DISCOVERY OF TEMPORAL GRAPH FUNCTIONAL DEPENDENCIES

DISCOVERY OF TEMPORAL GRAPH FUNCTIONAL DEPENDENCIES

BY

LEVIN NORONHA, B.A.Sc.

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTING AND SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

Master of Science (2022) McMaster University

(Computing and Software) Hamilton, Ontario, Canada

TITLE:	Discovery of Temporal Graph Functional Dependencies
AUTHOR:	Levin Noronha B.A.Sc. (Computer Science), McMaster University, Hamilton, Canada
SUPERVISOR:	Fei Chiang
NUMBER OF PAGES:	xi, 59

Abstract

Real-world graphs are dynamic and evolve over time. Data quality in evolving graphs is essential to downstream decision making and fact checking. This work studies the discovery of Temporal Graph Functional Dependencies (TGFDs), a recently defined class of data quality rules for enforcing consistency over evolving graphs [4]. TGFDs impose topological and attribute dependency constraints *over a period of time*. We define minimality and support for TGFDs and formalize the TGFD discovery problem. Defining TGFDs manually is a laborious task and requires domain expertise. Hence, we introduce TGFDMiner, a sequential algorithm that discovers minimal and frequent TGFDs. We define various optimizations for TGFDMiner that improve runtime efficiency by pruning redundant candidates. Using real-world and synthetic data, we experimentally evaluate the efficiency, effectiveness, scalability of TGFDMiner, and the utility of TGFDs.

Dedicated to my parents, my love, and my friends.

Acknowledgements

I would like to express my most sincere gratitude to my supervisor, Dr. Fei Chiang, for her continued support and guidance during every step of my Masters journey. Her technical knowledge and expertise were an immense asset to me when I was identifying a research problem, publishing my first research paper, and writing this thesis. I would like to thank my colleague, Morteza Alipourlangouri, for sharing his valuable knowledge during our technical discussions.

I would also like to thank my comittee members, Dr. Christopher Anand and Dr. Lingyang Chu, for their time in reviewing my thesis, providing valuable feedback, and conducting my thesis defense.

Lastly, I would like to thank my parents and Kruthiga for their ongoing love and support. Without them, I would not be the person that I am today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Databases have traditionally stored data using relational tables, where each table represents an entity, each row represent an instance of an entity, and each column represents an attribute of an entity. However, relational data lacks modeling for relationships between entities. In contrast, graphs are a non-relational form of data storage that treat relationships between entities as data itself. Graph data has become increasing popular driven by the desire to model social networks and knowledge graphs such as DBpedia [19], Yago [21], Wikidata [24], and IMDB [1].

Knowledge graphs and social networks are highly dynamic and subject to frequent changes. These changes can be useful or erroneous. We analyzed dynamic real-world knowledge graphs such as DBpedia and IMDB and profiled the changes that occurred in these graphs over time. In DBpedia, we found an average of 8 inconsistencies per entity instance. In IMDB, which has monthly data for 26 months, information about an entity instance remained consistent for an average period of 5 months. This shows

the existence of consistencies and inconsistencies in evolving graphs.

The use of functional dependencies (FDs) to enforce data quality in relational data has been studied extensively [8, 9, 17, 18]. FDs have also been extended to include temporal constraints [2]. Since these FDs are defined over relational data, they cannot impose topological constraints. Recent works have also defined graph dependencies [14, 10, 16, 20], *i.e.* FDs over graphs, which incorporate topological constraints and attribute value constraints. However, graph dependencies have only been defined over static graphs and do not consider temporal graphs, *i.e.* graphs that change over time. Association rules have also been recently defined to capture topological changes in temporal graphs [22]. However, these rules do not enforce any attribute values constraints. Temporal graph functional dependencies, or TGFDs, have been recently defined to enforce consistency in temporal graphs [4] by imposing topological, attribute value, and temporal constraints.

There are many real world examples that highlight the need for temporal constraints in graph dependencies. For example, (i) a U.S. president can only hold office for up to 8 years, (ii) whereas U.S. governors are required to be residents of their respective states and also to have held the U.S. citizenship for a minimum of 5 years. Similar examples exist in many domains, but they cannot be captured by any existing graph dependencies [14, 16, 20].

Example 1: A hospital database stores timely data about patients, medications, and symptoms. This data can be modelled as a graph by representing instances of entities such as patients, medications, and symptoms as vertices and representing all relationships between these vertices as a edges. For example, Figure 1.1(b) depicts various timestamped graphs that comprise a temporal graph $\mathcal{G}_T$ for hospital data. At

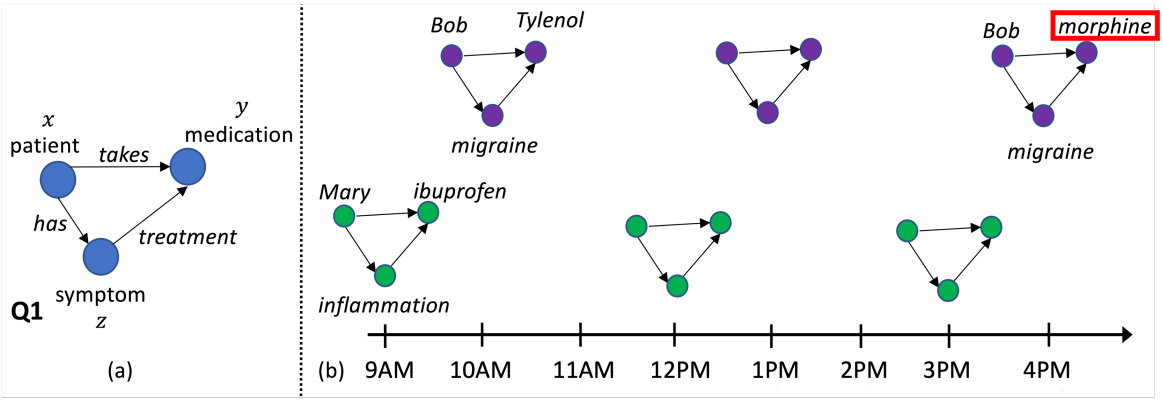


Figure 1.1: Graph pattern and temporal graph.

each timestamp, we see various instances of entities such as patient, medication and symptom.

A graph pattern is a graph that matches a set of subgraphs in a temporal graph via pattern matching or subgraph isomorphism. In Figure 1.1(a), we define a graph pattern Q_1 to represent the relationship between entities `patient`, `symptom`, and `medication`. In Figure 1.1(b), we see various subgraphs in the temporal graph $\mathcal{G}_T$ that match pattern Q_1.

In order to enforce consistency in the matches of a graph pattern, we need to define a functional dependency (FD) and specify a time interval such that for all pairs of matches of a graph pattern that are within a time interval, the FD must hold. An FD $X \rightarrow Y$ is a relationship between two sets of attributes X and Y, where Y is considered to be functionally dependent on X if for all instances of X, the value of X uniquely determines the value of Y. For example, to enforce consistency in the matches of Q_1, we would define an FD $\{\texttt{patient}, \texttt{symptom}\} \rightarrow \texttt{medication}$ and specify a time interval of 3 to 6 hours such that FD $\{\texttt{patient}, \texttt{symptom}\} \rightarrow$ `medication` must hold for all pairs of matches of Q_1 that have a temporal distance

between 3 to 6 hours. In Figure 1.1(b), we see that matches for `patient` Mary are consistent because she takes the `medication` Ibuprofen at 9AM, 12PM, and 3PM, i.e. every 3 to 6 hours, to treat her inflammation. However, the data for `patient` Bob is not consistent because at 4PM, Bob takes morphine which creates an inconsistency with his previous dosage of Tylenol 3 hours ago at 1PM. □

Enforcing consistency via dependencies containing topological, attribute value, and temporal constraints is critical to maintaining data quality in temporal graphs. However, defining these dependencies manually is not feasible because real-world temporal graphs are large and highly dynamic. For example, DBpedia contains roughly 5M vertices and nearly 15M edges. Other real-world temporal graphs such as IMDB are updated weekly, making it very onerous to keep dependencies up-to-date. Furthermore, defining dependencies over a dataset also requires domain expertise. To make TGFDs useful, we propose TGFDMiner, a sequential algorithm for discovering TGFDs over temporal graphs. Discovering TGFDs is challenging when compared to existing dependencies because: (a) unlike relational dependencies, TGFDs need to enforce topological constraints, and (b) unlike graph dependencies, TGFDs need to enforce temporal constraints.

1.1 Contributions

1. We define metrics for TGFD discovery such as minimality and support, and also introduce and formalize the TGFD discovery problem.
2. We present TGFDMiner, a sequential algorithm that discovers a set of frequent and minimal TGFDs.

3. We also describe four optimizations for TGFDMiner that use axioms to efficiently prune redundant candidates. In our evaluation, we show that our four optimizations achieve an average decrease in runtime of 19.4%.

4. Finally, we evaluate TGFDMiner on large, real-world and synthetic, temporal graphs to show that our algorithm is scalable and efficient. For instance, we evaluate scalability by varying graph size, and compare against an existing graph dependency mining algorithm as a baseline. Our evaluation shows that TGFDMiner achieves a comparable runtime. We also present examples of real-world TGFDs discovered from IMDB.

1.2 Thesis Organization

This thesis is organized as follows. In Chapter 2, we introduce a list of preliminary concepts that readers need to be familiar with to understand our work. In Chapter 3, we aim to help readers contextualize the role of our work with respect to related works. In Chapter 4, we introduce and formalize the TGFD discovery problem, as well as define metrics such minimality and support with respect to TGFD discovery. In Chapter 5, we describe our TGFD discovery algorithm, TGFDMiner, for finding minimal and frequent TGFDs, and we also describe various optimization for improving the performance of our algorithm. In Chapter 6, we present results from the evaluation of our algorithm on real-world and synthetic datasets. In Chapter 7, we provide some concluding remarks for this thesis and present directions for future work.

Chapter 2

Preliminaries

We introduce preliminary definitions for temporal graphs and temporal graph functional dependencies (TGFDs).

2.1 Temporal Graphs

A temporal graph $\mathcal{G}_T$, as defined in [4], consists of T snapshots of directed graph G_t, where $t \in [1, T]$, denoted by $\mathcal{G}_T = \{G_1, ..., G_T\}$. A directed graph $G_t = (V_t, E_t, L_t, F_{A_t})$ consists of (i) a vertex set V_t, (ii) an edge set E_t, (iii) a label $L_t(v)$ for each $v \in V_t$, (iv) a label $L_t(e)$ for each $e \in E_t$, and (v) for each vertex v, $F_{A_t}(v)$ is a tuple $(A_{1,t} = a_1, ..., A_{n,t} = a_n)$, where a_i is a constant value of an attribute A_i of vertex v at time t.

Example 2: Figure 1.1b illustrates a temporal graph $\mathcal{G}_T$ consisting of several timestamped snapshots. Each snapshot G_t consists of: (a) vertices that represent instances

of entities such as patients, medications, and symptoms, and (b) edges, where each edge represent the relationship between two entity instances, (c) vertex labels such as `patient`, `medication`, `symptom`, (d) edge labels `takes`, `has`, and `treatment` which specify that a `patient` x takes `medication` y to treat `symptom` z, and (e) constant values such as "Bob" and "Mary" for attribute `patient.name`, "migraine" and "inflammation" for attribute `symptom.name`, "Tylenol" and "ibuprofen" for attribute `medication.brand_name`. □

2.2 Graph Pattern

A graph pattern $Q[\bar{x}] = (V_Q, E_Q, L_Q, \mu)$, as defined in [4, 16], is a directed, connected graph consisting of (i) a vertex set V_Q, (ii) and edge set (E_Q), (iii) a label $L(v)$ for each $v \in V_Q$, (iv) a label $L(e)$ for each $e \in E_Q$, and (v) a bijective function μ which maps each $v \in V_Q$ to a variable in $x \in \bar{x}$, i.e. $\mu(v) = x$. We use, interchangeably, the notation $Q[\bar{x}]$ and Q, and $\mu(v)$ and x, when it is clear based on the context.

Example 3: Figure 1.1a illustrates a graph pattern Q_1, or $Q_1[\bar{x}]$, with a set of variables $\bar{x} = [x, y, z]$, where μ maps x to `patient`, y to `medication`, and z to `symptom`. The edge labels `takes`, `has`, and `treatment` specify that a `patient` x takes `medication` y to treat `symptom` z. □

2.3 Temporal Graph Pattern Matching

A match $h_t(\bar{x})$, as defined in [4], between a snapshot G_t of $\mathcal{G}_T$ and a pattern Q is a subgraph $G'_t = (V'_t, E'_t, L'_t, F'_{A_t})$ that is isomorphic to Q. We denote a match in vectorized form $h_t(\bar{x})$, for $h_t(x)$ for all $x \in \bar{x}$. That is, h_t is a bijective function from V_Q to V'_t such that:

(i) for each $v \in V_Q$, $L_Q(v) = L'_t(h_t(v))$, and

(ii) for each $e = (v, v') \in E_Q$, there exists an edge $e' = (h_t(v), h_t(v'))$ in G'_t such that $L_Q(e) = L'_t(h_t(e'))$.

If vertex label $L_Q(v)$ (resp. edge label $L_Q(e)$) is a wildcard '_', then $h_t(v)$ (resp. $h_t(e)$) can match any vertex label (resp. edge label) in G_t.

