTime$, λ, m, k, l, $minedSubgraphs$)

```
 1: if |E(S)| ≥ l then
 2:     return
 3: end if
 4: for each  e = (u, a, t) ∈ Inc(v) ∧ e ∉ E(S) do
 5:     if t ∈ [minTime, maxTime + λ] then
 6:         if u ∉ V(S) then
 7:             if |V(S)| ≤ (k − 1) then
 8:                 V(S) = V(S) ∪ {u}
 9:                 added = true
10:             else
11:                 continue
12:             end if
13:         end if
14:         E(S) = E(S) ∪ {e}
15:         if S ∉ minedSubgraphs then
16:             timestampSet = timestampSet ∪ {t}
17:             minedSubgraphs = minedSubgraphs ∪ S
18:             UpdateBounds(timestampSet, Δ, minTime, maxTime, λ)
19:             M = StandardizeTimestamps(S)
20:             C = ComputeCanonization(M)
21:             UpdateOccurrences(m, C)
22:             RecursiveSearch(S, u, Δ, timestampSet, minTime, maxTime, λ, m, k, l,
    minedSubgraphs)
23:             RecursiveSearch(S, a, Δ, timestampSet, minTime, maxTime, λ, m, k, l,
    minedSubgraphs)
24:             timestampSet = timestampSet \ {e.t}
25:             E(S) = E(S) \ {e}
26:             if added = true then
27:                 V(S) = V(S) \ {u}
28:                 added = false
29:             end if
30:             UpdateBounds(timestampSet, Δ, minTime, maxTime, λ)
31:         end if
32:     end if
33: end for
34: return
```

worst case, d_{max} times. Assuming $\Delta = \infty$, all d_{max} edges are candidates to extend the motif. Lines 4-14 of Algorithm 13 require a constant time. The complexity of Algorithm 12 on page 157 depends on the number of distinct timestamps of the current motif. In the worst case there are l edges and all timestamps are different. Identifying the minimum and maximum requires an ordering of timestamps that has linear complexity. The rest of the operations can be done in constant time. So, the complexity of Algorithm 12 on page 157 is $O(l)$. However, in practice, these operations are done faster because MODIT stores timestamps in a data structure that is self-sorted as elements are inserted/removed. Standardization of timestamps requires linear time with respect to the number of edges. Since a motif can have at most l edges, the complexity is $O(l)$. The time required to build the canonical form depends on the number of nodes and the number of edges of the motif. Sorting the adjacency list of a node requires, in the worst case, l operations. Since a motif can have at most k nodes, sorting their adjacency lists requires at most $k \cdot l$ operations. The ordering of nodes has linear complexity with respect to the number of nodes, thus performs in the worst case, k operations. Therefore, the number of operations required to calculate the canonical form is at most $k \cdot l + k$. Updating the number of occurrences of a motif is done using a hash map, so it takes constant time.

To derive the final complexity of Algorithm 13, we need to evaluate the maximum depth of the recursion. Each call to Algorithm 13 adds one edge at the time, so the maximum recursion depth is l. Assuming no early backtracking, this implies that the complexity of the recursive procedure is $O\Big((l \cdot k \cdot d_{max})^l\Big)$. So overall, the complexity of MODIT is $O\Big(|E(G)| \cdot (l \cdot k \cdot d_{max})^l\Big)$.

5.2 Results

MODIT has been implemented in Java and tested on two datasets of real temporal networks of different sizes, denoted as Dataset 1 and Dataset 2, respectively. Table 5.1 on page 165 lists the main features of the networks of the two datasets. For each graph we report the number of nodes, the number of edges, the number of distinct timestamps and the resolution, i.e. the minimum difference between consecutive timestamps.

Dataset 1 is formed by six networks:

1. SFHH-Conf
2. As-Topology
3. Contacts-Dublin
4. Enron-Email
5. Digg-Friends
6. Yahoo-messages.

SFHH-Conf is a network that describes the interactions between the 405 participants of the SFHH conference in Nice, France [53]. As-Topology is a peer-to-peer communication network between autonomous systems, with data collected between February and March of 2010. Contacts-Dublin is a contact network of attendees at the *Infectious SocioPatterns* event held in the Science Gallery in Dublin, Ireland [75]. Enron-Email is a network of e-mail exchanges of *Enron corporation* employees between 1999 and 2003 [81]. Digg-Friends describes friendly bonds between users of *Digg*, a web news aggregator used in America [68]. It is based on data collected in one month of 2009. Yahoo-messages represents the exchange of e-mails between users of *Yahoo-Mail* in one month of 2010.

Dataset 2 includes 6 temporal networks taken from the SNAP dataset[1].

CollegeMsg consists of private messages sent on an online social network at the University of California, Irvine [114]. The network Email-Eu-core-temporal was generated using email data from a large European research institution. Only emails exchanged between institution members were taken into account. Email-Eu-core-temporal-Dept1, Email-Eu-core-temporal-Dept2, Email-Eu-core-temporal-Dept3 and Email-Eu-core-temporal-Dept4 are four sub-networks including communications between members of four different departments of the institution [115].

We first ran MODIT on each network of Dataset 1 for different combinations of values of Δ, k (maximum number of motif nodes) and l (maximum number of motif edges). Then, we compared MODIT to the algorithm proposed by Paranjape et al. in [115], which is included in the SNAP platform for network analysis and uses the same definition of temporal motif. All other methods were discarded because they use different definitions of temporal motifs. This comparison was done on the networks of Dataset 2.

All experiments were performed on an Intel Core i5-7500 processor with 16GB of RAM, 10 of which were used for the Java Virtual Machine.

5.2.1 Experiments on Dataset 1

In this section we illustrate the results of the experiments on Dataset 1. We report in Tables 5.2 on page 166, 5.3 on page 167 and 5.4 on page 168 the results in terms of i) execution times, ii) number of distinct motifs identified, iii) number of occurrences of the most frequent motif and iv) average number of motif occurrences. The experiments were performed for different combinations of values of Δ, k and l. Δ was set to r, $2r$ and $3r$, where r is the resolution of the temporal network. For k we used values 3, 4 and 5. l was varied as a function of k and set to values $k-1$, $2 \cdot (k-1)$ and $3 \cdot (k-1)$.

[1] https://snap.stanford.edu/data/index.html

In some networks (in particular, in As-Topology and Enron-Email) and for some configurations of the parameters, MODIT went out of memory and was unable to finish the execution. In these cases we did not report any running time. This is due to the large number of distinct motifs present in the networks, which leads to an excessive growth of the map of motif counts, together with a large number of occurrences causing many recursive calls of Algorithm 13 on page 159. In fact, as k and l increase, the number of motif topologies and the number of combinations of standardized timestamps increases exponentially (e.g., SFHH-conf network in the following configurations: ($\Delta = 3r$, $k = 3$, $l = 2$), ($\Delta = 3r$, $k = 3$, $l = 4$) and ($\Delta = 3r$, $k = 3$, $l = 6$)). This is confirmed by the high values of the number of occurrences of the most frequent motif and the average number of motif occurrences found for small values of k and l.

Interestingly, in some networks (e.g. Enron-Email, Digg-Friends) we observe that keeping Δ and k fixed and varying l, the number of distinct motifs, the number of occurrences of the most frequent motif and average number of occurrences remain the same.

5.2.2 Comparison with Paranjape's algorithm

Finally we compared MODIT with the algorithm proposed by [115] on the networks of Dataset 2.

For the comparison, we set $k = 3$ and $l = 3$ because Paranjape's algorithm can handle only motifs with 2 or 3 nodes and 3 edges.

Results are reported in Table 5.5 on page 169 and show that Paranjape's algorithm is much faster than MODIT. This gap is mainly due to the fact that Paranjape's method uses a series of efficient dynamic programming algorithms, which are specifically designed to count specific classes of motifs, i.e. motifs with 3 edges. On the other hand, MODIT is general and designed to find motifs of any size and any type.Furthermore, Paranjape's algorithm

searches only motifs having exactly the specified number of nodes and edges. On the other hand, MODIT looks for all motifs having at most the number of nodes and edges specified by the user.

Dataset	Network	Nodes	Edges	Timestamps	Resolution
Dataset 1	SFHH-Conf	403	70,261	3,509	20
	As-Topology	34,761	155,507	32,824	1
	Contacts-Dublin	10,972	415,912	76,944	20
	Enron-Email	86,978	1,134,990	213,218	1
	Digg-Friends	279,374	1,729,983	1,644,369	1
	Yahoo-Messages	100,001	3,157,315	898,174	1
Dataset 2	CollegeMsg	1,899	59,835	58,911	114,878
	email-Eu-core-temporal-Dept1	309	61,046	35,097	21,799
	email-Eu-core-temporal-Dept2	162	46,772	32,340	507
	email-Eu-core-temporal-Dept3	89	12,216	8,911	2,635
	email-Eu-core-temporal-Dept4	142	48,141	26,496	88
	email-Eu-core-temporal	986	332,334	207,880	2,797

Table 5.1: Datasets of temporal networks used for the experiments. For each network we indicate the number of nodes, the number of edges, the number of distinct timestamps and the *resolution*, i.e. the minimum difference between the timestamps of two consecutive edges.

Configuration	Network	Time (s)	Mem (GB)	n	Max	AVG
$k = 3$ $l = 2$	SFHH-Conf	3.06	<1	8	44,221	30,387
	As-Topology	83.10	5	10	16,068,985	1,885,655
	Contacts-Dublin	8.21	2	9	192,318	112,497
	Enron-Email	202.57	8	10	40,484,380	4,050,215
	Digg-Friends	48.16	2	9	259,097	37,211
	Yahoo-messages	30.86	3	9	230,863	47,886
$k = 3$ $l = 4$	SFHH-Conf	11.56	1	67	44,221	10,554
	As-Topology	2,185.33	9	159	16,068,985	12,6181
	Contacts-Dublin	17.49	2	120	192,318	18,141
	Enron-Email	1,527.17	8	30	40,484,380	1,350,257
	Digg-Friends	54.96	2	32	259,097	10,738
	Yahoo-messages	32.05	3	49	230,863	9,825
$k = 3$ $l = 6$	SFHH-Conf	12.63	1	97	44,221	8,032
	As-Topology	2,197.95	7	614	16,068,985	32,710
	Contacts-Dublin	19.10	2	224	192,318	10,259
	Enron-Email	1,527.50	8	30	40,484,380	1,350,257
	Digg-Friends	55.57	2	32	259,097	10,738
	Yahoo-messages	34.52	3	55	230,863	7,863
$k = 4$ $l = 3$	SFHH-Conf	9.98	1	62	87,189	5,932
	As-Topology	-	OOM	-	-	-
	Contacts-Dublin	21.16	2	74	192,318	41,959
	Enron-Email	-	OOM	-	-	-
	Digg-Friends	52.78	2	54	259,097	8,811
	Yahoo-messages	30.50	4	60	311,440	15,852
$k = 4$ $l = 6$	SFHH-Conf	191.19	4	3,097	87,189	5,932
	As-Topology	-	OOM	-	-	-
	Contacts-Dublin	86.88	4	6,603	192,318	1,666
	Enron-Email	-	OOM	-	-	-
	Digg-Friends	48.79	2	175	259,097	2,869
	Yahoo-messages	31.99	4	320	311,440	2,986
$k = 4$ $l = 9$	SFHH-Conf	378.52	4	10,916	87,189	2,623
	As-Topology	-	OOM	-	-	-
	Contacts-Dublin	118.13	4	38,389	192,318	357
	Enron-Email	-	OOM	-	-	-
	Digg-Friends	52.34	2	175	259,097	2,869
	Yahoo-messages	33.72	5	338	311,440	2,827
$k = 5$ $l = 4$	SFHH-Conf	96.65	4	446	31,228	31,228
	As-Topology	-	OOM	-	-	-
	Contacts-Dublin	52.19	3	575	192,318	13,404
	Enron-Email	-	OOM	-	-	-
	Digg-Friends	46.74	2	174	259,097	3,744
	Yahoo-messages	38.87	5	241	466,671	6,970
$k = 5$ $l = 8$	SFHH-Conf	-	OOM	-	-	-
	As-Topology	-	OOM	-	-	-
	Contacts-Dublin	632.27	7	530,099	192,318	119
	Enron-Email	-	OOM	-	-	-
	Digg-Friends	47.69	2	540	259,097	1,315
	Yahoo-messages	38.87	5	1,229	466,671	1,376
$k = 5$ $l = 12$	SFHH-Conf	-	OOM	-	-	-
	As-Topology	-	OOM	-	-	-
	Contacts-Dublin	-	OOM	-	-	-
	Enron-Email	-	OOM	-	-	-
	Digg-Friends	47.55	3	540	259,097	1,315
	Yahoo-messages	37.67	5	1,243	466,671	1,360

