CMiner: An Algorithm to Discover Frequent Structures in Conceptual Models

Simone Avellino
Università di Catania
Catania, Italy
simone.avellino@studium.unict.it

Emanuele Valore
Università di Catania
Catania, Italy
e.valore@studium.unict.it

Giovanni Micale
Università di Catania
Catania, Italy
giovanni.micale@unict.it

Antonio Di Maria
Università di Catania
Catania, Italy
antonio.dimaria1@unict.it

Mattia Fumagalli
Free University of Bozen-Bolzano
Bozen-Bolzano, Italy
mattia.fumagalli@unibz.it

Tiago Prince Sales
University of Twente
Enschede, The Netherlands
t.princesales@utwente.nl

Alfredo Pulvirenti
Università di Catania
Catania, Italy
apulvirenti@dmi.unict.it

Diego Calvanese
Free University of Bozen-Bolzano
Bozen-Bolzano, Italy
diego.calvanese@unibz.it

Abstract

The analysis of conceptual models to find recurrent structures and patterns is a lively research area, aimed at finding good or bad modeling practices or, more generally, recurrent phenomena represented within the model. In the last few years, several approaches have been explored using automated techniques to support such a discovery process. However, due to the complex structure of the graphs encoding conceptual models and the challenging nature of the discovery task, the available techniques can still be largely enhanced. This paper presents a novel *Frequent Subgraph Mining (FSM)* algorithm called CMiner, designed specifically for discovering recurrent structures in conceptual models, which offers a guided, performance-oriented approach to mining semantically rich graphs. The proposed solution is inspired by established methods but is tailored to address specific needs arising in the conceptual modeling context. It supports heuristic tasks, enhances the discovery of recurring structures, and thus can serve as a key ally in pattern identification. Besides providing a detailed overview of CMiner, which is freely available, we validate it as a useful support tool for pattern discovery and compare it with state-of-the-art solutions.

Keywords

Frequent Subgraph Mining, Conceptual Modeling, Pattern Discovery, Pattern Mining, Conceptual Modeling Patterns

1 Introduction

Conceptual models are abstract representations of systems formed by objects, processes and their relationships and typically take the form of a graph constructed according to the syntax of a specific language. Examples of such conceptual languages include the *Business Process Model and Notation (BPMN)*[1] and the *Unified Modeling Language (UML)*.[2] These models can later be mapped to a logical language with formal semantics.

Conceptual models serve as essential tools for designing, implementing, maintaining, and understanding applications, software, and, more broadly, information systems, helping to untangle complex domain-specific problems. They have a broad applicability in many different domains [2, 6, 22]. In security, conceptual models can be used to highlight financial challenges or security concerns, such as patterns of recurring attacks. In databases, these models can reveal how certain objects (e.g., buildings or books) or processes (e.g., purchasing workflows) are typically represented and are at the basis of widely adopted database design methodologies. In software engineering, conceptual models provide insight into how the modeling languages themselves are applied, e.g., by showing how taxonomic relationships are used in UML or how gates are implemented in BPMN. In biology, conceptual models can serve as a valuable resource for analyzing correlations between molecules, depending on the chosen reference system.

A significant portion of the efforts within the conceptual modeling community focuses on identifying recurring model structures or patterns that correspond to interesting modeling behaviors or common ways of representing certain phenomena. Patterns may help to understand how a modeling language is adopted, identify language dialects for specific application domains, verify possible restrictions in using constructs or their combinations, and determine the frequent *subversions* of the language.[3]

Pattern analysis in conceptual models has seen significant growth in recent years [4, 8, 13, 16], focusing on the identification of reference patterns (used as case studies) to infer modeling strategies for specific problems. Several archives of conceptual patterns have been developed [21], highlighting common modeling practices (both good and bad) and documenting frequently occurring modeling scenarios.