2.4 Functional Dependency (FD)

In relational data, a functional dependency $X \rightarrow Y$ is defined over attributes within a table. An FD $X \rightarrow Y$ specifies a relationship between two sets of attributes X and Y, where Y is considered to be functionally dependent on X if for all instances of X, the value of X uniquely determines the value of Y.

In the context of temporal graphs, an FD $X \rightarrow Y$ is defined over all pairs of matches $(h_i(\bar{x}), h_j(\bar{x}))$ that lie within a time interval (see semantics in subsection 2.5.2). For all pairs $(h_i(\bar{x}), h_j(\bar{x}))$ within a time interval, Y is considered to be functionally dependent on X if for all instances of X in each pair, the value of X uniquely determines the value of Y.

2.5 Temporal Graph Functional Dependency (TGFD)

TGFDs have been recently introduced by [4] as a graph data dependency to enforce the following constraints on temporal graphs:

- *Topological constraint*: imposed by a graph pattern $Q[\bar{x}]$.
- *Attribute value constraint*: imposed by an attribute dependency $X \rightarrow Y$.
- *Temporal constraint*: imposed by time interval Δ, such that dependency $X \rightarrow Y$ is expected to hold over all pairs of matches of $Q[\bar{x}]$ that lie within a time interval Δ.

In this section, we describe the syntax and semantics of TGFDs. We also describe axioms that are relevant to the discovery of TGFDs.

2.5.1 Syntax

A TGFD σ is a triple $(Q[\bar{x}], \Delta, X \rightarrow Y)$, as defined in [4], that consists of:

- a graph pattern $Q[\bar{x}]$;
- a time interval $\Delta = (p, q)$, with $p \leq q$, where p represents the lower time bound and q represents the upper time bound;
- a dependency $X \rightarrow Y$, where X and Y are two (possibly empty) set of literals in $\bar{x}$.

Both X and Y are sets of literals, and each literal in a set can be one of two forms: (a) constant literal, denoted as $x.A = c$, or (b) variable literal, denoted as $x.A = y.B$. For both constant and variable literals, x and y are variables in $\bar{x}$, *i.e.* $x, y \in \bar{x}$, A and B are attributes, and c is a constant value.

For simplicity, TGFDMiner will only consider, without loss of generality, TGFDs in a normal form, i.e. TGFDs with a dependency $X \rightarrow l$, where l is a single literal.

Example 4: In Figure 1.1b, to enforce consistency in Bob's matches, we would need to enforce a topological constraint, an attribute value constraint, and a dependency constraint. A topological constraint can be enforced using graph pattern Q_1 from Figure 1.1a. An attribute value constraint can be enforced via a dependency $X_1 \rightarrow l_1$, where $X_1 = \{x.\texttt{name}, z.\texttt{name}\}$ and $l_1 = y.\texttt{brand\_name}$. A temporal constraint can be enforced by specifying a time interval $\Delta_1 = (3, 6 \text{ hours})$. In summary, we would need to define a TGFD $\sigma_1 = (Q_1, \Delta_1, X_1 \rightarrow l_1)$ to enforce consistency in Bob's matches. □

2.5.2 Semantics

The semantics of a TGFD, as defined in [4], state that a given a set of pairs of matches that satisfies a topological constraint specified by a graph pattern $Q[\bar{x}]$ and a temporal constraint specified by a time interval Δ, we enforce an attribute value constraint specified via a dependency $X \rightarrow Y$ over all pairs of matches that are within an interval Δ.

A TGFD $\sigma = (Q[\bar{x}], \Delta, X \rightarrow Y)$ specifies an interval of time Δ in which a topological constraint Q and attribute dependency $X \rightarrow Y$ must hold. We denote

$(h_i(\bar{x}), h_j(\bar{x}))$ to be a pair of matches of Q in G_i and G_j respectively, where $i \leq j$ and $i, j \in [1, T]$. A match $(h_i(\bar{x}), h_j(\bar{x}))$ satisfies a constant literal $x.A = c$ if both $v = h_i(x)$ and $v' = h_j(x)$ contain attribute A, such that $v.A = v'.A = c$ and $|j - i| \in \Delta$. Similarly, a match $(h_i(\bar{x}), h_j(\bar{x}))$ satisfies a variable literal $x.A = y.B$ if $v = h_i(x)$ contains attribute A and $v' = h_j(y)$ contains attribute B such that $v.A = v'.B$ and $|j - i| \in \Delta$.

Temporal graph $\mathcal{G}_T$ satisfies σ, denoted by $\mathcal{G}_T \models \sigma$, if *all pairs of matches* $(h_i(\bar{x}), h_j(\bar{x}))$ of Q in $\mathcal{G}_T$, with $|j - i| \in \Delta$, satisfy σ, denoted by $(h_i(\bar{x}), h_j(\bar{x})) \models \sigma$. A pair of matches $(h_i(\bar{x}), h_j(\bar{x}))$ satisfies σ, denoted by $(h_i(\bar{x}), h_j(\bar{x})) \models \sigma$, whenever $(h_i(\bar{x}), h_j(\bar{x})) \models X$ implies $(h_i(\bar{x}), h_j(\bar{x})) \models Y$.

Lastly, a temporal graph $\mathcal{G}_T$ satisfies a set of TGFDs Σ, denoted by $\mathcal{G}_T \models \Sigma$, if *for each* $\sigma \in \Sigma$, $\mathcal{G}_T \models \sigma$.

Example 5: In Figure 1.1b, to enforce consistency in Bob's matches, we would need to enforce a dependency $X_1 \rightarrow l_1$ consisting of $X_1 = \{x.\texttt{name}, z.\texttt{name}\}$ and $l_1 = y.\texttt{brand\_name}$, over all pairs of Bob's matches shown in Figure 1.1b, that lie within the time interval $\Delta_1 = (3, 6$ hours$)$. □

2.5.3 Axioms

We present a list of axioms for TGFDs, defined by [4], that will be used as a basis for optimizations during TGFD discovery.

Axiom 1: *(Literal Augmentation)[4] If* $\sigma' = (Q[\bar{x}], \Delta, X' \rightarrow Y)$, $\sigma = (Q[\bar{x}], \Delta, X \rightarrow Y)$, *and* $X' \subseteq X$, *then* $\sigma' \models \sigma$. □

According Armstrong's axiom of augmentation for relational FDs, an FD $X \rightarrow Y$ can be inferred by another FD $X' \rightarrow Y$ if X is a superset of X'. Similarly, we prune all candidate TGFDs $\sigma = (Q[\bar{x}], \Delta, X \rightarrow Y)$ if there exists an already discovered TGFD $\sigma' = (Q[\bar{x}], \Delta, X' \rightarrow Y)$, such that X is a superset of X'.

Axiom 2: *(Pattern Augmentation)[4] If* $\sigma' = (Q'[\bar{x}'], \Delta, X \rightarrow Y)$*, and* $\sigma = (Q[\bar{x}], \Delta, X \rightarrow Y)$*,* $Q' \subseteq Q$*, then* $\sigma' \models \sigma$*.* □

We prune all candidate TGFDs $\sigma = (Q[\bar{x}], \Delta, X \rightarrow Y)$ if there exists an already discovered TGFD $\sigma' = (Q'[\bar{x}'], \Delta, X \rightarrow Y)$, such that Q is a superset of Q'.

Axiom 3: *(Interval Containment)[4] If* $\sigma = (Q[\bar{x}], \Delta, X \rightarrow Y)$*, and* $\sigma' = (Q[\bar{x}], \Delta', X \rightarrow Y)$*,* $\Delta' \subseteq \Delta$*, then* $\sigma \models \sigma'$*.* □

We prune all candidate TGFDs $\sigma' = (Q[\bar{x}], \Delta', X \rightarrow Y)$ if there exists an already discovered TGFD $\sigma = (Q[\bar{x}], \Delta, X \rightarrow Y)$, such that Δ is a superset of Δ'.

In summary, we have described various concepts in this section that are important to understanding TGFD discovery. We have described temporal graphs, graph patterns, matching graph patterns to temporal graphs, TGFD syntax and semantics, and also axioms relevant to TGFD discovery.

Chapter 3

Related Work

We summarize a list of work related to functional dependencies over relational data and graphs, association rules for graphs, and also the discovery of various aforementioned rules and dependencies.

3.1 Dependencies

In this section, we describe functional dependencies and their various extensions and address their similarities and differences to TGFDs.

3.1.1 Functional Dependencies (FDs)

Traditional FDs are fundamental to the study of databases and have been studied extensively within the context of data quality rules [8, 9, 17, 18]. Over the years, FDs have been extended as conditional functional dependencies (CFDs) [11], and denial

constraints [5]. FDs have been extended with temporal constraints in the form of temporal functional dependencies (TFDs) [2]. FDs have also been extended to graph data in the form of graph functional dependencies (GFDs) [16]. However, there does not exist a class of functional dependencies that is defined over temporal graphs and also includes temporal constraints.

3.1.2 Keys for graphs

Keys are a special case of traditional FDs. Graph keys, or GKeys, have been defined to uniquely identify entities in graphs [10]. Ontological graph keys, or OGKs, have also been defined to extend GKeys with ontological subgraph matching between entity labels and an external ontology. However, both GKeys and OGKs are defined over static graphs and do not consider temporal graphs.

3.1.3 Dependencies for graphs

Traditional FDs were extended to graphs in the form of graph functional dependencies (GFDs) [16]. GFDs enforce topological constraints and attribute dependencies over static graphs. A GFD consists of a graph pattern $Q[\bar{x}]$ and an attribute dependency $X \rightarrow l$, where $Q[\bar{x}]$ imposes a topological constraint and $X \rightarrow l$ imposes a dependency constraint on all attributes found in matches of $Q[\bar{x}]$. Recently, graph entity dependencies (GEDs) [14] were defined to subsume GFDs and GKeys by supporting literal equality of an *id* literal between keys of two matches of a graph pattern. Numeric graph dependencies (NGDs) [13] have been defined to extend GFDs with linear

arithmetic expressions and comparison predicates. However, none of the aforementioned graph dependencies consider temporal graphs. TGFDs extend GFDs to impose temporal constraints over temporal graphs.

3.1.4 Temporal constraints over FDs

Temporal constraints over traditional FDs have been defined in the form of temporal functional dependencies, or TFDs [2]. TFDs are defined over relational tables where every tuple has a timestamp attribute. A TFD is of the form $X \wedge \Delta \rightarrow Y$ and it enforces the dependency $X \rightarrow Y$ over all pairs of tuples where the difference between timestamps is within Δ. However, TFDs only enforce temporal constraints over relational data and do not consider graphs. In contrast, TGFDs enforce temporal constraints over temporal graphs.

3.2 Association Rules (ARs)

In this section, we describe traditional association rules (ARs) and their various extensions. For each rule, we compare the similarities and differences of each to TGFDs.

3.2.1 Traditional ARs

ARs have different semantics than FDs. An AR, as defined in [3], is an implication of the form $X \Rightarrow Y$, that holds over a set of tuples or a relational table, where the *antecedent* X and the *consequent* Y are a set of literals. $X \Rightarrow Y$ is said to hold over a

set of tuples if a certain percentage of tuples that contain X also contain Y. However, ARs, as defined by [3], do not consider temporal constraints or graph data.

3.2.2 Rules for evolving topology in graphs

Works such as GERM[7] and EvoMine[23] have defined *evolution rules* that describe topological changes in evolving graphs. These topological changes include edge insertions and deletions and also edge and vertex relabelling. However, these rules do not enforce any constraints over attributes, which are an important feature of real-world knowledge graphs.

3.2.3 ARs for graphs

ARs have been extended to static graphs in the form of graph pattern association rules (GPARs)[15]. GPARs use a graph pattern as the antecedent and single connecting edge as the consequent. GPARs do not consider attribute value constraints, temporal graphs, or temporal constraints. Recently, graph temporal association rules (GTARs)[22] were defined to enforce temporal constraints over temporal graphs. GTARs use general graph patterns as the antecedent and consequent. As a temporal constraint, GTARs specify that a match of the consequent pattern must occur within an interval of time following the match of the antecedent pattern. GTARs subsume GPARs and evolution rules as special cases. Even though GTARs impose temporal constraints over graphs, GTARs are ARs and their semantics are different from TGFDs.

3.3 Discovery Algorithms

Our discovery algorithm is closely related to the GFD discovery algorithm [12]. The GFD discovery algorithm generates graph patterns with up to k edges, such that $k \geq 2$ and k is a user input integer. Following the generation of each new pattern, the algorithm defines attribute value constraints by generating functional dependencies over the attributes in the pattern. Thus, GFD discovery performs pattern generation and dependency generation in an interleaved manner. Similarly, TGFDMiner also interleaves pattern generation and dependency generation. However, unlike the GFD discovery algorithm, TGFDMiner further interleaves dependency generation and identifies time intervals where the dependency is expected to hold.

In summary, we have provided a brief overview of related work, their shortcomings, and their differences to our work. We have described how our work is unique and how it addresses the shortcomings of related work.

Chapter 4

Discovery Metrics

In this chapter, we define minimality and support with respect to TGFDs. We also introduce and formalize the TGFD discovery problem.

4.1 Minimality

We are interested in discovering TGFDs that are non-trivial and simplified. A simplified TGFD has the smallest possible graph pattern and dependency and cannot be implied by any other TGFD.

Non-trivial. A TGFD $\sigma = (Q[\bar{x}], \Delta, X \rightarrow l)$ is considered *trivial* if X is false or if $l \in X$.