Table 5.2: Experiments on Dataset 1 with $\Delta = 3r$ and different combination of values of k and l. For each experiment we report: i) running time, ii) maximum memory approximately used (GB), iii) number of distinct motifs identified, iv) number of occurrences of the most frequent motif and v) average number of motif occurrences. In the cases where no running time is reported, MODIT went out of memory (OOM) due to the high number of motifs present in the network.

Configuration	Network	Time (s)	Mem (GB)	n	Max	AVG
$k = 3$ $l = 2$	SFHH-Conf	3.91	<1	9	84,265	45,239
	As-Topology	80.61	6	10	16,068,985	1,988,007
	Contacts-Dublin	10.89	2	9	352,016	184,917
	Enron-Email	193.68	8	10	40,484,380	4,050,941
	Digg-Friends	54.84	2	9	540,758	69,030
	Yahoo-messages	33.02	3	10	389,792	59,811
$k = 3$ $l = 4$	SFHH-Conf	23.65	2	155	109,003	16,673
	As-Topology	2,191.53	9	202	16,068,985	106,737
	Contacts-Dublin	43.74	3	193	352,016	33,024
	Enron-Email	1,702.50	8	46	40,484,380	88,0880
	Digg-Friends	49.16	2	32	540,758	19,798
	Yahoo-messages	30.12	3	52	389,792	11,550
$k = 3$ $l = 6$	SFHH-Conf	49.99	3	488	109,003	9,248
	As-Topology	2,203.59	7	842	16,068,985	25,650
	Contacts-Dublin	69.95	3	1,035	352,016	8,690
	Enron-Email	1,719.70	8	56	40,484,380	723,600
	Digg-Friends	59.08	2	32	540,758	19,798
	Yahoo-messages	31.65	4	58	389,792	10,355
$k = 4$ $l = 3$	SFHH-Conf	23.00	2	79	282,531	50,403
	As-Topology	-	OOM	-	-	-
	Contacts-Dublin	42.01	3	82	352,016	88,450
	Enron-Email	-	OOM	-	-	-
	Digg-Friends	62.87	3	54	540,758	18,847
	Yahoo-messages	39.92	5	64	508,964	24,395
$k = 4$ $l = 6$	SFHH-Conf	-	OOM	-	-	-
	As-Topology	-	OOM	-	-	-
	Contacts-Dublin	894.01	7	13,421	352,016	5,487
	Enron-Email	-	OOM	-	-	-
	Digg-Friends	54.39	2	183	540,758	5,920
	Yahoo-messages	36.72	4	399	508,964	3,933
$k = 4$ $l = 9$	SFHH-Conf	-	OOM	-	-	-
	As-Topology	-	OOM	-	-	-
	Contacts-Dublin	-	OOM	-	-	-
	Enron-Email	-	OOM	-	-	-
	Digg-Friends	62.12	3	183	540,758	5,920
	Yahoo-messages	43.18	5	429	508,964	3,658
$k = 5$ $l = 4$	SFHH-Conf	340.22	6	633	1,607,533	90,145
	As-Topology	-	OOM	-	-	-
	Contacts-Dublin	165.53	4	672	352,016	39,229
	Enron-Email	-	OOM	-	-	-
	Digg-Friends	60.97	3	186	540,758	9,015
	Yahoo-messages	49.68	5	266	445,230	11,524
$k = 5$ $l = 8$	SFHH-Conf	-	OOM	-	-	-
	As-Topology	-	OOM	-	-	-
	Contacts-Dublin	-	OOM	-	-	-
	Enron-Email	-	OOM	-	-	-
	Digg-Friends	64.95	3	612	540,758	3,195
	Yahoo-messages	52.23	5	1,958	775,230	1,578
$k = 5$ $l = 12$	SFHH-Conf	-	OOM	-	-	-
	As-Topology	-	OOM	-	-	-
	Contacts-Dublin	-	OOM	-	-	-
	Enron-Email	-	OOM	-	-	-
	Digg-Friends	75.33	4	612	540,758	3,195
	Yahoo-messages	55.36	5	2,016	775,230	1,533

Table 5.3: Experiments on Dataset 1 with $\Delta = 3r$ and different combination of values of k and l. For each experiment we report: i) running time, ii) maximum memory approximately used (GB), iii) number of distinct motifs identified, iv) number of occurrences of the most frequent motif and v) average number of motif occurrences. In the cases where no running time is reported, MODIT went out of memory (OOM) due to the high number of motifs present in the network.

Configuration	Network	Time (s)	bf Mem (GB)	n	Max	AVG
$k = 3$, $l = 2$	SFHH-Conf	3.76	<1	9	124,450	62,716
	As-Topology	84.85	6	10	16,068,985	2,056,209
	Contacts-Dublin	12.94	2	9	497,191	253,808
	Enron-Email	192.64	8	10	40,484,380	4,051,491
	Digg-Friends	47.80	3	9	791,420	97,274
	Yahoo-messages	33.82	4	10	510,303	73,198
$k = 3$, $l = 4$	SFHH-Conf	53.57	1	198	222,458	32,430
	As-Topology	2,277.25	8	209	16,068,985	108,927
	Contacts-Dublin	85.07	3	203	497,191	67,993
	Enron-Email	1,716.09	8	54	40,484,380	750,629
	Digg-Friends	58.37	2	33	791,420	26,977
	Yahoo-messages	32.04	4	55	510,303	13,368
$k = 3$, $l = 6$	SFHH-Conf	213.16	4	1,044	222,458	19,003
	As-Topology	2,282.17	7	1,026	16,068,985	22,251
	Contacts-Dublin	235.862	4	1,652	497,191	17,978
	Enron-Email	1,752.45	8	95	40,484,380	426,725
	Digg-Friends	59.23	3	33	791,420	26,977
	Yahoo-messages	33.52	4	61	510,303	12,053
$k = 4$, $l = 3$	SFHH-Conf	38.82	3	82	578,085	90,761
	As-Topology	-	OOM	-	-	-
	Contacts-Dublin	70.91	4	82	634,058	156,582
	Enron-Email	-	OOM	-	-	-
	Digg-Friends	67.76	3	56	791,420	29,108
	Yahoo-messages	43.18	5	64	890,576	33,600
$k = 4$, $l = 6$	SFHH-Conf	-	OOM	-	-	-
	As-Topology	-	OOM	-	-	-
	Contacts-Dublin	-	OOM	-	-	-
	Enron-Email	-	OOM	-	-	-
	Digg-Friends	56.39	3	201	791,420	8,659
	Yahoo-messages	40.62	5	430	890,576	5.030
$k = 4$, $l = 9$	SFHH-Conf	-	OOM	-	-	-
	As-Topology	-	OOM	-	-	-
	Contacts-Dublin	-	OOM	-	-	-
	Enron-Email	-	OOM	-	-	-
	Digg-Friends	58.64	3	201	791,420	8,659
	Yahoo-messages	50.21	2	460	890,576	4,702
$k = 5$, $l = 4$	SFHH-Conf	1,001.28	7	677	4,526,786	217,632
	As-Topology	-	OOM	-	-	-
	Contacts-Dublin	391.67	6	682	806,886	90,608
	Enron-Email	-	OOM	-	-	-
	Digg-Friends	77.35	4	203	791,420	15,624
	Yahoo-messages	63.14	5	276	1,346,934	14,226
$k = 5$, $l = 8$	SFHH-Conf	-	OOM	-	-	-
	As-Topology	-	OOM	-	-	-
	Contacts-Dublin	-	OOM	-	-	-
	Enron-Email	-	OOM	-	-	-
	Digg-Friends	83.79	4	737	791,420	5,228
	Yahoo-messages	64.45	5	2,258	1,346,934	2,125
$k = 5$, $l = 12$	SFHH-Conf	-	OOM	-	-	-
	As-Topology	-	OOM	-	-	-
	Contacts-Dublin	-	OOM	-	-	-
	Enron-Email	-	OOM	-	-	-
	Digg-Friends	83.96	4	737	791,420	5,228
	Yahoo-messages	71.64	5	2,325	1,346,934	2,064

Table 5.4: Experiments on Dataset 1 with $\Delta = 3r$ and different combination of values of k and l. For each experiment we report: i) running time, ii) maximum memory approximately used (GB), iii) number of distinct motifs identified, iv) number of occurrences of the most frequent motif and v) average number of motif occurrences. In the cases where no running time is reported, MODIT went out of memory(OOM) due to the high number of motifs present in the network.

Network	Δ	Paranjape et al.	MODIT
CollegeMsg	350,000	0.18	1,148.96
email-Eu-core-temporal-Dept1	70,000	0.11	81.01
email-Eu-core-temporal-Dept2	70,000	0.09	56.71
email-Eu-core-temporal-Dept3	70,000	0.04	3.47
email-Eu-core-temporal-Dept4	70,000	0.08	46.45
email-Eu-core-temporal	70,000	0.66	281.32

Table 5.5: Comparison between MODIT and Paranjape's algorithm. For each network, we indicate the value of Δ used and the running time (in seconds) of both algorithms. MODIT was run with $k = 3$ and $l = 3$ because Paranjape's algorithm can only handle motifs with 2 or 3 nodes and 3 edges.

5.3 Discussion

In this chapter, we presented MODIT (MOtif DIscovery in Temporal Networks) [122], an algorithm for counting motifs of any size in temporal networks. Given the three parameters k, l and Δ, MODIT scans the whole temporal graph to search for all subgraphs having at most k nodes and l edges and in which the difference between the maximum and the minimum timestamp is no greater than Δ. We ran MODIT on a dataset of real temporal networks of medium and large size by varying Δ, the maximum number of nodes and edges. We also compared MODIT with the algorithm proposed by [115] using a different dataset of temporal networks downloaded from SNAP.

For the future, we plan to:

1. introduce measures of statistical significance of motifs similar to the ones already devised for static graphs (i.e. the z-score);
2. make MODIT iterative, in order to reduce the overhead introduced by recursion, and optimize the search strategy;
3. implement MODIT on a SPARK framework, to manage very large networks.