More recent work has explored the application of automated techniques *to support pattern discovery tasks* traditionally performed manually [11, 12, 28, 35]. In this context, *Frequent Subgraph Mining* (FSM) algorithms offer a promising solution to

[1] https://www.bpmn.org/
[2] https://www.uml.org/

[3] The notion of *systematic language subversion* [17] refers to an ungrammatical use of a language's constructs that becomes recurrent in a language community signaling a design limitation of that language. It is closely related to *coding traditions* [9] which cover coding policies, notational guidelines, naming conventions, implementation patterns, programming idioms, etc.

EDBT '26, Tampere (Finland)
© 2026 Copyright held by the owner/author(s). Published on OpenProceedings.org under ISBN 978-3-98318-104-9, series ISSN 2367-2005. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

10.48786/edbt.2026.28

support the automation of pattern discovery.[4] FSM is a well-known technique that consists of finding all frequent subgraphs in a database of graphs, based on a predefined threshold value. FSM typically involves the generation of candidate subgraphs and the calculation of their frequency in the database [24].

However, many FSM algorithms do not scale well with the size of database graphs, are not flexible, and cannot even handle the multiple features of the graphs that encode conceptual models [10]. Indeed, many conceptual models are directed multigraphs, with nodes and relationships labeled in multiple ways. Moreover, in many applications, it is important to assess the frequency of a subgraph not only within individual models, but also across different models. Finally, the recurrent structures discovered in conceptual models resemble those already recognized as important by researchers, suggesting that search algorithms should be guided to some extent in the identification of frequent subgraphs.

To address the aforementioned challenges, we introduce in this paper a new FSM algorithm called CMiner, which uses a *pattern-growth approach* [18] to search for candidate subgraphs. CMiner introduces novel compact data structures to efficiently generate candidate subgraphs for mining, namely the Node Extension Table (NET) and the Edge Extension Table (EET), inspired by the bit matrix concept introduced in our previous work on subgraph matching in multigraphs [29]. CMiner is able to calculate both the support and the frequency of each candidate subgraph within each graph of the database and gives the possibility to guide the search by specifying *templates*, i.e., topological and/or label constraints that subgraphs should follow, making the algorithm more flexible and adaptable to the requirements of the research community working on conceptual models. Experimental results show that CMiner's performance is comparable to state-of-the-art algorithms, such as *gSpan* [38] and *Gaston* [30]. In addition, CMiner is scalable, memory-efficient and accurate in both directed and undirected networks,[5] resulting in a tool suitable for mining semantically rich graphs, such as conceptual models. CMiner is open-source and freely available at `https://github.com/SimoneAvellino/CMiner`.

The rest of the paper is structured as follows. Section 2 defines preliminary concepts regarding multigraphs and frequent subgraph mining. In Section 3, we set the problem requirements based on the feedback received from expert language engineers in conceptual models. In Section 4, we describe the functioning of the CMiner algorithm. In Section 5 we discuss the complexity of the algorithm. In Section 6, we validate CMiner and evaluate its performance. In Section 7, we discuss the experimental results. In Section 8, we review the literature on frequent subgraph mining. Finally, Section 9 concludes the paper.

2 Preliminary Definitions

In this section, we give some preliminary definitions about multigraphs, subgraph matching in multigraphs and frequent subgraph mining.

Firstly, we formalize a graph as follows:

Definition 2.1. A *graph* is a pair $G = (V, E)$, where:

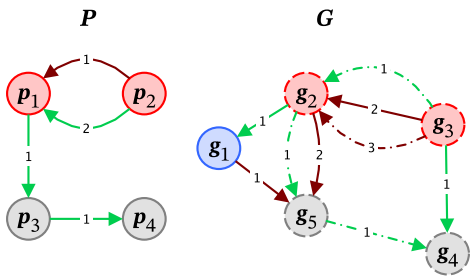


Figure 1: A multigraph P occurring in a multigraph G. Nodes and edges of the occurrence are drawn with dashed lines.

- V is a set of *nodes*,
- E is a set of ordered pairs of nodes called *edges*.