Simplified. A TGFD $\sigma_1 = (Q_1[\bar{x}], \Delta_1, X_1 \rightarrow l_1)$ *simplifies* another TGFD $\sigma_2 = (Q_2[\bar{x}], \Delta_2, X_2 \rightarrow l_2)$, denoted by $\sigma_1 \preceq \sigma_2$, if there exists an isomorphism function f from Q_1 to a subgraph of Q_2, i.e. $Q_1 \subseteq Q_2$, such that (1) $f(\tilde{v}_1) = \tilde{v}_2$, where $\tilde{v}_i$

denotes the interest vertex of Q_i; (2) $f(X_1) \subseteq X_2$ and $f(l_1) = l_2$; and (3) $\Delta_1 \subseteq \Delta_2$ and $\nexists \Delta' \subset \Delta_1$ where $\sigma' = (Q_1, \Delta', X_1 \rightarrow l_1) \models \mathcal{G}_T$. A TGFD σ is considered simplified if $\mathcal{G}_T \models \sigma$ and there exists no TGFD σ' such that $\mathcal{G}_T \models \sigma'$ and $\sigma' \preceq \sigma$. Simplified TGFDs are an extension of reduced GFDs [12].

Example 6: In Figure 1.1, a TGFD $\sigma' = (Q'_1, \Delta, X \rightarrow l)$, where Q' is derived from Q by removing the `treatment` edge, would simplify the TGFD $\sigma = (Q_1, \Delta, X \rightarrow l)$, i.e. $\sigma' \preceq \sigma$. □

Minimal. A TGFD σ is considered *minimal*, if it is both non-trivial and simplified. Given a temporal graph $\mathcal{G}_T$ and TGFD set Σ, we can say Σ is minimal if there does not exist any redundant TGFD $\sigma' \in \Sigma$, such that removing σ' from Σ creates a new set Σ' that is no longer equivalent to Σ.

4.2 Support

Support measures for dependencies over static graphs consider the frequency of individual pattern matches [16]. However, the temporal aspect of TGFDs requires us to consider the frequency of all pairs of matches that exists in $\mathcal{G}_T$ and lie within a time interval Δ. In this section, we define support for TGFDs and prove that TGFD support is not monotonic.

4.2.1 TGFD Support

To measure the support of a TGFD $\sigma = (Q, \Delta, X \rightarrow l)$, we consider all pairs of matches of Q that lie within Δ. We consider a pair of matches $(h_i(\bar{x}), h_j(\bar{x}))$ to be

an *occurrence* of σ in $\mathcal{G}_T$ if $(h_i(\bar{x}), h_j(\bar{x})) \models \sigma$. The set of all occurrences of σ in $\mathcal{G}_T$ is denoted by $\mathcal{O}(\sigma, \mathcal{G}_T)$. TGFD support, as defined in Equation 4.2.1, measures the ratio of the number of actual occurrences of σ in $\mathcal{G}_T$ to the number of all possible occurrences of σ in $\mathcal{G}_T$. $|S|$ is the number of unique entity instances *i.e.* the number of unique matches of X among all matches of Q. For each entity instance, $\binom{|T|+1}{2}$ counts the number of all possible occurrences of that entity instance that could exist in $\mathcal{G}_T$. A TGFD σ is considered frequent if for some threshold θ, $\mathsf{supp}(\sigma, \mathcal{G}_T) \geq \theta$.

$$\mathsf{supp}(\sigma, \mathcal{G}_T) = \frac{|\mathcal{O}(\sigma, \mathcal{G}_T)|}{|S|\binom{|T|+1}{2}} \tag{4.2.1}$$

4.2.2 Anti-monotonic

Similar to the support measures defined for GFDs and GTARs, TGFD support is also anti-monotonic.

Lemma 1: *For* TGFDs σ_1, σ_2*, temporal graph* $\mathcal{G}_T$*, if* $\sigma_1 \preceq \sigma_2$*, then* $\mathsf{supp}(\sigma_1, \mathcal{G}_T) \geq \mathsf{supp}(\sigma_2, \mathcal{G}_T)$. □

Proof Sketch. We need to show that if $\sigma_1 \preceq \sigma_2$, then $|\mathcal{O}(\sigma_1, \mathcal{G}_T)| \geq |\mathcal{O}(\sigma_2, \mathcal{G}_T)|$. For fixed Δ, any occurrence $(h_{t_i}(\bar{x}), h_{t_j}(\bar{x}))$ of σ_2 is also an occurrence of σ_1, i.e., $\mathcal{O}(\sigma_2, \mathcal{G}_T) \subseteq \mathcal{O}(\sigma_1, \mathcal{G}_T)$. Therefore, $|\mathcal{O}(\sigma_1, \mathcal{G}_T)| \geq |\mathcal{O}(\sigma_2, \mathcal{G}_T)|$, and TGFD support is anti-monotonic.

4.3 Problem Definition

The TGFD discovery problem involves searching a temporal graph $\mathcal{G}_T$ for frequent graph patterns $Q[\bar{x}]$, defining minimal functional dependencies $X \rightarrow l$ over attributes associated with each $Q[\bar{x}]$, and finding the largest interval of time Δ where $X \rightarrow l$ holds over all pairs of matches of $Q[\bar{x}]$. We limit the size of graph patterns using an integer k as an upper-bound, *i.e.* a pattern can have at most k edges. We discover k-bounded graph patterns because pattern matching is expensive and larger patterns tend to be less frequent and harder to interpret [12]. We also seek to define dependencies with the smallest number of literals that hold over the largest interval of time. We formalize the problem of TGFD discovery as follows.

Given a temporal graph $\mathcal{G}_T$, a pattern-size upper-bound $k \geq 2$, and a support threshold θ, we compute a set Σ of k-bounded, minimal TGFDs such that $\mathcal{G}_T \models \Sigma$ and $\forall \sigma \in \Sigma$, $\mathsf{supp}(\sigma, \mathcal{G}_T) \geq \theta$.

In summary, we have defined minimality and support for TGFDs and also used those definitions to introduce and formalize the TGFD discovery problem. In the next chapter, we present a solution to the TGFD discovery problem.

Chapter 5

TGFD Discovery Algorithm

In this chapter, we present TGFDMiner, a sequential algorithm for discovering a set Σ of minimal and frequent TGFDs in a temporal graph $\mathcal{G}_T$. First, we provide an overview of the TGFDMiner architecture, followed by a detailed description of the various modules that comprise TGFDMiner. We also provide a list of optimizations that contribute to the efficiency of our discovery algorithm.

5.1 Overview

TGFDMiner, as shown in Algorithm 1 and illustrated in Figure 5.1, is a sequential algorithm that discover minimal and frequent TGFDs. Given a temporal graph $\mathcal{G}_T$, the Statistics module in TGFDMiner identifies a set of vertices, edges, and attributes in $\mathcal{G}_T$ that are ranked in decreasing order of frequency. In the spirit of GFD discovery[12], TGFDMiner also interleaves pattern generation and dependency generation. Pattern

generation is performed by the PattGen module. PattGen uses the set of frequent vertices and edges from the Statistics module to generate candidate graph patterns with up to k edges.

For each candidate graph pattern $Q[\bar{x}]$, the pattern matching module, PattMatch, find matches of $Q[\bar{x}]$ across temporal graph $\mathcal{G}_T$. If $Q[\bar{x}]$ is determined to be frequent in $\mathcal{G}_T$, *i.e.*, $\mathsf{supp}(Q, \mathcal{G}_T) \geq \theta$, then $Q[\bar{x}]$ is considered for dependency generation. Dependency generation is performed by the DependGen module. Given a pattern $Q[\bar{x}]$, DependGen retrieves from the Statistics module a set of attributes that are (a) associated with vertex labels in $Q[\bar{x}]$ and also (b) either frequent or of user interest, in order to generate candidate dependencies of the form $X \rightarrow l$ for $Q[\bar{x}]$.

Finally, for each candidate dependency, the delta discovery module, DeltaDisc, finds an interval of time Δ during which $X \rightarrow l$ holds over occurrences of $Q[\bar{x}]$ and defines a candidate TGFD $\sigma = (Q[\bar{x}], \Delta, X \rightarrow Y)$. If σ is determined to be frequent, *i.e.* $\mathsf{supp}(\sigma, \mathcal{G}_T) \geq \theta$, and minimal, then we add TGFD σ to a set Σ of minimal and frequent TGFDs.

Algorithm 1: TGFDMiner($\mathcal{G}_T$, k, θ)

```
1 Σ := ∅; T = ∅;
2 R, Γ := Statistics(G_T); /* Frequent vertices, edges, and attributes */
3 Σ := PattGen(T);
4 return Σ
```

5.2 Candidate Forest

TGFDMiner uses an auxiliary data structure to track candidate patterns and their

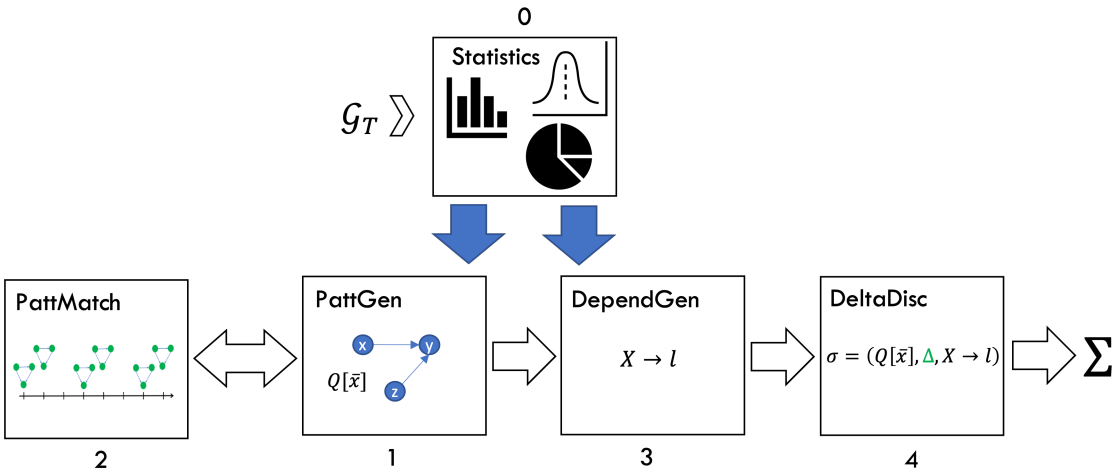


Figure 5.1: TGFDMiner architecture.

respective dependencies. TGFDMiner maintains a forest $\mathcal{T}$ of TGFD candidate trees. Each candidate tree $\mathcal{T}(Q) = (V_{\mathcal{T}}, E_{\mathcal{T}})$ has the following properties:

(a) A set of candidate nodes $V_{\mathcal{T}}$, where each node $v \in V_{\mathcal{T}}$ at level i of tree $\mathcal{T}$, is a pair (Q', d), where $Q[\bar{x}]$ has i edges, and d is a forest of attribute trees.

(b) A set of edges $E_{\mathcal{T}}$, where each edge $e \in E_{\mathcal{T}}$ connects a parent node $v = (Q'', d)$ with a child node $v' = (Q', d')$ such that $Q'' \subset Q'$ and $\tilde{v}_{Q''} = \tilde{v}_{Q'}$.

(c) The root of tree $\mathcal{T}$ is a pair $(Q, \emptyset)$, where Q consists of one vertex and zero edges.

(d) A tree $\mathcal{T}$ rooted at pattern Q is denoted by $\mathcal{T}(Q)$.

(e) The height of $\mathcal{T}$ is at most $k + 1$, *i.e.* $\mathcal{T}$ has at most $k + 1$ levels.

Example 7: Figure 5.2 illustrates a forest $\mathcal{T}$ which consists of two candidate trees. One tree $\mathcal{T}(Q_0)$ is rooted at graph pattern Q_0, where Q_0 consists of single vertex x

with label $L_{Q_0}(x) = \texttt{patient}$. □

Example 8: Figure 5.3 illustrates a candidate node (Q_3, d), where Q_3 is a graph pattern that consists of i edges, and d is represented by a forest of attribute trees generated for Q_3. □

5.3 Pattern Generation

Algorithm 2: PattGen(i)

```
1  Init(T, R_V); /* Initialize T with frequent single-vertex patterns */
2  Σ := ∅; i := 1;
3  while i ≤ k do
4  |  Σ_i := ∅;
5  |  for (Q', d) ∈ T_{i-1} do
6  |  |  for e ∈ R_E ∧ e ∉ E_{Q'} do
7  |  |  |  Q := Q' ∪ e;
8  |  |  |  if ¬iso(Q, T) ∧ ¬HasPrunedSubgraph(Q, T) then
9  |  |  |  |  T_i += Q;
10 |  |  |  M_Q := PattMatch(Q);
11 |  |  |  if supp(Q, G_T) ≤ θ ∧ i ≥ 2 then
12 |  |  |  |  Σ_i := Σ_i ∪ DependGen(M_Q)
13 |  Σ := Σ ∪ Σ_i; i += 1;
14 return Σ
```

Algorithm 3: HasPrunedSubgraph($Q, \mathcal{T}$)

```
1  for Q' ∈ T do
2  |  if supp(Q', G_T) ≥ θ ∧ Q' ⊂ Q then
3  |  |  return true
4  return false
```

Pattern generation in TGFDMiner is performed by the PattGen module (see Algorithm 2). PattGen uses the most frequent vertices and edges to generate general graph