6
MPXGAT
An Attention based Deep Learning Model for Multiplex Graphs Embedding

Graph representation learning has rapidly emerged as a pivotal field of study. Despite its growing popularity, the majority of research has been confined to embedding single-layer graphs, which fall short in representing complex systems with multifaceted relationships. To bridge this gap, we introduce MPXGAT, an attention based deep learning model for multiplex graphs embedding. Our methodology, based on GATs[1], consists in embedding the nodes of a multiplex network by leveraging the information about their intra-layer and inter-layer connections, allowing for link prediction tasks both within the same layer and across different layers. We carry out a thorough experimental analysis on three benchmark datasets, showing that MPXGAT out-performs state-of-the-art competing algorithms. We conclude with an in-depth study of the model main features, proving how their use positively impacts the performance of the algorithm itself.

[1]Graph Attention Networks (GATs) are neural network architectures designed for graph-structured data. GATs use attention mechanisms to aggregate and weight information from a node's neighbors, allowing for more nuanced embeddings by focusing on the most relevant connections in a graph [152].

6.1 Methods

In this section, we provide details of our model, *MPXGAT*, which can embed multiplex networks, namely networks where multiple types of link exist. We first provide some preliminary notions, next, we introduce the mathematical formulation of the model, and finally, we describe the algorithmic implementation.

6.1.1 The MPXGAT General Framework

The main idea of MPXGAT is to generate two separate embeddings for each node in two different phases [19]. In the first phase, each node is embedded according to the horizontal layers, where it has multiple types of relation with other nodes. In the second phase, nodes are embedded according to the vertical network, where they are linked to their counterparts on different layers. In the following, we will use the superscript $\cdot^H$ when we refer to the horizontal network, while the superscript $\cdot^V$ will refer to the vertical one. The notation used, together with the most relevant variables of our model, are reported in Table 2.4 on page 51.

Our algorithm uses two sub-models, one for each embedding phase: MPXGAT-H and MPXGAT-V (an illustration of the model structure is shown in Figure 6.1). In the most general framework, MPXGAT-H applies a series of GAT convolutional layers independently to each layer of the horizontal network.

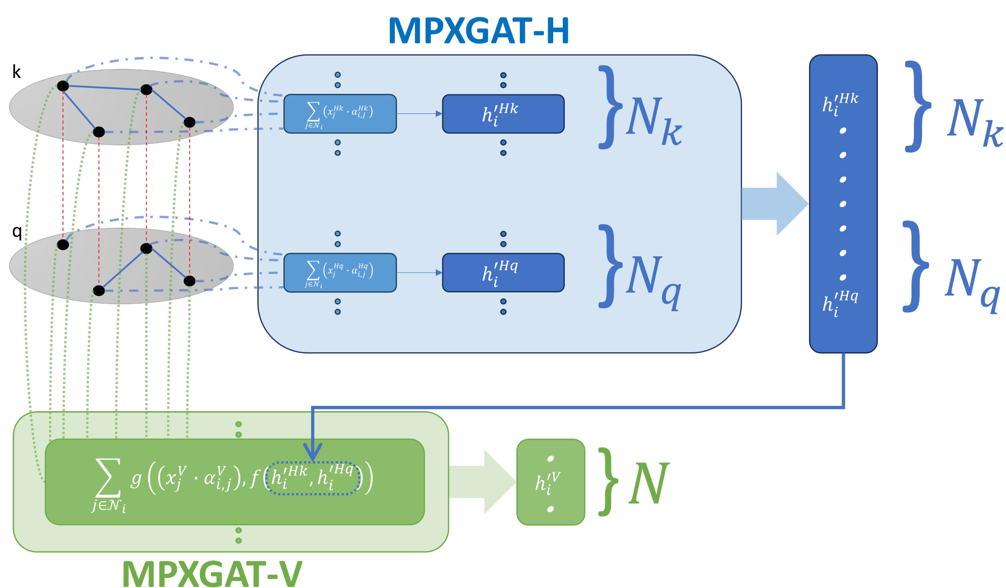


Figure 6.1: The structure of the MPXGAT model. In this toy example, it is applied on a multiplex network with 2 horizontal layers where the solid blue edges represent the intra-layer connections while the dashed red edges are the inter-layer edges. The data is provided to the *MPXGAT-H* throughout the dot-and-dash blue lines. Once processed these are used to feed the *MPXGAT-V* together with the inter-layer links (the dotted green lines). The output of the model consists of both horizontal and vertical nodes embedding.

Mathematically, the embedding is described by the following equations:

$$e_{i,j}^{H_k} = \text{LeakyReLU}\left(\left[\boldsymbol{W}^{H_k} \cdot \boldsymbol{h}_i^{H_k} \cdot (\boldsymbol{v}^{H_k})^T || \boldsymbol{W}^{H_k} \cdot \boldsymbol{h}_j^{H_k} \cdot (\boldsymbol{v}^{H_k})^T\right]\right) \quad (6.1a)$$

$$\alpha_{i,j}^{H_k} = \text{softmax}(e_{i,j}^{H_k}) = \frac{\exp(e_{i,j}^{H_k})}{\sum_{z \in \mathcal{N}_i^{H_k}} \exp(e_{i,z}^{H_k})} \quad (6.1b)$$

$$\boldsymbol{h}_i'^{H_k} = \sigma\left(\sum_{j \in \mathcal{N}_i^{H_k}} \alpha_{i,j}^{H_k} \cdot \boldsymbol{W}^{H_k} \cdot \boldsymbol{h}_j^{H_k}\right) \quad (6.1c)$$

Equations 6.1 govern the convolutional layer as described in the GAT model [152]. In particular, in Equation 6.1c, $\boldsymbol{h}_i'^{H_k}$ represents the horizontal embedding of the node i that belongs to the layer k, while Equation 6.1b describes the self attention mechanism in layer k.

Leveraging the horizontal embeddings, MPXGAT-V generates the vertical embeddings by making use of a custom mechanism inspired by the one applied in the GAT model, which we call GAT-V. The equations to compute the vertical embedding of node i are the following:

$$e_{i,j}^{V^{k,q}} = \text{LeakyReLU}\left(\left[\boldsymbol{W}^{V} \cdot \boldsymbol{h}_i^{V} \cdot (\boldsymbol{v}^{V})^T || \boldsymbol{W}^{V} \cdot \boldsymbol{h}_j^{V} \cdot (\boldsymbol{v}^{V})^T\right]\right) \quad (6.2a)$$

$$\alpha_{i,j}^{V^{k,q}} = \text{softmax}(e_{i,j}^{V^{k,q}}) = \frac{\exp(e_{i,j}^{V^{k,q}})}{\sum_{z \in \mathcal{N}_i^{V}} \exp(e_{i,z}^{V})} \quad (6.2b)$$

$$\boldsymbol{h}_i'^{V} = \sigma\left(\sum_{j \in \mathcal{N}_i^{V}} g\left(\left[\alpha_{i,j}^{V} \cdot \boldsymbol{W}^{V} \cdot \boldsymbol{h}_j^{V}\right], f(\boldsymbol{h}_i^{H_k}, \boldsymbol{h}_j^{H_q})\right)\right) \quad (6.2c)$$

Equations 6.2 contain two novel elements. The first is the function f, which transforms the final horizontal embeddings to the same dimensional space of the vertical embeddings. Such a function can be a simple linear transformation of the node horizontal embeddings, or it can be more complex, depending on the specific task considered. The second feature we introduce is the function g, which combines the data obtained from the current vertical embedding with the results of the the function f.

6.1.2 MPXGAT Implementation for Link Prediction

We now present an implementation of *MPXGAT* that is suitable for link prediction [19]. We customize the attention mechanisms of both *MPXGAT-H* and *MPXGAT-V*, which are described in Equations 6.1a, 6.1b and 6.2a, 6.2b, respectively. In particular:

- in Equations 6.1a and 6.2a, instead of a single weight matrix for both i and j nodes, i.e., $\boldsymbol{W}^H$, two different weight matrices, $\left(\boldsymbol{W}_i^H, \boldsymbol{W}_j^H\right)$, are used. By doing that, we improve the representation capabilities of the model, at the expense of a greater number of parameters. The same choice is done for the weight vector, where instead of a single vector $\boldsymbol{v}^H$, we use two, $\left(\boldsymbol{v}_i^H, \boldsymbol{v}_j^H\right)$. Moreover, to improve the generalization capabilities of the model, the concatenation operator $||$ is implemented as a sum $+$, and both parts of this operation are augmented with the introduction of a bias vector;

- in Equations 6.1b and 6.2b an additional dropout mechanism is applied to the attention coefficients. In this way, we sub-sample the paths present in the graph, granting the model a better generalization capability.

A further addition is the use of *multiple attention heads*, consisting in calculating different embeddings of the same nodes applying multiple times the convolutional layer. This generates multiple node representations that highlight different kinds of patterns about the nodes, as each attention head can be seen as a distinct information channel representing a certain aspect of nodes. The implemented sub-models concatenates the embeddings for each attention head in the hidden convolutional layers of the model, and uses a mean operation for the final one.

Finally, for both sub-models the activation function σ in Equations 6.1c and 6.2c we consider the *LeakyReLU*.

The general *MPXGAT* framework rely on the f and g functions for the embedding, which can be specified according to the particular task we want to solve. Here, the two functions f and g introduced in Equation 6.2c have the following characteristics:

- f performs a linear transformation of the horizontal embedding of the source node, $\boldsymbol{h}_i^H$, to the same dimensional space of its vertical embedding, $\boldsymbol{h}_i^V$, here described with $\boldsymbol{x}_i^H$. This is done through an additional weight matrix $\boldsymbol{Z}^H$. The result is then multiplied by a vector of parameters $\boldsymbol{v}_i^H$, whose role is to enhance the patterns from the horizontal embeddings. Finally, the LeakyReLU function is applied.

- The function g performs a weighted sum of the output of the function f (see Equation 6.3c) and the data computed with the convolutional layer for the vertical network (see Equation 6.3d). A scalar parameter β, automatically inferred during the learning process, is introduced to tune the information coming from the horizontal and vertical embeddings, respectively. In particular when $\beta = 0$ the horizontal embedding is not considered, while for $\beta = 0.5$ both the components are considered with the same weight.

  Formally, the equations describing the implementation of *MPXGAT* for link predictions are:

$$\boldsymbol{x}_i^H = \boldsymbol{h}_i^H \cdot \boldsymbol{Z}^H + \boldsymbol{b}_{h_i} \tag{6.3a}$$

$$\alpha_i^H = \text{LeakyReLU}\left(\boldsymbol{x}_i^H \cdot \boldsymbol{v}_i^H \right) \tag{6.3b}$$

$$\boldsymbol{m}_i^H = f(\boldsymbol{x}_i^H) = \alpha_i^H * \boldsymbol{x}_i^H \tag{6.3c}$$

$$\boldsymbol{m}_{i,j}^V = \alpha_{i,j}^V \cdot \boldsymbol{x}_j^V \tag{6.3d}$$

$$g\left(\boldsymbol{m}^V, \boldsymbol{m}^H\right) = \left(\boldsymbol{m}_{i,j}^V * (1 - \text{ReLU}(\beta))\right) + \left(\left(\boldsymbol{m}_i^H\right) * \text{ReLU}(\beta)\right) \tag{6.3e}$$

A few aspects of our model are worth remarking. First, we assume neither to have the same number of nodes in each horizontal layer nor to have

all possible inter-layer links, making our algorithm a valuable tool for reconstructing inter-layer connectivity. Second, *MPXGAT* is quite flexible, as it supports the use of node features and edge weights. In particular, different features can be used by the two sub-models, since the horizontal and the vertical networks are defined and processed as separated entities. As we will show in the next section, these characteristics guarantee our model good performances even in datasets where the edge density in the horizontal or vertical networks is high, which represents a strong limit of algorithms based on GraphSAGE [49].