If $\forall (u, v) \in E : (v, u) \in E$, G is an *undirected graph*, otherwise G is a *directed graph* or a *digraph*. Given an edge $e = (u, v)$, we call u the *source* of e, v the *destination* of e and u and v the *endpoints* of e. If the nodes and edges of a graph G are associated with labels, we call G a *labeled graph*.

A multigraph is a labeled graph in which multiple edges between two nodes are admitted. Formally:

Definition 2.2. A *multigraph* is a tuple $G = (V, E, \Sigma, \Pi)$, where:

- V is a set of *nodes*;
- E is a multiset of ordered pairs of nodes called *edges*;
- $\Sigma : V \to \mathcal{P}(\mathcal{L}) \setminus \{\emptyset\}$ is the node label function, which associates one or more distinct labels in a set $\mathcal{L}$ to each node ($\mathcal{P}(X)$ denote the power set of X);
- $\Pi : E \to \mathcal{T}$ is the edge label function, which associates one label from the set $\mathcal{T}$ to each edge.

Given two nodes u and v in G, the *multiplicity* of pair (u, v) is the number of edges connecting u and v. The *maximum edge multiplicity* of G is the maximum multiplicity among all possible pair of nodes in G.

Fig. 1 depicts two examples of multigraphs P and G. In this example and in all subsequent ones, the color of an edge represents its label, while progressive numbers shown on edges' arrows are used to distinguish multiple edges connecting two nodes.

Definition 2.3. Given a multigraph $P = (V_P, E_P, \Sigma_P, \Pi_P)$ and a multigraph $G = (V_G, E_G, \Sigma_G, \Pi_G)$, the *SubMultigraph Matching* (SMM) problem consists in finding an injective function $f : V_P \to V_G$, called *node mapping*, and an injective function $g : E_P \to E_G$, called *edge mapping*, such that the following conditions hold:

(1) $\forall e = (u, v) \in E_P, g(e) = (f(u), f(v)) \in E_G$;
(2) $\forall u \in V_P, \Sigma(u) \subseteq \Sigma(f(u))$;
(3) $\forall e \in E_P, \Pi(e) = \Pi(g(e))$.

Condition 1 ensures that edge mapping g is consistent with node mapping f, i.e., neighbor nodes in P are mapped to neighbor nodes in G. Condition 2 states that all the labels of a node in P must be present among the labels of the corresponding node in G. Likewise, Condition 3 states that mapped edges in P and G must have the same label.

Given an edge mapping g, an *occurrence* of P in G is a multigraph S formed by edges $g(e_1), g(e_2), \ldots, g(e_k)$ and all nodes that are endpoints of these edges. Note that, since the edge mapping

[4] Note that, in the context of FSM, the notion of a "pattern" corresponds to that of a frequent subgraph. We acknowledge that, in the context of conceptual modeling, the statistical relevance of a subgraph is not a sufficient condition to identify it as a pattern, since some structures may be useful or interesting even if they are not highly recurrent. In this respect, the approach we propose should be regarded as a technique for identifying recurrent structures, intended to support a possible subsequent pattern identification phase.

[5] From now on, we use "network" and "graph" interchangeably.

is injective, if P contains multiple edges with the same label between two nodes u and v, the occurrence S of P in G must also contain a corresponding number of multiple edges with the same label between nodes $f(u)$ and $f(v)$, where f is the node mapping consistent with g.

If all edges in G that connect nodes of S are mapped through g, we say that S is an *induced* occurrence of P in G, otherwise we say that S is a *non-induced* occurrence of P in G. The number of (induced or non-induced) occurrences of P in G is also called the *frequency* of P in G.

In Fig. 1 multigraph P occurs as a subgraph in target G. A possible occurrence is the graph whose nodes and edges are drawn with dashed lines. This occurrence is non-induced, because there are two extra brown edges in G (one from g_3 to g_2 and one from g_2 to g_5) and an extra green edge from g_3 to g_4, that are not included in the occurrence. If these three edges were not present in G, the subgraph of G formed by g_2, g_3, g_4, g_5 and all edges connecting them would be an induced occurrence of P in G.