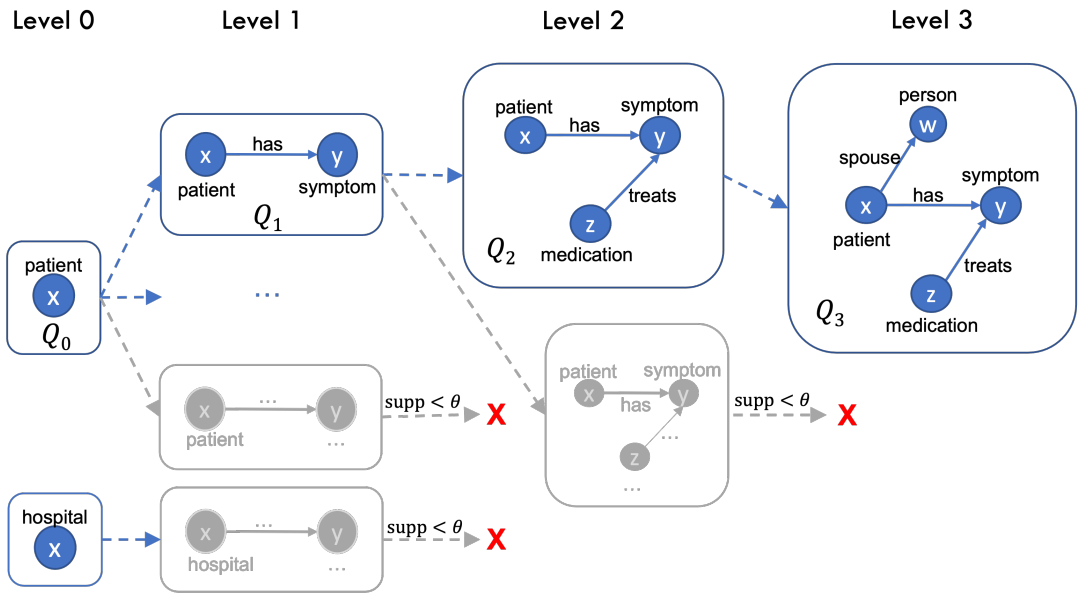


Figure 5.2: Pattern Generation (PattGen).

patterns with i edges, where $0 \leq i \leq k$. PattGen also calculates the support of each pattern and prunes patterns that are not frequent. PattGen stores each generated pattern in a node in a candidate tree.

PattGen initializes a tree $\mathcal{T}(Q_0)$ with root node $(Q_0, \emptyset)$, by generating a single-vertex graph pattern Q_0. All single vertex patterns are created using the set R_V of frequent vertices discovered by the Statistics module.

At each level i of a candidate tree, where $0 < i \leq k$, PattGen expands a tree $\mathcal{T}_i(Q_0)$ by creating a new candidate node $v' = (Q', d')$, such that:

(i) Q' extends existing pattern Q by adding of a frequent edge $e \in R_E$ to Q.

(ii) Q' is not isomorphic to any existing pattern Q in forest $\mathcal{T}$, *i.e.* $\neg\mathsf{iso}(Q, \mathcal{T})$.

(iii) There does not exist a pruned pattern Q in forest $\mathcal{T}$ such that $Q \subset Q'$ (described

in Algorithm 3, reference in Algorithm 2 line 8, see its use in Section 5.7.3).

(iv) Q is frequent in $\mathcal{G}_T$, *i.e.* $\mathsf{supp}(Q,\mathcal{G}_T) \geq \theta$.

Example 9: In Figure 5.2, PattGen creates a new candidate node for pattern Q_3, such that Q_3 adds an edge $e = (x, w)$ to a pattern Q_2 in existing candidate node (Q_2, d). Although this example only depicts patterns that are directed acyclic graphs, our algorithm has no such limitation and is capable of generating general graph patterns. □

For each candidate node (Q', d'), PattGen keeps track of the following:

(a) All candidate nodes (Q, d) where $Q \subset Q'$ (see its use in Section 5.7.4).

(b) A candidate node (Q, d), where $Q \subset Q'$, and $\tilde{v}_{Q'} = \tilde{v}_Q$ (see its use in Section 5.7.1).

In summary, after successfully generating a pattern $Q[\bar{x}]$ with up to k edges, PattGen calls PattMatch to find a set $\mathcal{M}_T$ of all matches of $Q[\bar{x}]$ in temporal graph $\mathcal{G}_T$. PattGen uses $\mathcal{M}_T$ to calculate the support of $Q[\bar{x}]$ in $\mathcal{G}_T$, denoted by $\mathsf{supp}(Q[\bar{x}], \mathcal{G}_T)$. If $Q[\bar{x}]$ is frequent, *i.e.* $\mathsf{supp}(Q[\bar{x}], \mathcal{G}_T) \geq \theta$, PattGen passes $Q[\bar{x}]$ to the dependency generation module.

5.4 Pattern Matching

Pattern matching is performed by the PattMatch module. Given a graph pattern $Q[\bar{x}]$ from PattGen, PattMatch finds all matches of $Q[\bar{x}]$ in temporal graph $\mathcal{G}_T$. PattGen

outputs a set $\mathcal{M}_T$ of all matches of $Q[\bar{x}]$ in $\mathcal{G}_T$. We define $\mathcal{M}_T = \{M_1, ..., M_T\}$ as a sequence of set of matches $\mathcal{M}_t$ with $t \in [1, T]$, such that each set $\mathcal{M}_t$ is a set of matches of $Q[\bar{x}]$ in G_t.

PattMatch performs localised subgraph isomorphism by searching for pattern matches within a r_Q-neighbourhood of every instance of interest vertex $\tilde{v}$ in $\mathcal{G}_T$. We define r_Q to be the longest shortest path from $\tilde{v}$ to any vertex $v \in Q$. PattMatch performs matching in two different modes:

(i) $\mathsf{High-memory\ mode}$ loads all T snapshots of $\mathcal{G}_T$ into memory and achieves a faster runtime in pattern matching. However, high-memory mode is only applicable for a small T.

(ii) $\mathsf{Low-memory\ mode}$, in contrast to high-memory mode, only loads the first snapshot G_1 into memory and applies changes to this snapshot in order to generate all subsequent snapshot G_t, where $1 < t \leq T$. Instead of applying all necessary changes to create subsequent snapshots, this approach only applies changes that are relevant to a given pattern $Q[\bar{x}]$. As expected, low-memory mode uses less memory during execution when compared to high-memory mode. However, low-memory mode has a slower runtime due to the overhead of applying changes to each snapshot G_t.

In summary, PattMatch outputs a set $\mathcal{M}_T$ of matches of a pattern $Q[\bar{x}]$ in temporal graph $\mathcal{G}_T$. PattMatch also includes various modes of matching to allow TGFDMiner to run in both high and low memory environments.

5.5 Dependency Generation

Algorithm 4: DependGen($Q[\bar{x}]$)

```
1 d := ∅; Σ_Q = ∅; n := |Γ| − 1;
2 for x.A ∈ Γ do
3 |   for A ∈ (Γ−{x.A} choose n) do
4 |   |   if IsMinimal(Q, A, {x.A}) then
5 |   |   |   d(x.A) := d(x.A) ∪ A;
6 |   |   |   Σ_Q := Σ_Q ∪ DeltaDisc(X → l, M_Q);
7 return Σ_Q
```

Algorithm 5: IsMinimal($Q, \mathcal{A}, \{x.A\}$)

```
1 for σ' = (Q', X' → l', Δ) ∈ Σ do
2 |   if Q' ⊆ Q ∧ x.A ∈ l then
3 |   |   isMinimal := false;
4 |   |   for y.B ∈ X' do
5 |   |   |   if y.B ∉ A then
6 |   |   |   |   isMinimal := true;
7 |   |   if ¬isMinimal then
8 |   |   |   return false;
9 return true;
```

Dependency generation is performed by the DependGen module (see Algorithm 4). The Statistics module records a set Γ of attributes that are either frequent or of user interest, and are associated with vertex labels in $Q[\bar{x}]$. Using set Γ from Statistics, DependGen generates all possible sets of attributes for candidate dependencies over $Q[\bar{x}]$. DependGen is a deterministic algorithm because it considers all possible set of attributes.

To generate candidate dependencies, DependGen maintains a forest $d = (V_d, E_d)$ of attribute trees. Each attribute tree is generated using attributes from set Γ. An attribute tree $d = (V_d, E_d)$ has the following characteristics:

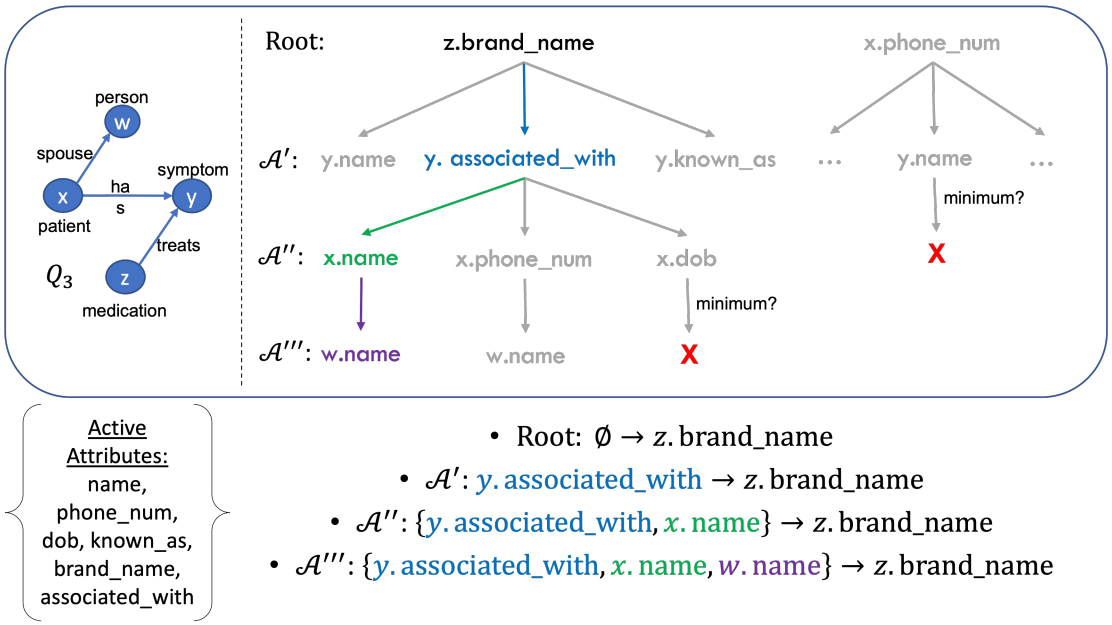


Figure 5.3: Dependency Generation (DependGen).

(a) A tree $d(x.A)$ is rooted at attribute $x.A$, where $A \in \Gamma$ and $x \in \bar{x}$. We simply refer to the root node as $x.A$.

(b) A node $v \in V_d$ stores an attribute $y.B$, where $B \in \Gamma$ and $y \in \bar{x}$. We simply refer to non-root nodes as $y.B$.

(c) An edge $e \in E_d$, where $e = (v, v')$, such that node v stores attribute $y.B$, and node v' stores attribute $z.C$, *i.e.* $v = y.B$ and $v' = z.C$.

(d) A path from the root node $x.A$ to a leaf node $y.B$ has a height of $|\bar{x}|$ because each node v in the path maps to exactly one variable $x \in \bar{x}$.

(e) A path from a leaf node $y.B$ to the root node $x.A$ represents a set of attributes $\mathcal{D}$, where each attribute in $\mathcal{D}$ is of the form $y.B$. The set $\mathcal{D}$ can be further divided into two sets $\{x.A\}$ and $\mathcal{A}$, where $\mathcal{A} = \mathcal{D} - \{x.A\}$.

(f) Sets $\mathcal{A}$ and $\{x.A\}$ can be used to construct a candidate dependency $X \rightarrow l$, such that every attribute $y.B$ in X maps to exactly one attribute $y.B \in \mathcal{A}$, and attribute $x.A$ in l maps to the lone attribute $x.A$ in set $\{x.A\}$.

(g) Every path from a leaf node $y.B$ to the root node $x.A$ represents a minimal dependency $X \rightarrow l$ over $Q[\bar{x}]$, such that there does not exist a general TGFD $\sigma' = (Q', \Delta, X' \rightarrow l) \in \Sigma$ such that $Q' \subset Q$, every attribute $y.B$ in X is also in $\mathcal{A}$, and l is a literal with $x.A$ (see Algorithm 5, called from line 4 in Algorithm 4).

Example 10: Figure 5.3 shows an attribute tree $d(z.\texttt{brand\_name})$ rooted at attribute $z.\texttt{brand\_name}$. Figure 5.3 also shows a highlighted path that begins with root node $z.\texttt{brand\_name}$ and ends with leaf node $w.\texttt{name}$. This path can be used to define sets $\mathcal{A}''' = \{y.\texttt{associated\_with}, x.\texttt{name}, w.\texttt{name}\}$ and $\{z.\texttt{brand\_name}\}$. Sets $\mathcal{A}'''$ and $\{z.\texttt{brand\_name}\}$ can in turn be used to construct candidate dependencies of the form $X \rightarrow l$, where:

(a) X and l consist of constant literals, *i.e.* $X = \{y.\texttt{associated\_with} = c_1, x.\texttt{name} = c_2, w.\texttt{name}= c_3\}$ and $l = \{z.\texttt{brand\_name} = c_3\}$; and

(b) X and l consist of variable literals, *i.e.* $X = \{y.\texttt{associated\_with} = y'.\texttt{associated\_with}, x.\texttt{name} = x'.\texttt{name}, w.\texttt{name} = w'.\texttt{name}\}$ and $l = \{z.\texttt{brand\_name} = z'.\texttt{brand\_name}\}$.