6.2 Results

In this section, we present the results obtained by *MPXGAT* in predicting both intra-layer and inter-layer links. We first present the datasets studied, as well as the experimental setup used for the analysis, and the competing methods considered as benchmarks for our algorithm. Then, we test the performances of *MPXGAT* in the link prediction task. We round up the analysis by assessing the impact of the horizontal and vertical sub-models on the embedding procedure.

6.2.1 Dataset

To assess the performance of *MPXGAT* in predicting both intra-layer and inter-layer connections, we employ the same datasets that have been originally used to test *MultiplexSAGE* [49]. These include data relative to three types of real-world multiplex networks, namely a collaboration, a biological, and an online social networks. Specifically, we considered the following datasets.

- **arXiv** [40]. The arXiv multiplex network represents collaborations in various research topics published on the pre-print archive. Each layer

of the network corresponds to a different research category or theme. The network was obtained by selecting papers published before May 2014 that contain the keyword *networks* in their titles or abstracts.

- **Drosophila** [133]. This multiplex network represents the interactions between proteins and genes in the common fruit fly, i.e., Drosophila melanogaster, with each layer corresponding to interactions of various types. These interaction types include physical protein-protein interactions, genetic interactions, and others derived from various experimental methodologies. Some notable interaction types found in the dataset involve protein modifications, synthetic lethality, and co-localization. These diverse interaction categories highlight the broad applicability of this dataset in studying the biological network of Drosophila melanogaster, especially in functional genomics and systems biology contexts. The dataset was collected from the Biological General Repository for Interaction Datasets (BioGRID), with data updated until January 2014.

- **ff-tt-yt** [45]. This multiplex network is derived from Friendfeed (ff), a social media aggregation platform where users can link their accounts from various online social networks (OSNs). The network comprises users who have registered a single Twitter (tw) account and a sole YouTube (yt) account on Friendfeed. Additionally, the Twitter and YouTube accounts are linked to a single Friendfeed account.

For each empirical dataset, we consider exclusively the largest connected component of the multiplex networks and we treat all networks as undirected and unweighted. Details about the largest connected component for each of these datasets are provided in Table 6.1 on page 182.

We remark that, for all datasets considered in our analysis, nodes are not provided with any external features. Hence, we associate to each node, n, a one-hot encoding vector defined by the Kronecker function, $\delta_{i,n}$.

6.2.2 Experimental setup

To assess the performance of *MPXGAT*, we follow the experimental setup delineated in [49]. We partition the data using a multiple-step procedure. First, we randomly select 20% of the network nodes, labeling them as *marked nodes*. We then define the test and training sets, both of which encompass positive and negative examples. Positive examples correspond to actual links within the network, while negative examples consist of pairs of unconnected nodes.

In the test set, positive examples comprise a subset of 20% of the intra-layer links and all inter-layer links among the marked nodes. Conversely, positive examples in the training set include all remaining intra-layer and inter-layer links in the multiplex network. Negative examples within the test set include 20% of all possible negative intra-layer links among the marked nodes, as well as all possible inter-layer links between them. The negative examples in the training set correspond to the remaining pairs of unconnected nodes. This setup follows the *negative sampling* technique to balance the training of the model.

A stratified sampling approach was adopted to ensure that positive and negative examples were proportionally distributed across the training and test sets. This strategy prevents imbalanced datasets, potentially leading to more robust and accurate results.

Each experiment was repeated 10 times to test the performance of the algorithm. For every repetition, the training and test sets were redefined, ensuring that the results are not dependent on a single partition of the data. We employed a grid search method to tune the hyperparameters and obtain the best parameter settings. For the drop-out mechanism introduced in the model, we similarly applied a grid search to tune the drop-out rate, to identify the most effective rate for preventing overfitting while preserving the model's generalization capabilities.

6.2.3 Competing Methods

We conduct a comparative analysis of *MPXGAT* against three competing methodologies, specifically *GraphSAGE*, *GATNE* and *MultiplexSAGE*.

GraphSAGE [61]. *GraphSAGE* is an inductive node embedding algorithm that leverages node features to learn an embedding function capable of generalizing to unseen nodes. It was originally designed for single-layer network embeddings, so in our experiments, we apply it without distinguishing between intra-layer and inter-layer links.

GATNE [30]. *GATNE* is an embedding algorithm for attributed heterogeneous networks, encompassing multigraphs with diverse node and edge types. To adapt *GATNE* to our specific task, we introduced two categories of edges, denoted as intra-layer and inter-layer links. This adjustment allows us to create a multigraph that the algorithm can learn to embed effectively.

MultiplexSAGE [49]. *MultiplexSAGE* represents an extension of the GraphSAGE algorithm, specifically tailored for embedding multiplex networks. Its key feature is the distinction made between inter-layer and intra-layer links that allows for the prediction of both intra-layer and inter-layer connectivity patterns.

Other relevant methods such as [56], [74], [94], [118], [160], [129] and [71] can not be used as competing methods, as they assume that each layer has the same number of nodes and that all inter-layer connections are known. Therefore, as we are interested in predicting both intra-layer and inter-layer links, these methods are not suitable benchmarks for evaluating the performance of *MPXGAT*.

6.2.4 Embedding Multiplex Networks

Our first analysis concerns the prediction of both intra-layer and inter-layer links when embedding a multiplex network. For each embedding, we eval-

uate the Area Under the Receiver Operating Characteristic (ROC) Curve (AUC), and take an average over the different repetitions as a performance metric. We consider the standard deviation as an indicator of statistical error. Table 6.2 on the next page provides an overview of the performances obtained with *MPXGAT*, *GraphSAGE*, *GATNE*, and *MultiplexSAGE*. We note that *GATNE* outperforms *GraphSAGE* and *MultiplexSAGE* in intra-layer link prediction, having a higher average AUC for all three datasets. However, *MPXGAT* has comparable performances with *GATNE* on the ff-ww-tt and the Drosophila dataset, while the latter performs better on the arXiv dataset. Yet, with regards to inter-layer link prediction, *MPXGAT* clearly performs better than all other algorithms, including *MultiplexSAGE*, which is explicitly designed for that task. Overall, if we consider the general performance of the methods without distinguishing intra- and inter-layer connections, our algorithm emerges as the best solution, as reported in Table 6.3 on the following page.

As described in Section 6.1, *MPXGAT* employs two distinct embeddings, dealing with the horizontal, i.e., intra-layer, and vertical, i.e., inter-layer, embeddings, respectively. In contrast, the other models rely on a single embedding that serves both tasks. As we will further investigate in the next section, it is this architectural difference that allows *MPXGAT* to predict with a certain reliability both intra-layer and inter-layer links.

6.2.5 Measure the impact of Horizontal Embeddings

We now establish the impact of the horizontal embeddings on the performance of *MPXGAT*. To do so, we conduct two experiments. In the first, we perform the embedding with *MPXGAT*, but instead of using MPXGAT-V, which leverages the horizontal embeddings generated by MPXGAT-H, for embedding the vertical network, we consider a standard GAT [152], thus ignoring the contribution of MPXGAT-H. Table 6.4 on page 184 reports the

Table 6.1: Dataset Information including the number of nodes, edges, and average degree for the largest connected component of each dataset (arXiv, Drosophila, and ff-tw-yt)

Dataset	Nodes	Edges	Avg. Degree
arXiv	19,310	20,738	1.07
Drosophila	11,867	5,171	0.44
ff-tw-yt	11,827	6,028	0.51

Table 6.2: Performance comparison on intra-layer and inter-layer link prediction across the three distinct datasets: ff-tw-yt, Drosophila, and arXiv. The assessment is based on the AUC metric, using the standard deviation as the error metric. The best performing tool is highlighted in boldface.

Algorithm	**ff-tw-yt**		**Drosophila**		**arXiv**	
	intra	*inter*	*intra*	*inter*	*intra*	*inter*
GraphSAGE	0.47 (± 0.02)	0.56 (± 0.02)	0.54 (± 0.02)	0.63 (± 0.02)	0.72 (± 0.02)	0.70 (± 0.01)
GATNE	**0.83 (± 0.01)**	0.47 (± 0.01)	**0.78 (± 0.01)**	0.55 (± 0.01)	**0.91 (± 0.01)**	0.63 (± 0.01)
MultiplexSAGE	0.48 (± 0.02)	0.62 (± 0.02)	0.51 (± 0.01)	0.77 (± 0.02)	0.71 (± 0.02)	0.83 (± 0.01)
MPXGAT	0.76 (± 0.06)	**0.83 (± 0.01)**	0.76 (± 0.05)	**0.86 (± 0.02)**	0.80 (± 0.02)	**0.84 (± 0.01)**

Table 6.3: Overall AUC performance calculated as weighted sums of the results shown in Table 6.2, based on the number of edges used to evaluate the models.

Algorithm	**ff-tw-yt**	**Drosophila**	**arXiv**
GraphSAGE	0.49 (± 0.02)	0.57 (± 0.01)	0.70 (± 0.01)
GATNE	0.72 (± 0.01)	0.69 (± 0.02)	0.72 (± 0.01)
MultiplexSAGE	0.52 (± 0.02)	0.61 (± 0.01)	0.79 (± 0.01)
MPXGAT	**0.78 (± 0.03)**	**0.80 (± 0.03)**	**0.82 (± 0.04)**

average AUC for the inter-layer link prediction in both configurations. We observe that neglecting the horizontal embeddings leads to worse performances across all datasets, suggesting that including the MPXGAT-V submodel increases the algorithm adaptability and predictive power.

To validate the significance of these findings we performed a Welch's T-test. The p-values are 6.00×10^{-7}, 9.10×10^{-10}, and 1.10×10^{-7} for the Drosophila, arXiv, and ff-tw-yt datasets, respectively, confirming the statistical significance of our finding.

In the second experiment, we keep the original model architecture but instead of providing MPXGAT-V with the horizontal embeddings generated by MPXGAT-H, we replace them with random embeddings, i.e., vectors whose components are random values.

Table 6.5 on the following page shows the results of the comparison with regards to the inter-layer link prediction. We note that for both the ff-tw-yt and the arXiv dataset, using the horizontal embeddings increases the performance of the prediction task, while we observe similar values of the average AUC for the Drosophila dataset. The statistical significance of this result is confirmed by the Welch's T-test, for which we obtain p-values equal to $6.10 \cdot 10^{-6}$, 0.75, $3.60 \cdot 10^{-4}$ for the arXiv, Drosophila, and ff-ww-tt datasets, respectively.

We conjecture that this outcome is due to the structure of the multiplex network, both in terms of the size of the different layers, i.e., the number of nodes laying on them, and of the connectivity patterns, both within the same layer (e.g., randomness, clustering and community organization, and across different ones, i.e., overlapping).

Table 6.4: Impact of excluding horizontal embeddings on inter-layer link prediction, evaluated in terms of AUC across the three datasets. The second model, which omits horizontal embeddings, exhibits lower performance across all datasets.