Definition 2.4. Given a database $\mathcal{DB} = \{G_1, G_2, \ldots, G_k\}$ with k multigraphs, the *support* $supp(P, \mathcal{DB})$ of a multigraph P is the number of graphs in $\mathcal{DB}$ in which P occurs.

Definition 2.5. Let $\mathcal{DB} = \{G_1, G_2, \ldots, G_k\}$ a database with k multigraphs and σ a support threshold. The *Frequent Subgraph Mining (FSM)* problem consists in finding all multigraphs P such that $supp(P, \mathcal{DB}) \geq \sigma$, also called *frequent subgraphs*.

A frequent subgraph P is *closed* if there is no larger multigraph containing P that has the same support. A frequent subgraph P is *maximal* if no larger multigraph containing P is also frequent.

As an example, given the database $\mathcal{DB}$ of 3 graphs illustrated in Fig. 2 and $\sigma = 2$, P is a frequent subgraph as it occurs in all 3 graphs.

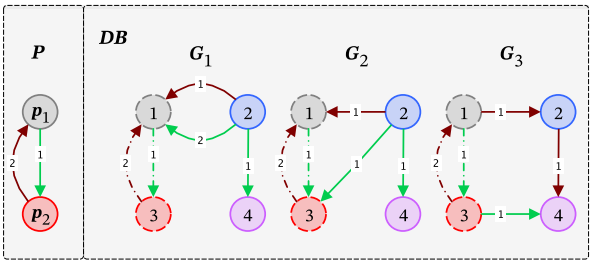


Figure 2: A frequent subgraph P in a database of graphs $\mathcal{DB} = \{G_1, G_2, G_3\}$, given $\sigma = 2$. Nodes and edges of P's occurrences in the $\mathcal{DB}$ are drawn with dashed lines.

3 Problem Requirements

Through discussions with experts in pattern discovery for conceptual models, we identified key requirements for the mining problem. This phase involved ten potential users, including modeling language designers and experienced conceptual modeling researchers, namely representative users of the proposed solution. Based on open-ended interviews and an analysis of existing FSM tools, we inferred the following key issues:

(1) **Graph Complexity:** Conceptual models can be encoded as graphs that are highly complex, e.g. multigraphs with multiple labels for both nodes and edges. Currently available and functioning FSM tools are designed to process only some specific graph types. This necessitates transforming the conceptual model graphs to fit the capabilities of existing algorithms. For instance, many FSM algorithms are designed only for undirected graphs. To use them for conceptual models with directed edges, one may need to reify every edge and keep track of the directionality with new edges [11]. This has an impact both on the algorithm's efficiency and the interpretability of its results.

(2) **Frequency Types:** When looking for patterns in conceptual models, it is important to analyze not only the frequency of the corresponding graph structure across multiple input models but also its frequency within each individual model. A pattern may be infrequent across models but highly recurrent within a specific one, providing insight into its domain-specific usage. These different frequency types are typically supported by distinct FSM algorithms, some designed for single-graph mining, others for mining across datasets.

(3) **Goal-Oriented Search:** To constrain and guide the search space of the algorithm, it is useful to enable user-defined queries that filter structures by specific characteristics—e.g., nodes with predefined labels, or tree-shaped structures only. Such goal-oriented search capabilities would align the mining process with the research objectives of the end users. This scenario is particularly common in conceptual model pattern discovery and would significantly reduce the number of irrelevant outputs. Currently, no existing FSM solutions support this type of operation.

(4) **Resource Constraints:** The target users of such a tool are typically not data scientists or machine learning experts. They often operate on standard laptops with limited computing power. Therefore, it is crucial to offer a solution capable of handling large graphs on such machines. One of the major limitations of current FSM algorithms is their scalability, which restricts their practical applicability.