□

In summary, DependGen generates attributes for a minimal dependency $X \rightarrow l$ over frequent pattern $Q[\bar{x}]$. After successfully generating attributes for a minimal

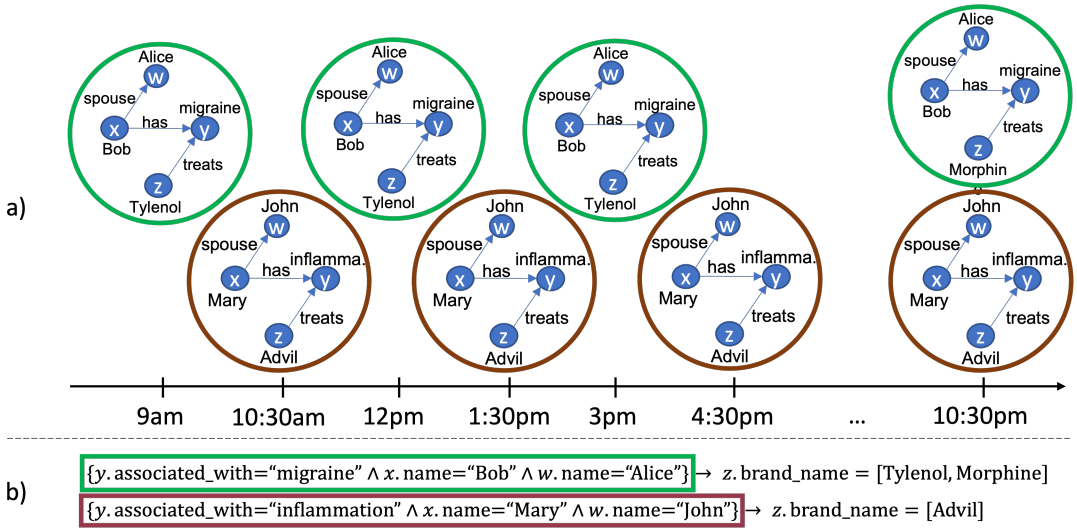


Figure 5.4: Delta Discovery (DeltaDisc).

dependency $X \rightarrow l$, we perform delta discovery to find a time interval Δ where $X \rightarrow l$ holds over the matches of $Q[\bar{x}]$.

5.6 Delta Discovery

Algorithm 6: DeltaDisc$(X \rightarrow l, \mathcal{M}_Q)$

```
1  S := FindEntities(A, {x.A}, M_Q);
2  Σ_S := ∅; Σ_C := ∅;
3  for (h_t(A), s) ∈ S do
4  |   σ_cst := DiscoverConstantTGFDs(h_t(A), s);
5  |   Σ_S := Σ_S ∪ σ_cst;
6  |   if supp(σ_cst, G_T) ≥ θ ∧ IsMinimal(σ_cst) then
7  |   |   Σ_C := Σ_C ∪ σ_cst;
8  Σ_G := DiscoverGeneralTGFDs(Σ_S);
9  Σ_{X→l} := Σ_C ∪ Σ_G;
10 return Σ_{X→l}
```

Delta discovery is performed by the DeltaDisc module in TGFDMiner. DeltaDisc

receives attribute sets $\mathcal{A}$ and $\{x.A\}$ from DependGen to define constant and variable literals for sets X and l, respectively, for candidate dependencies of the form $X \rightarrow l$. For a candidate dependency $X \rightarrow l$ defined over pattern $Q[\bar{x}]$, DeltaDisc attempts to discover a time interval Δ where $X \rightarrow l$ holds over the matches of $Q[\bar{x}]$.

The DeltaDisc module consists of the following submodules:

(i) Entity discovery (FindEntities)

(ii) Constant TGFD discovery (DiscoverConstantTGFDs)

(iii) General TGFD discovery (DiscoverGeneralTGFDs)

5.6.1 Entity Discovery

Algorithm 7: FindEntities($\mathcal{A}, \{x.A\}, \mathcal{M}_Q$)

```
1 S := ∅;
2 for h_t(x̄) ∈ M_Q do
3 |   if A ∈ h_t(x̄) ∧ {x.A} ∈ h_t(x̄) then
4 |   |   s_c(t) += 1;
5 |   |   S := S ∪ (h_t(A), s);
6 return S
```

Entity discovery is performed by the FindEntities submodule (see Algorithm 7) in DeltaDisc. Entity discovery search matches of pattern $Q[\bar{x}]$ to find unique sets of constant values for attributes in $\mathcal{A}$. We refer to a unique set of constant values for $\mathcal{A}$ as an entity instance $h_t(\mathcal{A})$. For every $h_t(\mathcal{A})$, we track the value of $x.A$ over time. Constant values discovered for attributes in $\mathcal{A}$ and $\{x.A\}$ by FindEntities are used by the DiscoverConstantTGFDs module to define constant literals.

Given a set $\mathcal{M}_T$ containing matches of $Q[\bar{x}]$ in $\mathcal{G}_T$, FindEntities identifies all entity instances in $\mathcal{M}_T$. We define an entity instance as a unique match of $Q[\bar{x}]$ in $\mathcal{M}_T$ that contains all attributes in $\mathcal{A}$. We denote an entity instance as $h_t(\mathcal{A})$. FindEntities also search the unique matches of $\mathcal{M}_T$ for all possible constant values associated with attribute $x.A$. For each entity instance, we use a vector s to record every constant value c associated with $x.A$ in $\mathcal{M}_T$. Each entry $s(c)$ stores a vector of size T, such that each entry $s_t(c)$ records the number of matches of $x.A = c$ at timestamp t in $\mathcal{M}_T$. An entity instance $h_t(\mathcal{A})$ and its respective list s of constant values associated with $x.A$ are stored as a pair $(h_t(\mathcal{A}), s)$, and the set of all such pairs is denoted as S. The DiscoverConstantTGFDs uses set S to define constant literals of the form $X \rightarrow l$ for candidate TGFDs.

Example 11: For example, in Figure 5.4a, we see matches of pattern Q_3 across $\mathcal{G}_T$. In Figure 5.4b, we see examples of entity instances, *i.e.* unique matches in $\mathcal{M}_T$ that contain all attributes in $\mathcal{A}$, and for each entity instance, we also see its associated list of matched values for attribute $x.A$. We can define a set S, where the first pair $(h_t(\mathcal{A}'''))_1, s)$ consists of entity instance $h_t(\mathcal{A}'''))_1 = \{y.\texttt{associated\_with} =$ "`migraine`" $\wedge x.\texttt{name} =$ "`Bob`" $\wedge w.\texttt{name} =$ "`Alice`"$\}$ and vector s, which consist of two vectors $s($"`Tylenol`"$) = [1, 0, 1, 0, 1, 0, 0, 0, 0]$ and $s($"`Morphine`"$) = [0, 0, 0, 0, 0, 0, 0, 0, 1]$.

□

In summary, FindEntities computes a set S of pairs $(h_t(\mathcal{A}), s)$, where $h_t(\mathcal{A})$ is a unique match of $Q[\bar{x}]$ that contains all attributes in $\mathcal{A}$, and s is a vector that tracks all possible values of $x.A$ over time. DiscoverConstantTGFDs uses S to define constant literals and build dependencies for candidate TGFDs.

5.6.2 Constant TGFD discovery

Algorithm 8: DiscoverConstantTGFDs($h_t(\mathcal{A}), s$)

```
1 l := max(s);
2 if ∃Δ s.t. (h_{t_i}(x̄), h_{t_j}(x̄)) ⊨ X ∧ (h_{t_i}(x̄), h_{t_j}(x̄)) ⊨ l then
3 |   σ_cst := (Q, X → l, Δ); /* Candidate constant TGFD */
4 |   return σ_cst ;
```

Algorithm 9: IsMinimal(σ_{cst})

```
1 for σ' = (Q', X' → l', Δ) ∈ Σ do
2 |   if Q' ⊆ Q_cst ∧ X' ⊆ X_cst ∧ l' ∈ l_cst then
3 |   |   return false;
4 return true;
```

The discovery of constant TGFDs, *i.e.* TGFDs with dependencies that consist of only constant literals, is performed by the the DiscoverConstantTGFDs submodule (see Algorithm 8) in DeltaDisc. DiscoverConstantTGFDs attempts to define a TGFD for every entity instance $h_t(\mathcal{A})$ discovered by FindEntities.

Given a set S, DiscoverConstantTGFDs attempts to discover a time interval Δ for graph pattern $Q[\bar{x}]$ and a dependency $X \rightarrow l$ which consists of only constant literals, such that $X = h_t(\mathcal{A})$, l is $x.A = c$, and c is the most frequent constant value in s.

For each pair $(h_t(\mathcal{A}), s) \in S$, DiscoverConstantTGFDs finds a constant value c in s, such that the sum of all values in $s(c)$ is greater than the sum of all values in any other vector $s(c')$ (see line 1 in Algorithm 8). Next, for the dependency $X \rightarrow l$ with $X = h_t(\mathcal{A})$, l is $x.A = c$, DiscoverConstantTGFDs find the largest time interval Δ where all pairs $(h_i(\bar{x}),\ h_j(\bar{x}))$ of matches in $\mathcal{M}_T$ satisfy $X \rightarrow l$, *i.e.* $(h_i(\bar{x}),\ h_j(\bar{x}))$ $\models X$ and $(h_{t_i}(\bar{x}),\ h_{t_j}(\bar{x})) \models l$ when $|i - j| \in \Delta$. If such a Δ exists, we define a candidate constant TGFD $\sigma_{\text{cst}} = (Q, X \rightarrow l, \Delta)$.

Given a candidate constant TGFD σ_{cst}, we consider σ_{cst} to be frequent if $\mathsf{supp}(\sigma_{\text{cst}}, \mathcal{G}_T) \geq \theta$. If there does not exist a TGFD $\sigma' \in \Sigma$ such that $\sigma' \preceq \sigma_{cst}$, we consider σ_{cst} to be minimal. If σ_{cst} is minimal and frequent (see line 5 in Algorithm 6), we add σ_{cst} to the set Σ of minimal and frequent TGFDs.

Example 12: In Figure 5.4b, given the entity instance $h_t(\mathcal{A}''')_1 = \{y.\texttt{associated\_with} = \text{“}\texttt{migraine}\text{”}, x.\texttt{name} = \text{“}\texttt{Bob}\text{”}, w.\texttt{name} = \text{“}\texttt{Alice}\text{”}\}$, and the most frequent vector $s(\text{“}\texttt{Tylenol}\text{”}) = [1, 0, 1, 0, 1, 0, 0, 0, 0]$, we define a candidate TGFD $\sigma_1 = (Q_3, \Delta, X_1 \rightarrow l_1)$ which consists entirely of constant literals, such that $X_1 = \{y.\texttt{associated\_with} = \text{“}\texttt{migraine}\text{”} \wedge x.\texttt{name} = \text{“}\texttt{Bob}\text{”} \wedge\ w.\texttt{name} = \text{“}\texttt{Alice}\text{”}\}$ and $l_1 = \{z.\texttt{brand\_name} = \text{“}\texttt{Tylenol}\}\text{”}$. To enforce dependency $X_1 \rightarrow l_1$ over the set of matches in In Figure 5.4a, we can compute an interval $\Delta_1 = (3, 6$ hours), which excludes any pairs of matches of Q_3 where $z.\texttt{brand\_name} = \text{“}\texttt{Morphine}\text{”}$. For the constant TGFD $\sigma_1 := (Q, X_1 \rightarrow l_1, \Delta_1)$, we denote the support as $\mathsf{supp}(\sigma_1, \mathcal{G}_T) = \frac{3}{2 \cdot \binom{9}{2}} = \frac{3}{18}$.

Similarly, for pair $(h_t(\mathcal{A}''')_2, s)$ with the entity instance $h_t(\mathcal{A}''')_2 = \{y.\texttt{associated\_with} = \text{“}\texttt{inflammation}\text{”}, x.\texttt{name} = \text{“}\texttt{Mary}\text{”}, w.\texttt{name} = \text{“}\texttt{John}\text{”}\}$ and single vector $s(\text{“}\texttt{Advil}\text{”})$, we can define another constant TGFD $\sigma_2 := (Q, X_2 \rightarrow l_2, \Delta_2)$, where $X_2 = \{y.\texttt{associated\_with} = \text{“}\texttt{inflammation}\text{”} \wedge x.\texttt{name} = \text{“}\texttt{Mary}\text{”} \wedge w.\texttt{name} = \text{“}\texttt{John}\text{”}\}$ and $l_2 = \{z.\texttt{brand\_name} = \text{“}\texttt{Advil}\text{”}\}$, $\Delta_2 = (3, 12$ hours), and $\mathsf{supp}(\sigma_2, \mathcal{G}_T) = \frac{6}{2 \cdot \binom{9}{2}} = \frac{6}{18}$. □

In summary, for each entity instance discovered by FindEntities, we discover a candidate constant TGFD σ_{cst} by defining a dependency $X \rightarrow l$ consisting of constant literals over graph pattern $Q[\bar{x}]$, finding a time interval Δ where $X \rightarrow l$ holds, and adding σ_{cst} to Σ if frequent and minimal. In the next section, we describe how the set Σ_S of all candidate constant TGFDs can be used to discover a broader type of TGFDs, *i.e.* general TGFDs.