Algorithm	**ff-tw-yt**	**Drosophila**	**arXiv**
MPXGAT-V (layer GAT-V*)*	**0.83 (± 0.01)**	**0.86 (± 0.01)**	**0.84 (± 0.01)**
GAT (layer GAT*)*	0.72 (± 0.02)	0.78 (± 0.02)	0.78 (± 0.01)

Table 6.5: Results of the experiment investigating the impact of replacing meaningful horizontal embeddings with random embeddings. The performance is evaluated in terms of AUC. The model with random embeddings performs worse in two out of the three datasets compared to the model with actual embeddings, indicating a significant decrease in predictive accuracy.

Algorithm	**ff-tw-yt**	**Drosophila**	**arXiv**
MPXGAT-V (actual embedding)	**0.83 (± 0.01)**	0.86 (± 0.01)	**0.84 (± 0.01)**
MPXGAT-V (random embedding)	0.80 (± 0.02)	0.86 (± 0.01)	0.81 (± 0.01)

6.3 Discussion

In this chapter we have introduced *MPXGAT*, an attention based deep learning model for the embedding of multiplex graphs [19]. Through a comprehensive experimental analysis we showed that *MPXGAT* out-performs state-of-the-art competing algorithms.

The performance of *MPXGAT* is influenced by the structure of the multiplex network, both in terms of the size of the different layers and the connectivity patterns within and between them.

Further analysis would be beneficial to better understand how the performance of *MPXGAT* is impacted by certain network characteristics. Future work should focus on evaluating the impact of these parameters.

7 Conclusions

7.1 Summary of Contributions

This thesis has presented significant advancements in the field of graph theory, specifically targeting the subgraph isomorphism problem, motif discovery in temporal networks, and graph embedding for complex, multiplex graphs. The key contributions of this work are:

- **ArcMatch**: Developed as a high-performance subgraph matching algorithm for labeled graphs, ArcMatch leverages vertex and edge domain filtering techniques to enhance efficiency [22]. The experimental results demonstrated its superiority in handling real-world scenarios such as protein-protein interaction networks, co-authorship networks, and email networks. The introduction of path-based domain reduction and cost-based search process further optimized the performance, making ArcMatch a versatile tool for various graph analysis tasks.
- **MultiGraphMatch**: Introduced to address subgraph matching in multi-relational graphs, MultiGraphMatch employs a novel bit signature data structure for efficient indexing and filtering [107]. Its innovative processing order and symmetry breaking conditions have shown to outperform existing graph database systems in various synthetic

and real-world graphs. By allowing nodes and edges to be associated with multiple properties, MultiGraphMatch provides a more nuanced approach to subgraph matching, capturing the complexity of multi-relational data more effectively.

- **MODIT**: This algorithm was designed for motif discovery in temporal networks, a challenging task due to the chronological constraints imposed on edges. MODIT's ability to efficiently count motifs of any size has been validated through extensive experiments, proving its capability in handling large temporal networks [122]. The algorithm's design ensures that MODIT can manage the combinatorial explosion of possible subgraphs as the size of the motif increases, making it a powerful tool for temporal network analysis.

- **MPXGAT**: An attention-based deep learning model for multiplex graph embedding, MPXGAT utilizes Graph Attention Networks to capture complex intra-layer and inter-layer relationships. Its effectiveness in link prediction tasks was confirmed through comprehensive evaluations on multiple benchmark datasets, establishing MPXGAT as a leading approach in the field [19]. The model's ability to handle the intricacies of multiplex networks highlights its potential for broader applications in network analysis and predictive modeling.

7.2 Broader Impact

The advancements presented in this thesis have broad implications across various domains where graph-based data plays a crucial role. By improving the efficiency and scalability of subgraph matching and motif discovery algorithms, this work facilitates deeper insights and more robust analyses in fields such as bioinformatics, social network analysis, and computational

chemistry. Additionally, the development of advanced graph embedding techniques opens new avenues for applying machine learning to complex network structures, potentially leading to breakthroughs in predictive modeling and data mining.

Bioinformatics

In bioinformatics, the ability to efficiently match subgraphs and discover motifs can significantly enhance the understanding of biological processes and molecular interactions. The tools developed in this thesis can be used to identify functional modules within protein-protein interaction networks, helping to elucidate the roles of specific proteins in various biological pathways and disease mechanisms [128].

Social Network Analysis

For social network analysis, the algorithms and models proposed here enable the detection of community structures, influential nodes, and patterns of interaction within large social networks. These insights can inform strategies for marketing, information dissemination, and understanding social dynamics [156].

Computational Chemistry

In computational chemistry, the ability to match subgraphs representing molecular structures can facilitate the search for compounds with desired properties, aiding in drug discovery and materials science. The graph embedding techniques developed can help predict molecular interactions and properties, accelerating the pace of innovation in these fields [120].

7.3 Future Work

Despite the significant progress made in this thesis, several avenues for future research remain. These include:

- **Scalability Enhancements**: Further improving the scalability of the proposed algorithms to handle even larger graphs and more complex queries, potentially through parallel and distributed computing techniques. This includes exploring cloud-based solutions and high-performance computing frameworks to manage the computational demands of large-scale graph analysis [126].

- **Algorithm Optimization**: Refining the algorithms to reduce computational overhead and memory consumption, enabling their application to real-time graph analytics. Techniques such as incremental updating, approximate matching, and memory-efficient data structures could be investigated to enhance performance [103].

- **Integration with Graph Databases**: Exploring the integration of the developed algorithms with existing graph database systems to leverage their querying capabilities and storage efficiency. This integration can provide a seamless workflow for users, combining the strengths of advanced subgraph matching and motif discovery with the robust storage and retrieval functionalities of graph databases [3].

- **Extended Applications**: Applying the developed techniques to new domains and types of graphs, such as heterogeneous networks, dynamic graphs, and hypergraphs, to further validate their versatility and effectiveness. Research into domain-specific adaptations of the algorithms could yield new insights and applications in areas like financial networks, brain connectivity analysis, and ecological networks [113].

- **Interdisciplinary Research**: Collaborating with experts from other fields to adapt and apply the proposed methods to specific real-world problems, fostering interdisciplinary innovation and discovery. For example, working with biologists to analyze genetic interaction networks or with sociologists to study social behavior patterns using the developed tools [98].

- **User-Friendly Software Tools**: Developing user-friendly software tools and libraries that encapsulate the algorithms and models proposed in this thesis. This would involve creating graphical user interfaces and APIs that make the tools accessible to non-expert users, facilitating broader adoption and impact [90].

- **Benchmarking and Standardization**: Establishing standard benchmarks and evaluation protocols for subgraph matching, motif discovery, and graph embedding. This would help in objectively comparing different methods and advancing the field through collaborative efforts [106].

- **Explainability and Interpretability**: Enhancing the explainability and interpretability of the proposed models, particularly the deep learning-based approaches. This is crucial for their application in sensitive domains like healthcare and finance, where understanding the rationale behind predictions and analyses is essential [157].

7.4 Final Remarks

This thesis has advanced the state of the art in subgraph matching, motif discovery, and graph embedding, addressing critical challenges in the analysis of complex networks. The proposed algorithms and models have demonstrated superior performance and broad applicability, contributing valuable

tools and insights to the scientific community. The findings and methodologies presented herein lay a strong foundation for ongoing research and development in graph theory and its myriad applications.

The journey of this research has underscored the importance of addressing both theoretical and practical aspects of algorithm design. The balance between computational efficiency and real-world applicability has been a guiding principle throughout this work. As we move forward, the integration of these algorithms into broader systems and their application to diverse datasets will continue to unlock new possibilities and drive innovation.

In conclusion, the advancements made in this thesis not only contribute to the academic discourse but also hold significant potential for practical applications. The continued exploration of graph-based techniques promises to yield further breakthroughs, enhancing our ability to understand and leverage complex networks in science, technology, and beyond.

Bibliography

[1] C Aggarwal and K Subbian. "Evolutionary network analysis: A survey". In: *ACM Computing Surveys (CSUR)* 47.1 (2014), p. 10.

[2] K. Alaoui. "A Categorization of RDF Triplestores". In: *Proceedings of the 4th International Conference on Smart City Applications*. SCA '19. Casablanca, Morocco: Association for Computing Machinery, 2019. ISBN: 9781450362894. DOI: 10.1145/3368756.3369047.

[3] R. Angles. "The Property Graph Database Model". In: *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, May 21-25, 2018*. Vol. 2100. CEUR Workshop Proceedings. Cali, Colombia: CEUR-WS.org, 2018, pp. 1–10.

[4] R. Angles, J.B. Antal, and A. et al. Averbuch. "The LDBC Social Network Benchmark". In: *CoRR* abs/2001.02299 (2020), pp. 1–160. URL: http://arxiv.org/abs/2001.02299.

[5] Antonino Aparo et al. "Fast subgraph matching strategies based on pattern-only heuristics". In: *Interdisciplinary Sciences: Computational Life Sciences* 11.1 (2019), pp. 21–32.

[6] Michael Azmy et al. "Matching entities across different knowledge graphs with graph embeddings". In: *arXiv preprint arXiv:1903.06607* (2019).

[7] Davide Bacciu et al. "A gentle introduction to deep learning for graphs". In: *Neural Networks* 129 (2020), pp. 203–221.

[8] Alexandru T Balaban. “Applications of graph theory in chemistry”. In: *Journal of chemical information and computer sciences* 25.3 (1985), pp. 334–343.

[9] Mahendran Balasubramanian et al. “Heuristics for graph isomorphism detection”. In: *IEEE Transactions on Knowledge and Data Engineering* 31.4 (2018), pp. 742–755.

[10] A. Barabási and A. Réka. “Emergence of Scaling in Random Networks”. In: *Science* 286.5439 (1999), pp. 509–512. DOI: 10.1126/science.286.5439.509.

[11] C. Barbagallo et al. “VECTOR: An Integrated Correlation Network Database for the Identification of CeRNA Axes in Uveal Melanoma”. In: *Genes* 12.7 (2021), pp. 1–19. ISSN: 2073-4425. DOI: 10.3390/genes12071004.

[12] Federico Battiston, Vincenzo Nicosia, and Vito Latora. “Structural measures for multiplex networks”. In: *Physical Review E* 89.3 (2014), p. 032804.

[13] Ali Behrouz and Farnoosh Hashemi. “CS-MLGCN: Multiplex Graph Convolutional Networks for Community Search in Multiplex Networks”. In: *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*. CIKM ’22. Atlanta, GA, USA: Association for Computing Machinery, 2022, pp. 3828–3832. ISBN: 9781450392365. DOI: 10.1145/3511808.3557572. URL: https://doi.org/10.1145/3511808.3557572.

[14] Ali Behrouz and Margo Seltzer. *Anomaly Detection in Multiplex Dynamic Networks: from Blockchain Security to Brain Disease Prediction*. 2022. arXiv: 2211.08378 [cs.LG].

[15] Michele Berlingerio et al. “Foundations of multidimensional network analysis”. In: *2011 international conference on advances in social networks analysis and mining*. IEEE. 2011, pp. 485–489.

[16] Michele Berlingerio et al. “Mining Graph Evolution Rules”. In: *ECML/PKDD 2009*. Bled, Slovenia, 2009, pp. 115–130.

[17] F. Bi et al. “Efficient Subgraph Matching by Postponing Cartesian Products”. In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD ’16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1199–1214. DOI: 10.1145/2882903.2915236.

[18] Stefano Boccaletti et al. “The structure and dynamics of multilayer networks”. In: *Physics reports* 544.1 (2014), pp. 1–122.