These issues motivated the development of CMiner and have been distilled into the following requirements:

- **R1:** Ability to handle multi-directed, multi-labeled graphs.
- **R2:** Ability to compute subgraph frequency both within each input graph and across all input graphs.
- **R3:** Ability to guide the search for recurring structures using user-defined *templates* expressed as input queries.
- **R4:** Ability to operate efficiently on low-performance machines while delivering competitive results compared to the state of the art.

R1, **R2**, **R3**, and **R4** will be considered to evaluate the effectiveness of the proposed algorithm.

4 *CMiner* Algorithm

CMiner takes as input the following parameters:

- $\mathcal{DB}$: a database of multigraphs;
- σ: the minimum support required for subgraphs to be considered frequent;
- T: the list of templates, i.e. subgraphs from which the search must start;
- m: the minimum number of nodes of frequent subgraphs;
- M: the maximum number of nodes of frequent subgraphs;
- $\mathcal{L}^-$: the set of node labels to exclude during the mining;
- $\mathcal{T}^-$: the set of edge labels to exclude during the mining;
- *minFreq*: the minimum frequency that a subgraph must have in each multigraph of $\mathcal{DB}$.

Upon user request, CMiner is also capable of retrieving closed and maximal subgraphs. All input parameters, with the exception of $\mathcal{DB}$ and σ, are optional.

For each retrieved frequent subgraph, CMiner yields: i) its support, ii) its frequency in each graph of $\mathcal{DB}$, iii) the list of graphs of $\mathcal{DB}$ in which the subgraph occurs, and optionally, iv) the list of all occurrences of the subgraph in each graph.

Algorithm 1 describes the pseudocode of CMiner. For clarity, we illustrate the algorithm considering only the mandatory parameters $\mathcal{DB}$ and σ. In Section 4.4, we briefly discuss how to adapt CMiner to support optional parameters.

Algorithm 1 CMiner

```
1:  procedure CMiner(DB, σ)
2:      C := ∅
3:      Sol := ∅
4:      S := Frequent one node patterns
5:      while S ≠ ∅ do
6:          P := pop(S)
7:          Sol := Sol ∪ {P}
8:          N_ext := freq_node_extensions(P, σ)
9:          for n_ext in N_ext do
10:             P' := apply_node_ext(P, n_ext)
11:             if canonical_code(P') ∉ C then
12:                 C := C ∪ {canonical_code(P')}
13:                 push(S, P')
14:                 E_ext := freq_edge_extension_groups(P', σ)
15:                 for Gr in E_ext do
16:                     P'' := apply_edge_ext(P', Gr)
17:                     if canonical_code(P'') ∉ C then
18:                         C := C ∪ {canonical_code(P'')}
19:                         push(S, P'')
20:                     end if
21:                 end for
22:             end if
23:         end for
24:     end while
25:     Return Sol
26: end procedure
```

CMiner uses three data structures to handle subgraphs, $\mathcal{S}$, C, and *Sol*:

- $\mathcal{S}$ is a stack containing the candidate subgraphs;
- C is a set of graph canonical codes that are used to reduce redundancies in the generation of candidate subgraphs;
- *Sol* is the set of all frequent subgraphs identified by CMiner.

The algorithm starts with C and *Sol* empty, while $\mathcal{S}$ contains all one-node frequent subgraphs (lines 2-4).

At each step, a frequent subgraph P is popped from the stack $\mathcal{S}$ (line 6), added to the final set of solutions (line 7) and extended, whenever possible, with a single node and one or more edges (lines 8-23).

To do this, CMiner first finds all possible frequent node extensions of P (line 8). Each node extension generates a new frequent subgraph P' (line 10). When the canonical code of P' is not already present in C (line 11), this is added to $\mathcal{S}$. Then, in a similar way, all possible frequent edge extensions of P' (by adding one or more edges between its nodes) are computed (line 14), to form a larger frequent subgraph. During this process, edge extensions that produce subgraphs occurring in the same set of $\mathcal{DB}$'s graphs are grouped together. Next, for each group, namely Gr, all edge extensions falling within Gr are applied to P' to generate a new subgraph P'' (line 16). P'' is then pushed to $\mathcal{S}$, provided that its canonical code has not been already inserted in C (lines 17-19).