5.6.3 General TGFD discovery

Algorithm 10: DiscoverGeneralTGFDs(Σ_S)

```
1 Σ_g := ∅;
2 Δ_G = intersections(Σ_S)
3 for Δ_g ∈ Δ_G do
4 |   Candidate general TGFD σ_g := (Q, X_g → l_g, Δ_g);
5 |   if supp(σ_g, G_T) ≥ θ then
6 |   |   Σ_g += σ_g;
7 return Σ_g
```

The discovery of general TGFDs, *i.e.* TGFDs with dependencies that consist of only variable literals, is performed by the DiscoverGeneralTGFDs submodule (see Algorithm 10) in DeltaDisc. DiscoverGeneralTGFDs discover TGFDs for the entity type defined by attribute sets $\mathcal{A}$ and $\{x.A\}$. DiscoverGeneralTGFDs uses the attribute sets $\mathcal{A}$ and $\{x.A\}$ to define a dependency $X_g \rightarrow l_g$ which consists of only variable literals, *i.e.* literals of the form $x.A = x'.A$.

Given a set Σ_S of all candidate constant TGFDs, we define a set Δ_S of time intervals, which consists every time interval Δ found in Σ_S. Using set Δ_S of time intervals, we compute a set Δ_G of non-overlapping time intervals, such that each time interval $\Delta_g \in \Delta_G$ intersects with a maximum number of time intervals in Δ_S (see line 2 in Algorithm 10). For each $\Delta_g \in \Delta_G$, DiscoverGeneralTGFDs defines a candidate general TGFD $\sigma_g = (Q[\bar{x}], \Delta_g, X_g \rightarrow l_g)$. If a candidate general TGFD σ_g is determined to be frequent, *i.e.* $\mathsf{supp}(\sigma_g, \mathcal{G}_T) \geq \theta$, we add σ_g to the set Σ of minimal and frequent TGFDs.

Example 13: Given two constant TGFDs σ_1 and σ_2 from Example 12 with intervals of Δ_1 and Δ_2, respectively, we can define a general TGFD $\sigma_g := (Q, X_g \rightarrow l_g, \Delta_g)$,

such that $X_g = \{y_i.\texttt{associated\_with} = y_j.\texttt{associated\_with} \wedge x_i.\texttt{name} = x_j.\texttt{name} \wedge w_.\texttt{name} = w_j.\texttt{name}\}$, $l_g = \{z_i.\texttt{brand\_name} = z_j.\texttt{brand\_name}\}$, $\Delta_g = \Delta_1 \cap \Delta_2 = (3,6) \cap (3,12) = (3,6)$, and $\mathsf{supp}(\sigma_g, \mathcal{G}_T) = \frac{6}{2 \cdot \binom{9}{2}} = \frac{6}{18}$. □

5.7 Optimizations

In this section, we present four optimization that improve the efficiency of TGFDMiner by pruning redundant candidates.

5.7.1 Opt-1: Reuse matches.

This optimization involves storing matches of the interest vertices of each graph pattern Q_i consisting of i edges. To find matches for a larger pattern Q_{i+1}, such that $Q_i \subset Q_{i+1}$, we perform pattern matching in the neighbourhood of all matches of the interest vertices of pattern Q_i. This approach allows us to decrease pattern matching time by reducing the search space in $\mathcal{G}_T$.

During PattGen level $i = 0$, we generate single-vertex patterns Q_0, where the only vertex in Q_0 is also designated as an interest vertex $\tilde{v}_0$. For each pattern Q_0, PattMatch store a list of matched instances of $\tilde{v}_0$ in $\mathcal{G}_T$ in the respective candidate node. At each subsequent level $i > 0$, for each i-sized pattern Q_i, PattGen designates a vertex $v \in V_Q$ as the interest vertex, denoted as $\tilde{v}_{Q_i}$, such that a path r_{Q_i} from $\tilde{v}_{Q_i}$ to any another vertex $v \in V_Q$ is the longest shortest path between all vertices in Q. Given a pattern Q_i, if there exists a pattern Q' with interest vertex $\tilde{v}_{Q'}$, such that $Q' \subset Q_i$ and $\tilde{v}_{Q'} = \tilde{v}_{Q_i}$, we re-use the list of matched instances of $\tilde{v}_{Q'}$ stored in candidate

node of Q' and perform subgraph isomorphism in the r_{Q_i}-neighbourhood of each $\tilde{v}_{Q'}$ instance to find matches of Q_i. We prune all matches of $\tilde{v}_{Q'}$ that do not contain any matches of Q_i and store the pruned list of matches in the candidate node of Q_i. If there does not exist a pattern Q', we locate all instances of $\tilde{v}$ in $\mathcal{G}_T$, perform subgraph isomorphism in the r_{Q_i}-neighbourhood of each $\tilde{v}$ instance to find matches of Q_i, and store all relevant instances of $\tilde{v}$ as a list in the candidate node for Q_i.

5.7.2 Opt-2: Incremental matching.

In a temporal graph $\mathcal{G}_T$, a snapshot G_j can be derived from a previous snapshot G_i, where $i < j$, via a list of changes such as updates, insertions, and deletions to vertices, edges, and attributes [4]. Given a list of changes that occur between consecutive G_i and G_j, the incremental matching approach allows us to reduce pattern matching time by only recomputing matches that are affected by this list of changes. This optimization works best when the number of changes between consecutive snapshots is small. For a small number of changes, only a few matches will be recomputed. However, for a large number of changes, each match might be recomputed several times, which offsets the benefits of this optimization.

If a match $h_i(\bar{x})$ from G_i is updated in G_j, we add an updated match $h_j(\bar{x})$ to $\mathcal{M}_T$. If a new match $h_j(\bar{x})$ is inserted in G_j, we add a new match $h_j(\bar{x})$ to $\mathcal{M}_T$. If a match $h_i(\bar{x})$ from G_i is deleted in G_j, we do not add anything to $\mathcal{M}_T$. If there is no change to $h_i(\bar{x})$ in G_j, we assume the match persists and add an identical match $h_j(\bar{x})$ to $\mathcal{M}_T$, where $h_j(\bar{x}) = h_i(\bar{x})$. For a temporal graph $\mathcal{G}_T$ with a small number of changes between snapshots, this optimization can decrease pattern matching time.

5.7.3 Opt-3: Support pruning.

Based on the anti-monotonic property of TGFD support, during PattGen, we prune any redundant candidate pattern Q_i, if there is an existing pattern Q', such that $Q' \subset Q_i$, $\tilde{v}_{Q'} = \tilde{v}_{Q_i}$, and $\mathsf{supp}(Q', \mathcal{G}_T) < \theta$. This approach allows us to decrease runtime by avoiding pattern matching for patterns with low support.

5.7.4 Opt-4: Minimality pruning.

This optimization decreases runtime by allowing TGFDMiner to prune candidate TGFDs that are not minimal. During DependGen and DeltaDisc, TGFDMiner uses Axioms 1, 2, and 3 to prune a redundant candidate TGFD $\sigma' = (Q', \Delta', X' \rightarrow l')$ if there is an existing TGFD $\sigma \in \Sigma$, such that $\sigma = (Q, \Delta, X \rightarrow l)$, $Q \subseteq Q'$, $X \subseteq X'$, $l = l'$, and $\Delta \subseteq \Delta'$ (see Algorithms 5 and 9). This approach does not cause a loss of TGFDs because our axioms ensure that only non-minimal candidate TGFDs, i.e. TGFDs that can be inferred by minimal TGFDs, are pruned during discovery.

In summary, we have introduced TGFDMiner, its modules, and its optimizations. In the next chapter, we evaluate the efficiency and scalability of TGFDMiner and compare its performance to a baseline to show its effectiveness. We also present examples of real-world TGFDs discovered from IMDB.

Chapter 6

Experiments

In this chapter, we evaluate the performance of TGFDMiner for varying parameters, and compare against an existing baseline. We also evaluate the benefits of our optimizations and present examples of real-world TGFDs.

6.1 Setup

We implemented TGFDMiner using JDK 15 and conducted our experiments on a cluster of four Red Hat Enterprise Linux servers, where three servers use Intel Xeon E5-2670 2.60Ghz with 8 to 12 cores and 128GB RAM, and one server which uses Intel Xeon E7-4870 2.40Ghz CPU with 16 cores and 256GB RAM.

6.2 Datasets

We evaluate TGFDMiner using two real-world graphs DBpedia and IMDB, and one synthetically generated graph.

1. DBpedia [19]: is a real-world knowledge graph that consists of three semi-annual snapshots from 2015 to 2016. Each snapshot contains an roughly 5M vertices with 422 vertex labels (e.g. soccer club, album, country, film) and 14M edges with 45K edge labels (e.g. team, occupation, starring).

2. IMDB [1]: consists of 26 monthly snapshots from October 2014 to December 2017. Each snapshot contains roughly 700K vertices with 6 vertex labels (e.g. actor, actress, director, movie, genre, country) and 1M edges with 5 edge labels (e.g. actor_of, actress_of, director_of, genre_of, country_of_origin).

3. Synthetic: social network, generated using a synthetic graph generator named gMark [6], consists of 21 snapshots, where each snapshot consists of roughly 2.7M vertices with 12 vertex labels (e.g. person, message, comment, city) and 3.5M edges with 22 edge labels (e.g. knows, replyof, islocatedin).

6.3 Results for varying parameters

In this section, we present evaluation results of TGFDMiner for varying parameters. By default, we set $\theta = 0.05$, $|\Gamma| = 20$, $k = 3$, and $T = 3$. For each dataset, by default, we use the graph size $|G| = (|V|, |E|)$ specified in the previous section.

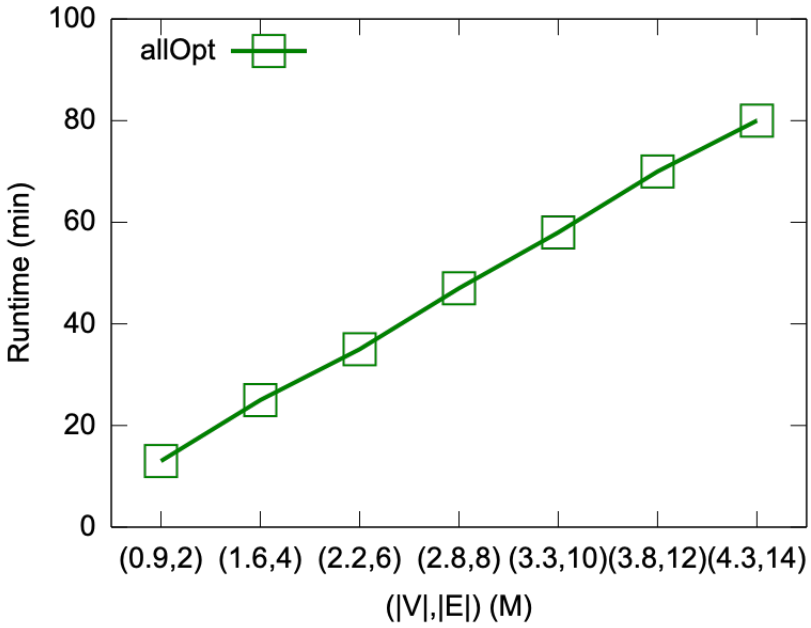


Figure 6.1: Vary $|G|$ (DBpedia).

6.3.1 Varying $|G|$

To evaluate the effect of varying $|G|$, we use a variety of uniformly increasing graph sizes $(|V|, |E|)$ of DBpedia: (0.9,2), (1.6,4), (2.2,6), (2.8,8), (3.3,10), (3.8,12), (4.3,14). In Figure 6.1, TGFDMiner runtime increases as $|G|$ increases, as expected. Our results show that TGFDMiner is scalable because it completes in under 81 minutes for 3 DBpedia snapshots of size $(4.3M, 14M)$.

6.3.2 Varying $|T|$

Since IMDB has many snapshots, we use IMDB to evaluate the effect of varying $|T|$. Figure 6.2 shows TGFDMiner runtime increasing steadily for larger $|T|$ over IMDB. For $|T| > 21$, the runtime increase less steadily because the memory usage of TGFDMiner

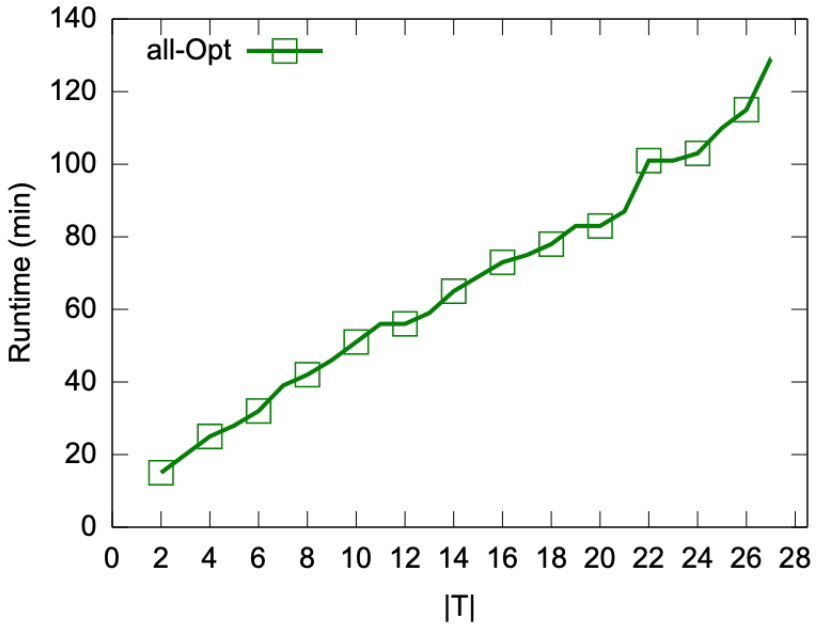


Figure 6.2: Vary $|T|$ (IMDB).

approaches the server's full RAM capacity.