[19] Marco Bongiovanni, Luca Gallo, **Roberto Grasso**, and Alfredo Pulvirenti. *MPXGAT: An Attention based Deep Learning Model for Multiplex Graphs Embedding*. 2024. arXiv: 2403.19246 [cs.LG]. URL: https://arxiv.org/abs/2403.19246.

[20] V. Bonnici and R. Giugno. "On the Variable Ordering in Subgraph Isomorphism Algorithms". In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 14.1 (2017), pp. 193–203. DOI: 10.1109/TCBB.2016.2515595.

[21] Vincenzo Bonnici and Rosalba Giugno. "A subgraph isomorphism algorithm and its application to biochemical data". In: *BMC Bioinformatics* 14.7 (2013), pp. 173–183.

[22] Vincenzo Bonnici, **Roberto Grasso**, Giovanni Micale, Antonio Di Maria, Dennis Shasha, Alfredo Pulvirenti, and Rosalba Giugno. "ArcMatch: high-performance subgraph matching for labeled graphs by exploiting edge domains". In: *Data Mining and Knowledge Discovery* (Aug. 2024), pp. 1–54. DOI: 10.1007/s10618-024-01061-8.

[23] Vincenzo Bonnici et al. "Enhancing graph database indexing by suffix tree structure". In: *IAPR International Conference on Pattern Recognition in Bioinformatics*. Springer. 2010, pp. 195–203.

[24] V. Carletti et al. "Challenging the Time Complexity of Exact Subgraph Isomorphism for Huge and Dense Graphs with VF3". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 40.4 (2018), pp. 804–818. DOI: 10.1109/TPAMI.2017.2696940.

[25] Vincenzo Carletti, Pasquale Foggia, and Mario Vento. "Performance comparison of five exact graph matching algorithms on biological databases". In: *International Conference on Image Analysis and Processing*. Springer. 2013, pp. 409–417.

[26] Vincenzo Carletti et al. "Comparing performance of graph matching algorithms on huge graphs". In: *Pattern Recognition Letters* 134 (2020), pp. 58–67.

[27] Vincenzo Carletti et al. "Introducing VF3: A new algorithm for subgraph isomorphism". In: *International Workshop on Graph-Based Representations in Pattern Recognition*. Springer. 2017, pp. 128–139.

[28] K M Carley et al. "Toward an interoperable dynamic network analysis toolkit". In: *Decision Support Systems* 43.4 (2007), pp. 1324–1347.

[29] A Casteigts et al. "Time-Varying Graphs and Dynamic Networks". In: *Ad-hoc, Mobile, and Wireless Networks*. 2011, pp. 346–359.

[30] Yukuo Cen et al. "Representation learning for attributed multiplex heterogeneous network". In: *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 2019, pp. 1358–1368.

[31] Fenxiao Chen et al. "Graph representation learning: a survey". In: *APSIPA Transactions on Signal and Information Processing* 9 (2020), e15.

[32] Donatello Conte et al. "Thirty years of graph matching in pattern recognition". In: *International Journal of Pattern Recognition and Artificial Intelligence* 18.03 (2004), pp. 265–298.

[33] Stephen A Cook. "The complexity of theorem-proving procedures". In: *Proceedings of the third annual ACM symposium on Theory of computing*. STOC '71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. DOI: 10.1145/800157.805047. URL: https://doi.org/10.1145/800157.805047.

[34] L.P. Cordella et al. "A (sub)graph isomorphism algorithm for matching large graphs". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.10 (2004), pp. 1367–1372. DOI: 10.1109/TPAMI.2004.75.

[35] Luigi P Cordella et al. "A new algorithm for subgraph isomorphism detection". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.10 (2001), pp. 1367–1372.

[36] Luigi Pietro Cordella et al. "An improved algorithm for matching large graphs". In: *3rd IAPR-TC15 workshop on graph-based representations in pattern recognition*. 2001, pp. 149–159.

[37] Emanuele Cozzo et al. *Multiplex networks: basic formalism and structural properties*. Vol. 10. Springer, 2018.

[38] J Crawford and T Milenkovic. "ClueNet: Clustering a temporal network based on topological similarity rather than denseness". In: *PLOS ONE* 13.5 (2018), pp. 1–25.

[39] Nicholas Dahm et al. "Efficient subgraph matching using topological node feature constraints". In: *Pattern Recognition* 48.2 (2015), pp. 317–330.

[40] Manlio De Domenico et al. "Identifying Modular Flows on Multilayer Networks Reveals Highly Overlapping Organization in Interconnected Systems". In: *Phys. Rev. X* 5 (1 Mar. 2015), p. 011027. DOI: 10.1103/PhysRevX.5.011027. URL: https://link.aps.org/doi/10.1103/PhysRevX.5.011027.

[41] Manlio De Domenico et al. “Mathematical formulation of multilayer networks”. In: *Physical Review X* 3.4 (2013), p. 041022.

[42] Manlio De Domenico et al. “Structural reducibility of multilayer networks”. In: *Nature communications* 6.1 (2015), p. 6864.

[43] Rina Dechter, David Cohen, et al. *Constraint processing*. San Francisco, California: Morgan Kaufmann, 2003.

[44] A. Deutsch et al. “TigerGraph: A Native MPP Graph Database”. In: *ArXiv* abs/1901.08248 (2019), pp. 1–28.

[45] Mark Dickison, Matteo Magnani, and Luca Rossi. *Multilayer Social Networks*. Cambridge University Press, 2016.

[46] A Divakaran and A Mohan. “Temporal Link Prediction: A Survey”. In: *New Generation Computing* 38 (2020), pp. 213–258.

[47] Paul Erdos and Alfréd Rényi. “On random graph”. In: *Publicationes Mathematicate* 6 (1959), pp. 290–297.

[48] N. Francis, A. Green, and P. et al Guagliardo. “Cypher: An Evolving Query Language for Property Graphs”. In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD ’18. Houston, TX, USA: Association for Computing Machinery, 2018, pp. 1433–1445. DOI: `10.1145/3183713.3190657`.

[49] Luca Gallo, Vito Latora, and Alfredo Pulvirenti. “MultiplexSAGE: A Multiplex Embedding Algorithm for Inter-Layer Link Prediction”. In: *IEEE Transactions on Neural Networks and Learning Systems* (2023).

[50] Erich Gamma et al. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[51] Hongyang Gao and Shuiwang Ji. “Graph u-nets”. In: *international conference on machine learning*. PMLR. 2019, pp. 2083–2092.

[52] Michael R Garey and David S Johnson. *Computers and intractability: a guide to the theory of NP-completeness*. W.H. Freeman and Company, 1979.

[53] M Génois and A Barrat. “Can co-location be used as a proxy for face-to-face contacts?” In: *EPJ Data Science* 7.11 (2018).

[54] Michelle Girvan and Mark EJ Newman. “Community structure in social and biological networks”. In: *Proceedings of the National Academy of Sciences* 99.12 (2002), pp. 7821–7826.

[55] R. Giugno et al. “GRAPES: A Software for Parallel Searching on Biological Graphs Targeting Multi-Core Architectures”. In: *PLOS ONE* 8.10 (2013), pp. 1–11.

[56] Maoguo Gong et al. “Heuristic 3D Interactive Walks for Multilayer Network Embedding”. In: *IEEE Transactions on Knowledge and Data Engineering* 34.7 (2020), pp. 3309–3321.

[57] Palash Goyal, Sujit Rokka Chhetri, and Arquimedes Canedo. “Dyngem: Deep embedding method for dynamic graphs”. In: *Proceedings of the 2018 IEEE International Conference on Big Data (Big Data)*. IEEE. 2018, pp. 3645–3648.

[58] Joshua A. Grochow and Manolis Kellis. “Network Motif Discovery Using Subgraph Enumeration and Symmetry-Breaking”. In: *Research in Computational Molecular Biology*. Vol. 4453. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 92–106. ISBN: 978-3-540-71681-5. DOI: 10.1007/978-3-540-71681-5_7.

[59] Martin Grohe. “word2vec, node2vec, graph2vec, x2vec: Towards a theory of vector embeddings of structured data”. In: *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 2020, pp. 1–16.

[60] Aditya Grover and Jure Leskovec. “node2vec: Scalable feature learning for networks”. In: *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 2016, pp. 855–864.

[61] Will Hamilton, Zhitao Ying, and Jure Leskovec. “Inductive representation learning on large graphs”. In: *Advances in neural information processing systems* 30 (2017).

[62] William L Hamilton, Rex Ying, and Jure Leskovec. “Representation learning on graphs: Methods and applications”. In: *arXiv preprint arXiv:1709.05584* (2017).

[63] M. Han et al. “Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together”. In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD ’19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 1429–1446. DOI: 10.1145/3299869.3319880.

[64] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. “Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases”. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 337–348. DOI: 10.1145/2463676.2465300.

[65] Wook-Shin Han et al. "iGraph in action: performance analysis of disk-based graph indexing techniques". In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 2011, pp. 1241–1242.

[66] 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*. SIGMOD '08. New York, NY, USA: Association for Computing Machinery, 2008, pp. 405–418. DOI: 10.1145/1376616.1376660.

[67] T Hiraoka et al. "Modeling temporal networks with bursty activity patterns of nodes and links". In: *Phys. Rev. Research* 2.2 (2020), p. 023073.

[68] T Hogg and K Lerman. "Social dynamics of Digg". In: *EPJ Data Science* 1.1 (2012), pp. 1–26.

[69] P Holme and J Saramaki. *Temporal network theory*. Cham: Springer International Publishing, 2019.

[70] P. Holme and J. Saramaki. "Temporal networks". In: *Physics Reports* 519.3 (2012), pp. 97–125.

[71] Zhichao Huang et al. "MR-GCN: Multi-Relational Graph Convolutional Networks based on Generalized Tensor Product." In: *IJCAI*. Vol. 20. 2020, pp. 1258–1264.

[72] Y Hulovatyy, H Chen, and T Milenkovic. "Exploring the structure and function of temporal networks with dynamic graphlets". In: *Bioinformatics* 31.12 (2015), pp. i171–i180.

[73] V. Ingalalli, D. Ienco, and P. Poncelet. "SuMGra: Querying Multigraphs via Efficient Indexing". In: *Database and Expert Systems Applications*. Cham: Springer International Publishing, 2016, pp. 387–401. ISBN: 978-3-319-44403-1.

[74] Vassilis N Ioannidis, Antonio G Marques, and Georgios B Giannakis. "Tensor graph convolutional networks for multi-relational and robust learning". In: *IEEE Transactions on Signal Processing* 68 (2020), pp. 6535–6546.

[75] L Isella et al. "What's in a crowd? Analysis of face-to-face behavioral networks". In: *Journal of Theoretical Biology* 271.1 (2011), pp. 166–180.

[76] Priya B Jain et al. “Composed solutions of synchronized patterns in multiplex networks of Kuramoto oscillators”. In: *Chaos: An Interdisciplinary Journal of Nonlinear Science* 33.10 (2023).

[77] Hawoong Jeong et al. “Lethality and centrality in protein networks”. In: *Nature* 411.6833 (2001), pp. 41–42.

[78] M.S. Katari, D. Shasha, and S. Tyagi. *Statistics is Easy: Case Studies on Real Scientific Datasets*. Synthesis Lectures on Mathematics & Statistics. Cham: Springer International Publishing, 2021. ISBN: 978-3-031-01305-8. DOI: 10.1007/978-3-031-02433-7.