The following sections provide detailed descriptions of key procedures in the CMiner algorithm: node extension, edge extension, and graph canonical code computation.

4.1 Node Extensions

During the search, new frequent subgraphs are retrieved from previously identified smaller frequent subgraphs by first adding a new node. Node extension is performed in two steps. In the first step, CMiner builds a table, called *Node Extension Table* (NET), containing a set of node extension configurations, each describing a possible way to node-extend a subgraph. Next, based on the identified set of configurations, extended frequent subgraphs are built and their occurrences are found in the database of graphs.

Let $\mathcal{DB}$ be a database of multigraphs and let $\mathcal{L}$ and $\mathcal{T}$ be the sets of all node and edge labels present in the graphs of $\mathcal{DB}$. The *node extension configuration* of a subgraph P is a quintuple $n_{ext} = (p, L_{end}, T_{in}, T_{out}, Loc)$ where:

- p is a node of P, called pivot of the configuration;
- $L_{end} \subseteq \mathcal{L}$ is the set of labels that the node to be added, called the endpoint of the configuration, should have;
- $T_{in} \subseteq \mathcal{T}$ is the set of labels of the incoming edges from the endpoint of the configuration to p;
- $T_{out} \subseteq \mathcal{T}$ is the set of labels of the outgoing edges from p to the endpoint of the configuration;
- *Loc* is the set of locations of the configuration, i.e., occurrences of P in $\mathcal{DB}$ in which the extension can be applied.

In other words, a node extension configuration serves as a blueprint to perform an extension. It specifies the subgraph node (referred to as the pivot) from which the extension originates, the labels that the new node to be added (the endpoint) must have, and the labels of all potential edges connecting the pivot to the endpoint.

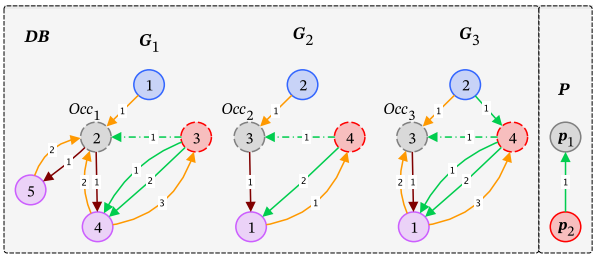
Starting from a subgraph P, its node extension configurations are searched. Once retrieved, the configurations are then collected in the NET. To ensure a compact memory representation of the NET, the labels of the endpoint and the labels of incoming and outgoing edges connecting the pivot and the endpoint are represented using a multi-hot encoding scheme, where bits 0 and 1 are used to indicate the presence or the absence of a specific label for a given configuration. If multiple incoming or outgoing edges share the same label, a separate bit is allocated for each edge.

Fig. 3 illustrates an example of NET built from a two-node subgraph P and a $\mathcal{DB}$ of three graphs G_1, G_2, and G_3. Subgraph P occurs in all three graphs and the nodes and edges of its occurrences in $\mathcal{DB}$ are highlighted using dashed lines.

Considering, for instance, subgraph node p_1 as pivot and a node with purple label as endpoint, there are two possible node extension configurations, C_1 and C_2. C_1 consists of an orange incoming edge and a brown outgoing edge and is located in G_1 and G_3. C_2 consists of only a brown outgoing edge and is located in G_2.

Some NET configurations could be located in a subset of $\mathcal{DB}$ graphs that is lower than the support threshold resulting in non-frequent subgraphs. For this reason, CMiner calculates support for each configuration and filters out configurations whose support is below the threshold.