6.3.3 Varying k

DBpedia has many vertex and edge labels which can be used to build large graph patterns. Hence, we use DBpedia to evaluate the effect of varying k. At larger k, larger and more complex pattern candidates are matched and larger dependencies are considered, causing a steep increase in runtime when performing PattMatch and DeltaDisc. For example, in Figure 6.3, this steep increase is most noticeable between $k = 4$ and $k = 5$, where the runtime increases by 70%. At $k = 6$, TGFDMiner was terminated after 38000 minutes.

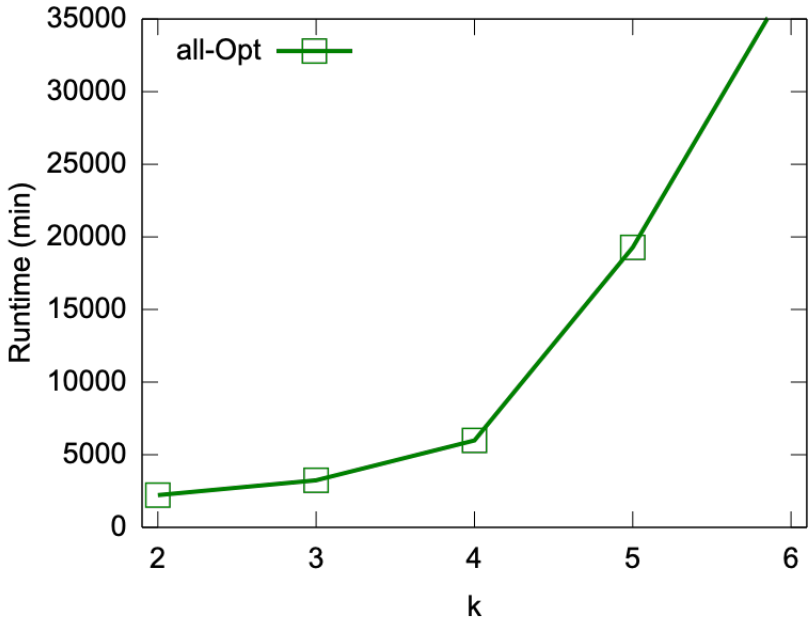


Figure 6.3: Vary k (DBpedia).

6.3.4 Varying θ

We use DBpedia and IMDB to evalulate the effect of varying θ. TGFDMiner runtime decreases as θ increases because at larger θ, more candidate patterns are pruned than at lower θ, which in turn eliminates more candidate dependency and delta discovery. Most patterns discovered by TGFDMiner in DBpedia have a pattern support below 0.1. Accordingly, when we studied the effects of varying θ over DBpedia in Figure 6.4, we noticed a nearly 50% decrease in runtime when θ was increased from 0.03 to 0.09. Similarly, in IMDB, most patterns discovered by TGFDMiner had a pattern support below 0.15, which is evident in the 83% increase in runtime in Figure 6.5 when θ is decreased from 0.15 to 0.12.

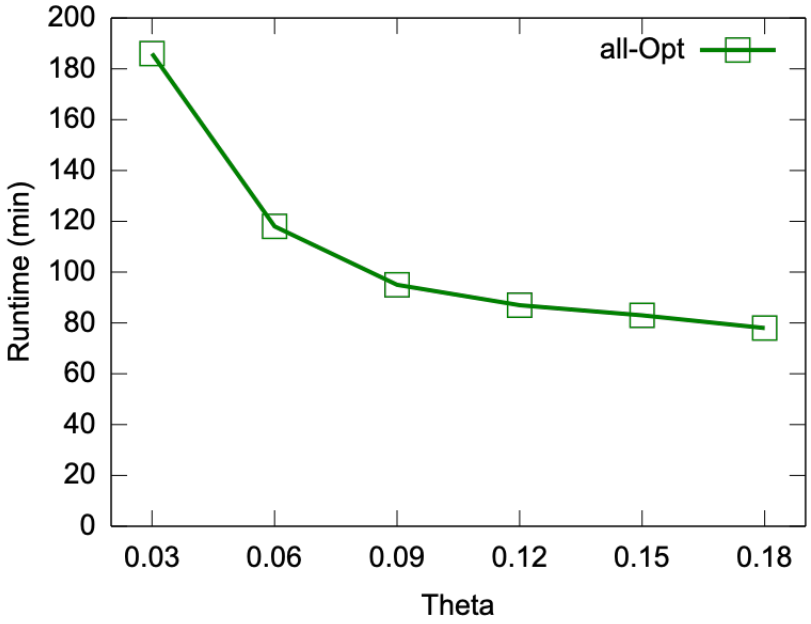


Figure 6.4: Vary θ (DBpedia).

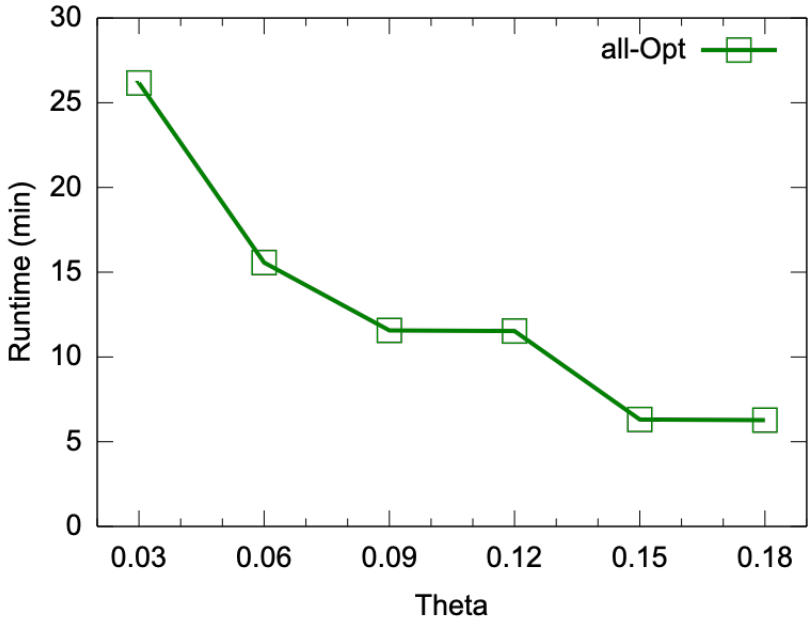


Figure 6.5: Vary θ (IMDB).

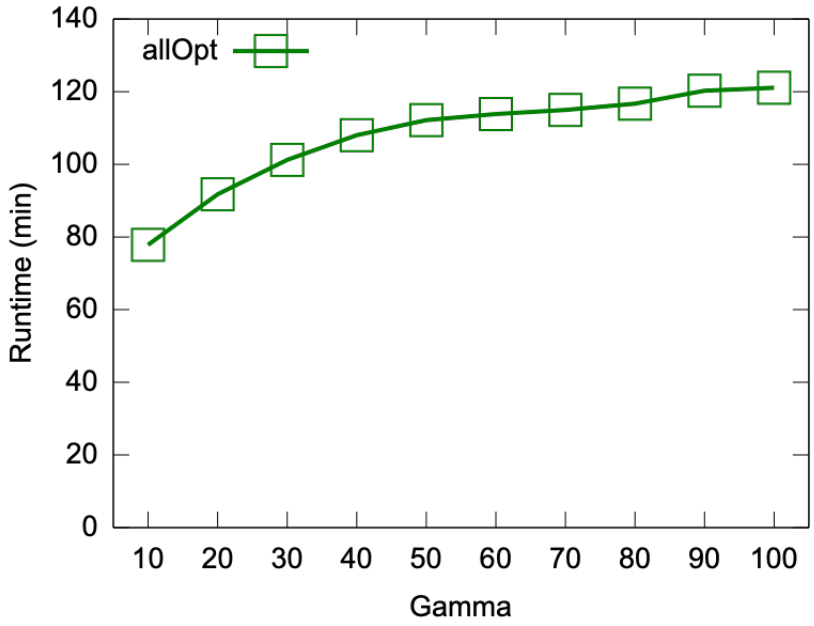


Figure 6.6: Vary $|\Gamma|$ (DBpedia).

6.3.5 Varying $|\Gamma|$

Since DBpedia has a large number of attributes, we use DBpedia to evaluate the effect of varying $|\Gamma|$. Γ contains a set of attributes that are ranked in order of decreasing frequency. In Figure 6.6, we notice TGFDMiner runtime increases steadily for $10 \leq |\Gamma| \leq 40$ because the most frequent attributes in Γ are chosen first. For $|\Gamma| > 50$, there increase in runtime is slower as less frequent attributes in Γ are chosen. Overall, a larger $|\Gamma|$ has a higher runtime because more candidate dependencies are generated during DependGen, which in turn results in more candidate TGFDs being processed during DeltaDisc.

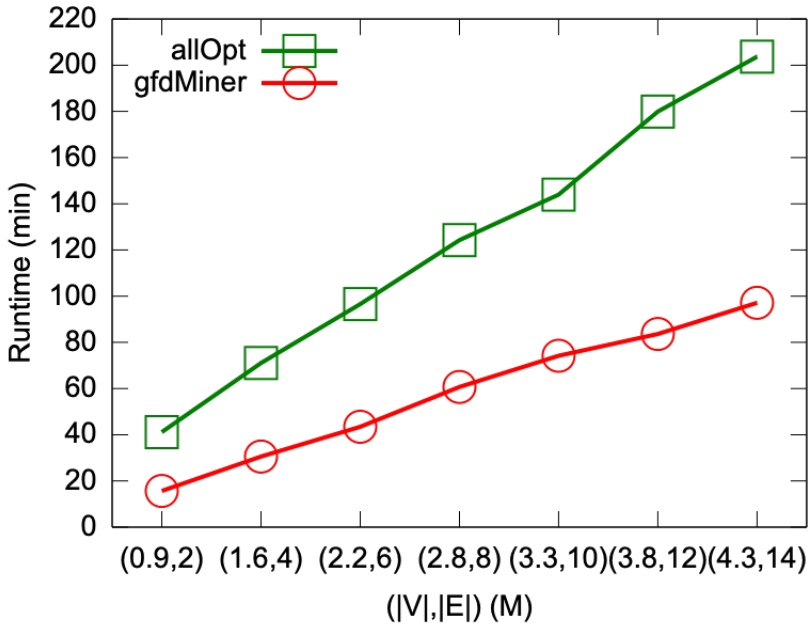


Figure 6.7: Varying $|G|$. GFDMiner vs. TGFDMiner comparison (DBpedia).

6.4 Comparative Performance

In this section, we vary parameters $|G|$ and k and compare the evaluation results of TGFDMiner against results from an existing baseline algorithm. We use GFDMiner as a baseline algorithm.

6.4.1 Comparison with GFDs (Varying $|G|$)

Since GFDMiner only runs on static graphs, we ran GFDMiner separately on each snapshot of DBpedia. In Figure 6.7, for DBpedia graphs of sizes (0.9,2), (1.6,4), (2.2,6), (2.8,8), (3.3,10), (3.8,12), (4.3,14), TGFDMiner is slower than GFDMiner discovery by 133%, 122%, 105%, 94%, 115%, and 110% respectively. On average, TGFDMiner

$(\|V\|, \|E\|)$	avg. # of GFDs discovered per snapshot	# of TGFDs discovered in G_T
(0.9,2)	1040	2059
(1.6,4)	1572	2160
(2.2,6)	1904	2374
(2.8,8)	2219	2611
(3.3,10)	2378	2692
(3.8,12)	2621	2745
(4.3,14)	2755	2559

Table 6.1: Comparing outputs of GFDMiner and TGFDMiner in Experiment 6.4.1

is slower than GFDMiner by 113% due to the additional cost of performing pairwise comparison of matches across snapshots in DeltaDisc. The number of TGFDs and GFDs discovered by TGFDMiner and GFDMiner, respectively, is comparable (see Table 6.1).

6.4.2 Comparison with GFDs (Varying k)

Synthetic has many vertex and edge labels which can be used to build large graph patterns. Hence, we use Synthetic to evaluate the effect of varying k. As seen in Figure 6.8, the performance of TGFDMiner over Synthetic is comparable to GFDMiner when k is varied. However, TGFDMiner is slightly faster than GFDMiner because TGFDMiner only runs DiscoverConstantTGFDs once over all snapshots, whereas GFDMiner needs to discover GFDs for each snapshot separately. The number of TGFDs and GFDs discovered by TGFDMiner and GFDMiner, respectively, is comparable. TGFDMiner discovered 256 TGFDs across all snapshots, whereas GFDMiner discovered an average of 282 GFDs in each snapshot.