[79] Foteini Katsarou, Nikos Ntarmos, and Peter Triantafillou. “Hybrid algorithms for subgraph pattern queries in graph databases”. In: *2017 IEEE International Conference on Big Data (Big Data)*. IEEE. 2017, pp. 656–665.

[80] Foteini Katsarou, Nikos Ntarmos, and Peter Triantafillou. “Performance and scalability of indexed subgraph query processing methods”. In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1566–1577.

[81] P. Keila and David Skillicorn. “Structure in the Enron Email Dataset”. In: *Computational and Mathematical Organization Theory* 11 (2005), pp. 183–199.

[82] Shima Khoshraftar and Aijun An. “A survey on graph representation learning methods”. In: *arXiv preprint arXiv:2204.01855* (2022).

[83] H. Kim et al. “Fast subgraph query processing and subgraph matching via static and dynamic equivalences”. In: *The VLDB Journal* 32.2 (2022), pp. 343–368. ISSN: 0949-877X. DOI: 10.1007/s00778-022-00749-x.

[84] Hyunjoon Kim et al. “Versatile Equivalences: Speeding up Subgraph Query Processing and Subgraph Matching”. In: *Proceedings of the 2021 International Conference on Management of Data*. 2021, pp. 925–937.

[85] Thomas N Kipf and Max Welling. “Semi-supervised classification with graph convolutional networks”. In: *arXiv preprint arXiv:1609.02907* (2016).

[86] L Kovanen et al. “Temporal motifs in time-dependent networks”. In: *Journal of Statistical Mechanics: Theory and Experiment* 2011.11 (2011), P11005.

[87] M Kuramochi and G Karypis. “Frequent subgraph discovery”. In: *Data Mining and Knowledge Discovery* 11.3 (2005), pp. 2005–2008.

[88] Vito Latora, Vincenzo Nicosia, and Giovanni Russo. *Complex networks: principles, methods and applications*. Cambridge University Press, 2017.

[89] Jinsoo Lee et al. “An in-depth comparison of subgraph isomorphism algorithms in graph databases”. In: *Proceedings of the VLDB Endowment* 6.2 (2012), pp. 133–144.

[90] J. Leskovec and R. Sosič. “SNAP: A General-Purpose Network Analysis and Graph-Mining Library”. In: *ACM Transactions on Intelligent Systems and Technology (TIST)* 8.1 (2016), p. 1.

[91] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. “Graphs over time: densification laws, shrinking diameters and possible explanations”. In: *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. 2005, pp. 177–187.

[92] Meng Liu, Hongyang Gao, and Shuiwang Ji. “Towards deeper graph neural networks”. In: *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*. 2020, pp. 338–348.

[93] P Liu, AR Benson, and M Charikar. “Sampling Methods for Counting Temporal Motifs”. In: (2019), pp. 294–302.

[94] Weiyi Liu et al. “Principled multilayer network embedding”. In: *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*. IEEE. 2017, pp. 134–141.

[95] Giorgio Locicero et al. “TemporalRI: A Subgraph Isomorphism Algorithm for Temporal Networks”. In: *Complex Networks & Their Applications IX* (Jan. 2021), pp. 675–687. DOI: 10.1007/978-3-030-65351-4_54.

[96] Linyuan Lü and Tao Zhou. “Link prediction in complex networks: A survey”. In: *Physica A: statistical mechanics and its applications* 390.6 (2011), pp. 1150–1170.

[97] L Lv et al. “PageRank centrality for temporal networks”. In: *Physics Letters A* 383.12 (2019), pp. 1215–1222.

[98] Cheng-Yu Ma and Chung-Shou Liao. “A review of protein–protein interaction network alignment: From pathway comparison to global alignment”. In: *Computational and Structural Biotechnology Journal* 18 (2020), pp. 2647–2656.

[99] Alan K Mackworth. “Consistency in networks of relations”. In: *Artificial intelligence* 8.1 (1977), pp. 99–118.

[100] N Masuda and P Holme. “Small inter-event times govern epidemic spreading on networks”. In: *Phys. Rev. Research* 2.2 (2020), p. 023163.

[101] N Masuda and R Lambiotte. *A Guide to Temporal Networks*. 2nd. Europe: World Scientific, 2020.

[102] Rudolf Mathon. “A note on the graph isomorphism counting problem”. In: *Information Processing Letters* 8.3 (1979), pp. 131–136. ISSN: 0020-0190. DOI: 10.1016/0020-0190(79)90004-8.

[103] C. McCreesh, P. Prosser, and J. Trimble. “The Glasgow Subgraph Solver: Using Constraint Programming to Tackle Hard Subgraph Isomorphism Problem Variants”. In: *Graph Transformation*. Cham: Springer International Publishing, 2020, pp. 316–324. ISBN: 978-3-030-51372-6.

[104] Ciaran McCreesh et al. “When subgraph isomorphism is really hard, and why this matters for graph databases”. In: *Journal of Artificial Intelligence Research* 61 (2018), pp. 723–759.

[105] Brendan D. McKay and Adolfo Piperno. “Practical graph isomorphism, II”. In: *Journal of Symbolic Computation* 60 (2014), pp. 94–112. ISSN: 0747-7171. DOI: 10.1016/j.jsc.2013.09.003.

[106] A. Mhedhbi et al. “LSQB: A Large-Scale Subgraph Query Benchmark”. In: *Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. GRADES-NDA ’21. Virtual Event, China: Association for Computing Machinery, 2021. ISBN: 9781450384773. DOI: 10.1145/3461837.3464516.

[107] Giovanni Micale, Antonio Di Maria, **Roberto Grasso**, Vincenzo Bonnici, Alfredo Ferro, Dennis Shasha, Rosalba Giugno, and Alfredo Pulvirenti. “MultiGraphMatch: a subgraph matching algorithm for multigraphs”. Under revision in ACM Transactions on Knowledge Discovery from Data (TKDD). 2024.

[108] Ron Milo et al. “Network motifs: simple building blocks of complex networks”. In: *Science* 298.5594 (2002), pp. 824–827.

[109] Prabhat Mishra et al. "Hardware trojan detection using ATPG and model checking: A survey". In: *IET Computers & Digital Techniques* 8.6 (2014), pp. 274–287.

[110] J. Monteiro, F. Sá, and J. Bernardino. "Experimental Evaluation of Graph Databases: JanusGraph, Nebula Graph, Neo4j, and TigerGraph". In: *Applied Sciences* 13.9 (2023), pp. 1–16. ISSN: 2076-3417. DOI: 10.3390/app13095770.

[111] J.D. Moorman et al. "Subgraph Matching on Multiplex Networks". In: *IEEE Transactions on Network Science and Engineering* 8.2 (2021), pp. 1367–1384. DOI: 10.1109/TNSE.2021.3056329.

[112] Mark Newman. *Networks: An Introduction.* Oxford University Press, 2010.

[113] Mark EJ Newman. "The structure and function of complex networks". In: *SIAM review* 45.2 (2003), pp. 167–256.

[114] P Panzarasa, T Opsahl, and KM Carley. "Patterns and dynamics of users' behavior and interaction: Network analysis of an online community". In: *Journal of the American Society for Information Science and Technology* 60.5 (2009), pp. 911–932.

[115] A Paranjape, A R Benson, and J Leskovec. "Motifs in Temporal Networks". In: *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining.* WSDM '17. 2017, pp. 601–610.

[116] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. "Deepwalk: Online learning of social representations". In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining.* 2014, pp. 701–710.

[117] J Petit et al. "Random walk on temporal networks with lasting edges". In: *Phys. Rev. E* 98.5 (2018), p. 052307.

[118] Meng Qu et al. "An Attention-Based Collaboration Framework for Multi-View Network Representation Learning". In: *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management.* CIKM '17. Singapore, Singapore: Association for Computing Machinery, 2017, pp. 1767–1776. ISBN: 9781450349185. DOI: 10.1145/3132847.3133021. URL: https://doi.org/10.1145/3132847.3133021.

[119] Meng Qu et al. *An Attention-based Collaboration Framework for Multi-View Network Representation Learning.* 2017. arXiv: 1709.06636 [cs.SI].

[120] JW Raymond and P Willett. “Heuristics for similarity searching of chemical graphs using a maximum common edge subgraph algorithm”. In: *Journal of Computer-Aided Molecular Design* 16.6 (2002), pp. 521–533.

[121] Pedro Ribeiro and Fernando Silva. “G-Tries: a data structure for storing and finding subgraphs”. In: *Data Mining and Knowledge Discovery* 28.2 (2014), pp. 337–377. ISSN: 1573-756X. DOI: 10.1007/s10618-013-0303-4.

[122] **Roberto Grasso** et al. “MODIT: MOtif DIscovery in Temporal Networks”. In: *Frontiers in Big Data* 4 (2022). ISSN: 2624-909X. DOI: 10.3389/fdata.2021.806014. URL: https://www.frontiersin.org/journals/big-data/articles/10.3389/fdata.2021.806014.

[123] L E C Rocha, N Masuda, and P Holme. “Sampling of temporal networks: Methods and biases”. In: *Phys. Rev. E* 96.5 (2017), p. 052302.

[124] G Rossetti and R Cazabet. “Community Discovery in Dynamic Networks: A Survey”. In: *ACM Comput. Surv.* 51.2 (2018), pp. 1–37.

[125] Sherif Sakr and Ghazi Al-Naymat. “Graph indexing and querying: a review”. In: *International Journal of Web Information Systems* (2010).

[126] M Schmidt and D Hermelin. “Large scale graph mining using parallel and distributed computing”. In: *IEEE Transactions on Knowledge and Data Engineering* 21.8 (2009), pp. 1210–1222.

[127] Haichuan Shang et al. “Taming verification hardness: an efficient algorithm for testing subgraph isomorphism”. In: *Proceedings of the VLDB Endowment* 1.1 (2008), pp. 364–375.

[128] Roded Sharan and Trey Ideker. “Modeling protein-protein interaction networks via a global regularizer”. In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 3.3 (2006), pp. 230–235.

[129] Yu Shi et al. “mvn2vec: Preservation and collaboration in multi-view network embedding”. In: *arXiv preprint arXiv:1801.06597* (2018).

[130] Kai Shu et al. “User identity linkage across online social networks: A review”. In: *Acm Sigkdd Explorations Newsletter* 18.2 (2017), pp. 5–17.

[131] E A Singh and H Cherifi. “Centrality-Based Opinion Modeling on Temporal Networks”. In: *IEEE Access* 8 (2020), pp. 1945–1961.

[132] C. Solnon. “AllDifferent-based filtering for subgraph isomorphism”. In: *Artificial Intelligence* 174.12 (2010), pp. 850–864. ISSN: 0004-3702. DOI: 10.1016/j.artint.2010.05.002.

[133] Chris Stark et al. "BioGRID: a general repository for interaction datasets". In: *Nucleic Acids Research* 34.Database issue (2006), pp. D535–D539. DOI: 10.1093/nar/gkj109.

[134] Ulrich Stelzl et al. "A human protein-protein interaction network: a resource for annotating the proteome". In: *Cell* 122.6 (2005), pp. 957–968.

[135] Shixuan Sun and Qiong Luo. "In-Memory Subgraph Matching: An In-depth Study". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 1083–1098. DOI: 10.1145/3318464.3380581.

[136] X Sun et al. "TM-Miner: TFS-Based Algorithm for Mining Temporal Motifs in Large Temporal Network". In: *IEEE Access* 7 (2019), pp. 49778–49789.