To correctly calculate the support of a configuration C, we need to count not only the locations of C, but also the locations of all configurations that actually represent an extension of C, i.e., configurations that contain all the labels of the endpoint and the incoming and outgoing edges of C. For example, in Fig. 3, configuration C_1 contains all the labels of configuration C_2, so the total support of C_2 is 3, including the location of C_2 in G_2 and the two locations of C_1 in G_1 and G_3. To check if a configuration C



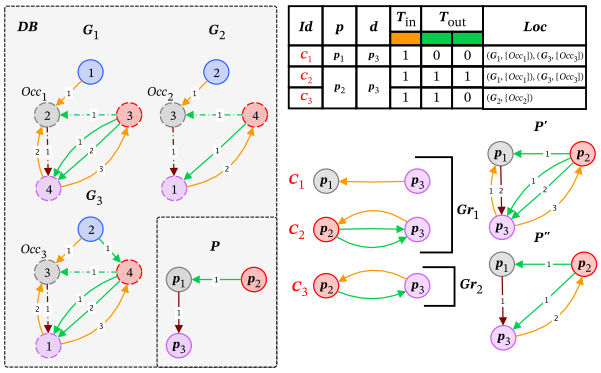
Id	p	L_{end}	T_{in}		T_{out}			Loc
C_1	p_1		1	0	0	0	1	$(G_1,\{Occ_1\}),(G_3,\{Occ_3\})$
C_2			0	0	0	0	1	$(G_2,\{Occ_2\})$
C_3	p_1		1	0	0	0	0	$(G_1,\{Occ_1\}),(G_2,\{Occ_2\}),(G_3,\{Occ_3\})$
C_4	p_2		0	1	0	0	0	$(G_3,\{Occ_3\})$
C_5	p_2		1	0	1	1	0	$(G_1,\{Occ_1\}),(G_3,\{Occ_3\})$
C_6			1	0	1	0	0	$(G_2,\{Occ_2\})$

Figure 3: NET of subgraph P in the database of graphs G_1, G_2, and G_3. Nodes and edges of the occurrences of P in the database are drawn with dashed lines. In the 'Id' column of the NET, configurations yielding frequent node extensions are highlighted in red.

contains a configuration C', we can simply calculate the bitwise AND operation between the bit vectors of labels of C and C'. If the result of the operation is the bit vector of C', then C contains C'.

For a given configuration C, CMiner finally applies all possible node extensions that adhere to C by extending the subgraph occurrences in each graph of the DB with nodes and edges whose labels correspond to those indicated by C.

4.2 Edge Extensions



Id	p	d	T_{in}		T_{out}		Loc
C_1	p_1	p_3	1	0	0		$(G_1,\{Occ_1\}),(G_3,\{Occ_3\})$
C_2	p_2	p_3	1	1	1		$(G_1,\{Occ_1\}),(G_3,\{Occ_3\})$
C_3			1	1	0		$(G_2,\{Occ_2\})$

Figure 4: EET of subgraph P in the database of graphs G_1, G_2 and G_3. Nodes and edges of the occurrences of P in the database are drawn with dashed lines. P has 3 edge extension configurations that produce frequent subgraphs, so they are highlighted in red in the "Id" column of the EET. Configurations having the same set of locations are grouped together, forming groups Gr_1 and Gr_2. Application of all configurations in each group yields subgraphs P' and P''.

Subgraphs obtained by node extension are further extended by adding one or more edges between existing subgraph nodes. This will be obtained according to a pre-computed set of edge extension configurations.

To accomplish this, CMiner first computes the set of all possible edge extension configurations, i.e., all possible ways in which a subgraph P can be edge-extended, based on the occurrences of P in each graph of the database.

Let $\mathcal{T}$ the sets of all edge labels observed in the multigraphs of a database $\mathcal{DB}$. The *edge extension configuration* of a subgraph P is a quintuple $e_{ext} = (s, d, T_{in}, T_{out}, Loc)$ where:

- s and d (with $id(s) < id(d)$) are nodes of P, called source and destination of the configuration, respectively;
- $T_{in} \subseteq \mathcal{T}$ is the set of labels of the edges going from d to s;
- $T_{out} \subseteq \mathcal{T}$ is the set of labels of the edges going from s to d;
- Loc is the set of locations of the configuration, i.e., occurrences of P in $\mathcal{DB}$ in which the extension can be applied.