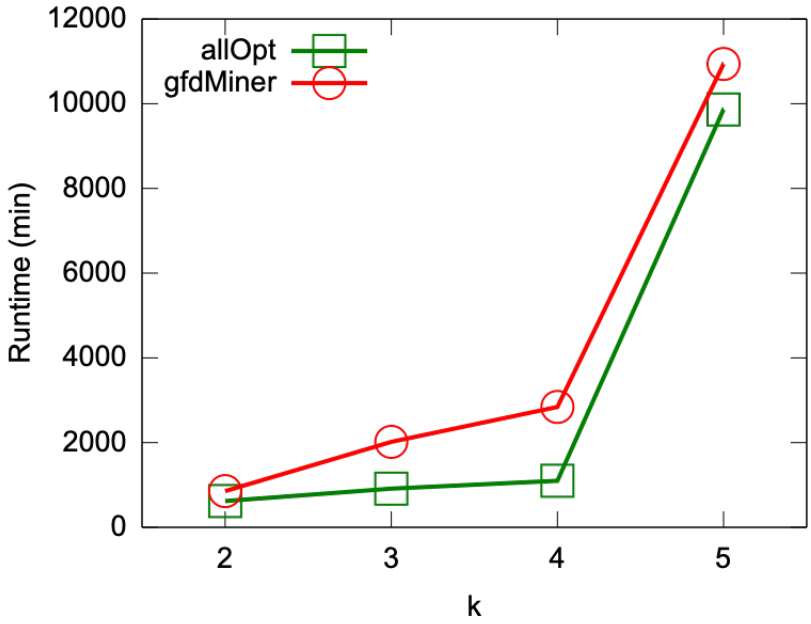


Figure 6.8: Varying k. GFDMiner vs. TGFDMiner comparison (DBpedia).

6.5 Benefits of optimizations

In Figure 6.9, we vary $|G|$ and compare the effect of each of our optimizations against a naive implementation of TGFDMiner. For this experiment, we use the DBpedia dataset, which has a large number of changes between consecutive snapshots. However, since Opt-2 works best when there are a small number of changes between consecutive snapshots, we use $\mathsf{Low-memory\ mode}$ and applying changes to 0.001% of the vertices and edges in the current snapshot to create the next snapshot. We apply insertions, deletions, and updates to vertices, edges, and attributes. In comparison to naive, TGFDMiner, Opt-1, Opt-2, Opt-3, and Opt-4 achieve a a decrease in runtime of 7.24%, 8.26%, 43.00%, and 19.04% respectively. Table 6.2 shows the number of output TGFDs, $|\Sigma|$ for each optimization and the naive implementation.

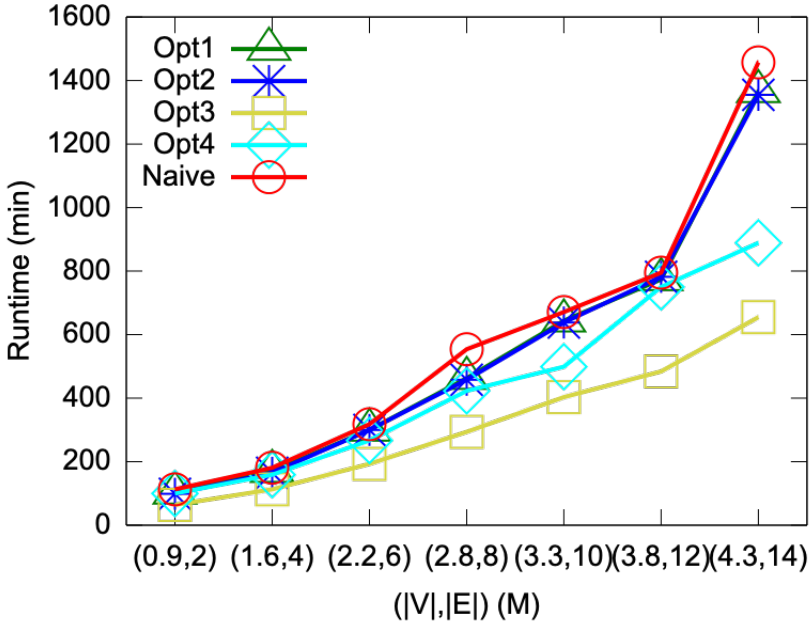


Figure 6.9: Benefits of optimizations (DBpedia).

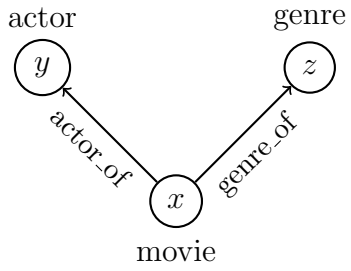
We observe that only Opt-4 causes a decrease in $|\Sigma|$ because only Opt-4 prunes non-minimal TGFDs. Opt-4 achieves the second highest decrease in runtime by pruning non-minimal candidate TGFDs from being processed by DependGen and DeltaDisc. In comparison, Opt-1 and Opt-2 have a slightly lower runtime than the naive implementation because they improve the efficiency of pattern matching. However, Opt-3 achieves the highest decrease in runtime by preventing previously matched low frequency graph patterns from being expanded by PattGen and matched by PattMatch.

6.6 Case Study

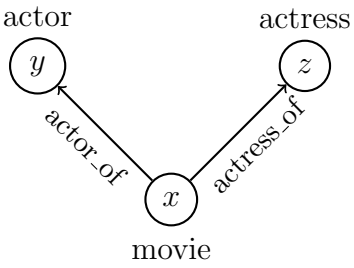
We present the following real-world TGFDs discovered over IMDB, with patterns illustrated in Figure 6.10.

Optimizations	$(\|V\|, \|E\|)$						
	(0.9, 2)	(1.6, 4)	(2.2, 6)	(2.8, 8)	(3.3, 10)	(3.8, 12)	(4.3, 12)
Opt-1	26863	5888	6960	10564	12411	7947	11590
Opt-2	26863	5888	6960	10564	12411	7947	11590
Opt-3	26863	5888	6960	10564	12411	7947	11590
Opt-4	4639	3243	3397	3806	3818	3534	3137
Naive	26863	5888	6960	10564	12411	7947	11590

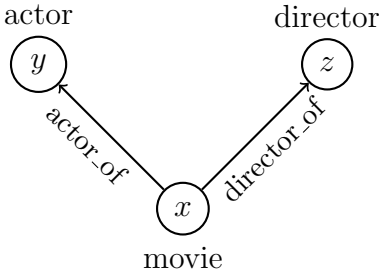
Table 6.2: Comparing $|\Sigma|$ of all optimizations in Experiment 6.5



Q_5



Q_6



Q_7

Figure 6.10: Case Study

TGFD$_1$: $\sigma_5 = (Q_5, (6, 8 \text{ months})\{movie.\text{name} \wedge genre.\text{name}\} \rightarrow actor.\text{name})$ acts as a time-constrained key which specifies that for an interval of 6 to 8 months, a movie's name and genre can uniquely determine its actor.

TGFD$_2$: $\sigma_6 = (Q_5, (9, 9 \text{ months})\{movie.\text{language\_of} \wedge actor.\text{name}\} \rightarrow genre.\text{name})$ specifies that an actor working in a specific movie-language industry acts in movies belonging to the same genre every 9 months.

TGFD$_3$: $\sigma_7 = (Q_6, (6, 6 \text{ months})\{actor.\text{name} \wedge actress.\text{name}\} \rightarrow movie.\text{language\_of})$ specifies that an actor working with a specific actress will act in the same movie-language industry every 6 months.

TGFD$_4$: $\sigma_8 = (Q_7, (6, 6 \text{ months})\{actor.\text{name} \wedge director.\text{name}\} \rightarrow movie.\text{language\_of})$ specifies that an actor working with a specific director will act in the same movie-language industry every 6 months.

In summary, we have evaluated TGFDMiner on large-scale real-world and synthetic graphs to show that TGFDMiner is an efficient and scalable algorithm for discovering TGFDs, and that our optimizations for TGFDMiner are effective. We have also shown that the performance of TGFDMiner is comparable to an existing baseline algorithm. We have also presented various real-world TGFDs from real-world dataset IMDB. In the next chapter, we conclude this thesis and provide directions for future work.

Chapter 7

Conclusion and Future Work

We have formalized the TGFD discovery problem, defined minimality and support for TGFDs, and presented an efficient and effective sequential algorithm, TGFDMiner, that discovers minimal and frequent TGFDs. We have also described various optimizations to improve the runtime efficiency of TGFDMiner by pruning redundant patterns and redundant TGFD candidates. Our evaluations, performed over real and synthetic large graph datasets, show that our optimizations are effective, and TGFDMiner is scalable, efficient, and discovers interesting TGFDs.

7.1 Future work

As future work, we intend to:

1. Discover a wider variety of constant TGFDs. For instance, currently, if there exist two candidate TGFDs $\sigma = (Q, X \rightarrow l.A = c, \Delta)$ and $\sigma' = (Q, X \rightarrow l.A =$

$c', \Delta')$, where $\mathsf{supp}(\sigma, \mathcal{G}_T) > \mathsf{supp}(\sigma', \mathcal{G}_T) > \theta$, TGFDMiner will only add σ to Σ. We plan to allow future iterations of TGFDMiner to discover constant TGFDs σ' that do not have maximum frequency, but still satisfy TGFD support.

2. Explore approximate TGFDs, as opposed to exact TGFDs. Currently, our algorithm discovers exact TGFDs, which are TGFDs that do not allow any violations of the satisfiability definition of TGFDs. In contrast to exact TGFDs, approximate TGFDs allow a small number of violations.

3. Discover richer TGFDs by allowing a dependency $X \rightarrow Y$ to contain both constant and variable literals simultaneously. Currently, for any TGFD with dependency $X \rightarrow Y$ discovered by TGFDMiner, all literals in X and Y must be only constant or only variable.

4. Develop a parallel algorithm for TGFD discovery to reduce pattern matching time and thereby perform discovery on even larger graphs.

5. During our evaluation of TGFDMiner over DBpedia, we noticed many TGFDs were lost over time due to type migration in vertices. For example, in DBpedia, vertices labelled as *settlement* would often be re-labelled as *village* or *city* over time due to population changes. By incorporating semantic knowledge and ontologies into TGFDMiner, we could reduce the loss of valuable TGFDs over time.

Bibliography

[1] IMDB dataset. `ftp://ftp.fu-berlin.de/pub/misc/movies/database/frozendata/`, 2021.

[2] Z. Abedjan, C. G. Akcora, M. Ouzzani, P. Papotti, and M. Stonebraker. Temporal rules discovery for web data cleaning. *PVLDB*, 9(4):336–347, 2015.

[3] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *Acm sigmod record*, volume 22, pages 207–216. ACM, 1993.

[4] M. Alipour Langouri, A. Mansfield, F. Chiang, and Y. Wu. Temporal graph functional dependencies: Technical report. *arXiv preprint arXiv:2108.08719*, 2021.

[5] M. Arenas, L. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 68–79, 1999.

[6] G. Bagan, A. Bonifati, R. Ciucanu, G. H. Fletcher, A. Lemay, and N. Advokaat. Generating flexible workloads for graph databases. *Proceedings of the VLDB Endowment*, 9(13):1457–1460, 2016.

[7] M. Berlingerio, F. Bonchi, B. Bringmann, and A. Gionis. Mining graph evolution rules. In *joint European conference on machine learning and knowledge discovery in databases*, pages 115–130. Springer, 2009.

[8] P. Bohannon, W. Fan, M. Flaster, and R. Rastogi. A cost-based model and effective heuristic for repairing. In *SIGMOD*, pages 143–154, 2005.

[9] F. Chiang and R. J. Miller. A unified model for data and constraint repair. In *ICDE*, pages 446–457, 2011.

[10] W. Fan, Z. Fan, C. Tian, and X. L. Dong. Keys for graphs. *Proceedings of the VLDB Endowment*, 8(12):1590–1601, 2015.

[11] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Transactions on Database Systems (TODS)*, 33(2):1–48, 2008.

[12] W. Fan, C. Hu, X. Liu, and P. Lu. Discovering graph functional dependencies. In *SIGMOD*, page 427–439, 2018.

[13] W. Fan, X. Liu, P. Lu, and C. Tian. Catching numeric inconsistencies in graphs. In *Proceedings of the 2018 International Conference on Management of Data*, pages 381–393, 2018.

[14] W. Fan and P. Lu. Dependencies for graphs. *ACM Transactions on Database Systems (TODS)*, 44(2):1–40, 2019.

[15] W. Fan, X. Wang, Y. Wu, and J. Xu. Association rules with graph patterns. *Proceedings of the VLDB Endowment*, 8(12):1502–1513, 2015.

[16] W. Fan, Y. Wu, and J. Xu. Functional dependencies for graphs. In *SIGMOD International Conference on Management of Data*, pages 1843–1857, 2016.

[17] L. Golab, I. F. Ilyas, G. Beskales, and A. Galiullin. On the relative trust between inconsistent data and inaccurate constraints. In *ICDE*, pages 541–552, 2013.

[18] S. Kolahi and L. V. S. Lakshmanan. On approximating optimum repairs for functional dependency violations. In *ICDT*, pages 53–62, 2009.

[19] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. Van Kleef, S. Auer, et al. Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic web*, 6(2):167–195, 2015.

[20] H. Ma, M. Alipourlangouri, Y. Wu, F. Chiang, and J. Pi. Ontology-based entity matching in attributed graphs. *Proceedings of the VLDB Endowment*, 12(10):1195–1207, 2019.

[21] F. Mahdisoltani, J. Biega, and F. Suchanek. Yago3: A knowledge base from multilingual wikipedias. In *7th biennial conference on innovative data systems research*. CIDR Conference, 2014.

[22] M. H. Namaki, Y. Wu, Q. Song, P. Lin, and T. Ge. Discovering graph temporal association rules. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 1697–1706, 2017.

[23] E. Scharwächter, E. Müller, J. Donges, M. Hassani, and T. Seidl. Detecting change processes in dynamic networks by frequent graph evolution rule mining.

In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 1191–1196, 2016.

[24] D. Vrandečić and M. Krötzsch. Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014.