[137] Y. Sun et al. "A subgraph matching algorithm based on subgraph index for knowledge graph". In: *Frontiers of Computer Science* 16.3 (2021), p. 163606. ISSN: 2095-2236. DOI: 10.1007/s11704-020-0360-y.

[138] Zhao Sun et al. "Efficient Subgraph Matching on Billion Node Graphs". In: *Proceedings of the VLDB Endowment* 5.9 (2012).

[139] Michael Szell, Renaud Lambiotte, and Stefan Thurner. "Multirelational organization of large-scale social networks in an online world". In: *Proceedings of the National Academy of Sciences* 107.31 (2010), pp. 13636–13641.

[140] Damian Szklarczyk et al. "STRING v11: protein–protein association networks with increased coverage, supporting functional discovery in genome-wide experimental datasets". In: *Nucleic acids research* 47.D1 (2019), pp. D607–D613.

[141] Jian Tang et al. "Line: Large-scale information network embedding". In: *Proceedings of the 24th international conference on world wide web*. 2015, pp. 1067–1077.

[142] Rui Tang et al. "Interlayer link prediction based on multiple network structural attributes". In: *Computer Networks* 203 (2022), p. 108651.

[143] Rui Tang et al. "Interlayer link prediction in multiplex social networks: An iterative degree penalty algorithm". In: *Knowledge-Based Systems* 194 (2020), p. 105598.

[144] S. Timón-Reina, M. Rincón, and R. Martínez-Tomás. "An overview of graph databases and their applications in the biomedical domain". In: *Database* 2021 (2021), pp. 1–22. ISSN: 1758-0463. DOI: 10.1093/database/baab026.

[145] M Tizzani et al. "Epidemic spreading and aging in temporal networks with memory". In: *Phys. Rev. E* 98.6 (2018), p. 062315.

[146] M Torricelli, M Karsai, and L Gauvin. "weg2vec: Event embedding for temporal networks". In: *Scientific Reports* 10 (2020), p. 7164.

[147] I Tsalouchidou et al. "Temporal betweenness centrality in dynamic graphs". In: *Int. J. Data Sci. Anal.* 9 (2020), pp. 257–272.

[148] Kun Tu et al. *Network Classification in Temporal Networks Using Motifs*. July 2018.

[149] Johan Ugander, Lars Backstrom, and Jon Kleinberg. "Subgraph frequencies: mapping the empirical and extremal geography of large graph collections". In: May 2013, pp. 1307–1318. DOI: 10.1145/2488388.2488502.

[150] J.R. Ullmann. "An Algorithm for Subgraph Isomorphism". In: *J. ACM* 23.1 (1976), pp. 31–42. ISSN: 0004-5411. DOI: 10.1145/321921.321925.

[151] J.R. Ullmann. "Bit-Vector Algorithms for Binary Constraint Satisfaction and Subgraph Isomorphism". In: *ACM J. Exp. Algorithmics* 15 (2011), pp. 1–64. ISSN: 1084-6654. DOI: 10.1145/1671970.1921702.

[152] Petar Veličković et al. "Graph attention networks". In: *arXiv preprint arXiv:1710.10903* (2017).

[153] R. Vilaça, F. Cruz, and R." Oliveira. "On the Expressiveness and Trade-Offs of Large Scale Tuple Stores". In: *On the Move to Meaningful Internet Systems, OTM 2010*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 727–744. ISBN: 978-3-642-16949-6.

[154] O E Williams, F Lillo, and V Latora. "Effects of memory on spreading processes in non-Markovian temporal networks". In: *New Journal of Physics* 21.4 (2019), p. 043028.

[155] Carl Yang et al. "Multisage: Empowering gcn with contextualized multi-embeddings on web-scale multipartite networks". In: *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*. 2020, pp. 2434–2443.

[156] Jaewon Yang and Jure Leskovec. "Defining and evaluating network communities based on ground-truth". In: *Knowledge and Information Systems* 42.1 (2015), pp. 181–213.

[157] Rex Ying et al. "GNNExplainer: Generating Explanations for Graph Neural Networks". In: *Advances in neural information processing systems* 32 (2019).

[158] Stéphane Zampelli, Yves Deville, and Christine Solnon. "Solving subgraph isomorphism problems with constraint programming". In: *Constraints* 15.3 (2010), pp. 327–353. ISSN: 1572-9354. DOI: 10.1007/s10601-009-9074-3.

[159] Chuxu Zhang et al. "Heterogeneous graph neural network". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (2019), pp. 793–803.

[160] Hongming Zhang et al. "Scalable multiplex network embedding." In: *IJCAI*. Vol. 18. 2018, pp. 3082–3088.

[161] Zhitao Zhang et al. "General framework for scalable transductive and inductive network embedding". In: *IEEE Transactions on Knowledge and Data Engineering* 32.6 (2019), pp. 1094–1107.

[162] Peixiang Zhao and Jiawei Han. "Large scale graph mining using parallel and distributed computing: Applications and algorithms". In: *IEEE Transactions on Knowledge and Data Engineering* 22.9 (2010), pp. 1295–1310.

[163] Quan Zheng and David Skillicorn. *Social networks with rich edge semantics*. London: Taylor & Francis, 2017.

[164] Qingyun Zhou et al. "Dynamic network embedding by modeling triadic closure process". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 32.1 (2018).

List of Tables

2.1 Main notation used for simple graphs. 38
2.2 Main notation used for multigraphs. 43
2.3 Main notation used for temporal graphs. 47
2.4 Main notation used for multiplex graphs. 51

3.1 Functional features of the compared approaches. 81
3.2 Average running times in seconds over the PPI benchmark. . . 83
3.3 Average running times over the co-autorship network. 86
3.4 Number of finished instances for the PPI benchmark. 97
3.5 Average running times (and standard deviation) over varying number of vertex labels on the PPI benchmark when searching for all the occurrences. 98
3.6 Average running times (and standard deviation) over the fully labelled PPI benchmark 99
3.7 Number of finished instances for the vertex-labelled PPI benchmark on searching for the first 100k occurrences. 100
3.8 Average running times over the fully labelled PPI benchmark when searching for all the occurrences. 101

3.9 Average running times (and standard deviation) over the vertex-labelled PPI benchmark on searching for the first 100k occurrences. 102

4.1 Performance of MultiGraphMatch (MGM), Sumgra, Neo4J and Memgraph (Mem) on a dataset of queries run on synthetic networks. 134
4.2 Median, p-value of significance and 90% confidence interval of the query-by-query running time difference (in seconds) between MultiGraphMatch (MGM) and the other tools. 137
4.3 Main features of the three real networks used in the experiments. 143
4.4 Experiments on a dataset of queries without WHERE clause. . 143
4.5 Experiments on a dataset of queries having WHERE clauses. . 144
4.6 Median, p-value of significance and 90% confidence interval of the query-by-query running time difference (in seconds) between MultiGraphMatch (MGM) and the other tools on a dataset of queries without WHERE clause. 144
4.7 Median, p-value of significance and 90% confidence interval of the query-by-query running time difference (in seconds) between MultiGraphMatch (MGM) and the other tools on a dataset of queries with WHERE clause. 145
4.8 Main features of the 4 LDBC networks used in scalability tests. 147
4.9 Times (in seconds) required to read the 4 LDBC networks used for scalability tests. 147
4.10 Times (in seconds) required to compute processing order of query edges, symmetry breaking conditions, compatibility domains and matching. 148

5.1 Datasets of temporal networks used for the experiments. . . . 165
5.2 Experiments on Dataset 1 (pt. 1). 166

5.3 Experiments on Dataset 1 (pt. 2). 167
5.4 Experiments on Dataset 1 (pt. 3). 168
5.5 Comparison between MODIT and Paranjape's algorithm. . . . 169

6.1 Dataset Information including the number of nodes, edges, and average degree for the largest connected component of each dataset (arXiv, Drosophila, and ff-tw-yt) 182
6.2 Performance comparison on intra-layer and inter-layer link prediction across the three distinct datasets. 182
6.3 Overall AUC performance. 182
6.4 Impact of excluding horizontal embeddings on inter-layer link prediction. 184
6.5 Results of the experiment investigating the impact of replacing meaningful horizontal embeddings with random embeddings. . 184

List of Figures

1.1 Example of motif Δ-occurrence in a temporal graph T, given Δ. 24
1.2 Example of application of the Temporal Motif Search (TMS) problem. 24

2.1 A use case example of subgraph searching over a co-authorship network. 37
2.2 Example of query and target graph. 37
2.3 Domains, and their transformation by means of reduction techniques. 37
2.4 Example of multigraph with actors and directors of the movie industry. 42
2.5 Toy example of SubMultigraph Matching (SMM) 42
2.6 A toy example of a multiplex network. 50

3.1 A 3-ways division of the neighborhood of a vertex v given a partial ordering θ_i. 64
3.2 Communities in the co-authorship network 81

3.3 Comparison of *ArcMatch* with those systems that allow an arbitrarily large number of returned matches on the Protein-Protein Interaction (PPI) graphs with both vertex and edges labels. 85
3.4 Comparison of *ArcMatch* with other systems that allow an arbitrarily large number of returned matches on the Protein-Protein Interaction (PPI) graphs with vertex labels but ignoring edge labels. 85
3.5 Comparison of *ArcMatch* with other systems that allow an arbitrarily large number of returned matches on the co-authorship network. 86
3.6 Algorithm with the lowest average running time, grouping queries by the number of query vertices and the number of vertex labels. 91
3.7 Scalability analysis for the synthetic benchmark built by means of the Barabási-Albert model. 91
3.8 Scalability analysis for the synthetic benchmark built by means of the Erdos model 92
3.9 Scalability analysis for the synthetic benchmark built by means of the Forest Fire model. 92

4.1 Indexing data structures associated with the target 113
4.2 Example of symmetry breaking condition on nodes and edges. 116
4.3 Computation of compatibility domains. 120
4.4 Boxplots of the query-by-query running time differences between MultiGraphMatch (MGM) and the other compared tools for the dataset of queries run on the uniform synthetic networks. 136
4.5 Boxplots of the query-by-query running time differences between MultiGraphMatch (MGM) and the other compared tools for the dataset of queries run on power-law synthetic networks. 136

4.6 Boxplots of running times of MultiGraphMatch (MGM), MultiGraphMatch with no Bit Matrix (MGM-NoBM), MultiGraphMatch with Random Ordering (MGM-RO) and MultiGraphMatch with no Bit Matrix and Random Ordering (MGM-NoBM-RO). 138
4.7 Boxplots of the query-by-query running time differences between MultiGraphMatch (MGM) and the state-of-the-art algorithms for a dataset of queries without WHERE clause . . . 145
4.8 Boxplots of the query-by-query running time differences between MultiGraphMatch (MGM) and the state-of-the-art algorithms for a dataset of queries having WHERE clauses . . . 147
4.9 Queries of the LSQB benchmark used for scalability experiments. 150
4.10 Running times of MultiGraphMatch for scalability tests. . . . 151

5.1 Example of construction of the canonical form. 157

6.1 The structure of the MPXGAT model. 173

List of Algorithms

1 Matching procedure 56
2 ArcConsistency 64
3 PathReduction 65
4 VerifyPath 71
5 VerifyPathDSF 72
6 RefineDomains 73
7 VerifyPath 74
8 SearchOccurrences 75
9 SubgraphMatching 124
10 FindCandidates 126
11 MODIT 157
12 MODIT:UpdateBounds 157
13 MODIT:RecursiveSearch 159