Therefore, an edge extension configuration is a scheme describing how the extension can be done, but unlike the node extension configuration it involves two specific subgraph nodes.

Starting from a subgraph P, CMiner searches for all possible edge extension configurations, based on the occurrences of P in each database graph, and collects the configurations in a table called *Edge Extension Table* (EET). For a compact representation of the EET, information about the configuration's labels are stored using a multi-hot encoding scheme, where 0 and 1 respectively denote the absence and presence of a specific label. Multiple edges connecting two nodes and having the same direction and label are represented with different bits.

Fig. 4 depicts an example of EET calculated for a three-node subgraph P in a $\mathcal{DB}$ of graphs G_1, G_2, and G_3. Nodes and edges of the occurrences of P in G_1, G_2 and G_3 are highlighted using dashed lines. The EET contains 3 configurations: C_1 for the pair of subgraph nodes (p_1, p_3), and C_2 and C_3 for the pair (p_2, p_3).

After constructing the EET, CMiner calculates for each edge extension configuration C a support, corresponding to the number of database graphs in which the subgraph extended according to C occur. The support of a configuration C is calculated as the sum of the number of locations of C and the number of locations of all configurations having the same source and destination as C and containing all the edge labels of C. Again, to check if a configuration C contains C', we can calculate the bitwise AND between the bit vectors of labels of C and C': if the result of the operation is the bit vector of edge labels of C', then C contains C'. Configurations with a support below the threshold are discarded. In Fig. 4 configurations C_1 and C_2 have support 2, while C_3 has support 3. So, no configuration is discarded by CMiner.

Next, the remaining configurations are grouped according to their locations, so that two configurations having the same set of locations belong to the same extension group. As an example, in Fig. 4 extension configurations C_1 and C_2 have the same set of locations, so they can be included in the same group Gr_1. A second group Gr_2 includes the other configuration C_3.

The application of edge extensions is finally done on the basis of the previously calculated extension groups. For each extension group Gr, CMiner extends the current subgraph (and its occurrences in the database) by simultaneously applying all extensions that adhere to the configurations of Gr. This is done to ensure that all candidate subgraphs generated by edge extension will be induced subgraphs that occur in the database. For example, in Fig. 4 CMiner generates from subgraph P a new subgraph P' that incorporates both edge extensions C_1 and C_2 belonging to group Gr_1. Notice that P' occurs as an induced subgraph in the database, while subgraphs that incorporate only the extension C_1

or only the extension C_2 would occur as non-induced subgraphs. A second extended subgraph P'' is generated from the extension C_3 belonging to group Gr_2.

4.3 Graph Canonical Codes

During the search for frequent subgraphs, the same subgraph can be generated from different node or egde extensions. To avoid duplication, a canonical code is computed for each subgraph. Using canonical codes not only guarantees the uniqueness of the results but also enhances the algorithm's performance.

Computation of the canonical code of a subgraph P with k nodes is performed in two steps. First, given a subgraph P with k nodes, CMiner uses BLISS algorithm [25, 26] to calculate the permutation of P's nodes that transforms P into a canonical form. Following the order of nodes in the permutation, the canonical code of P is calculated as the sequence $L_1R_1L_2R_2 \dots L_kR_k$ where:

- L_i is the lexicographically-sorted sequence of labels of the i-th node of P in the permutation;
- $R_i = (v_{i1}, v_{i2}, \dots, v_{ik})$ is a vector of values, with v_{ij} equals to:
  - The lexicographically-sorted sequence of labels of the outgoing edges from the i-th node of P to the j-the node of P, followed by L_j, if the two nodes are adjacent;
  - 0 if the i-th node of P and the j-the node of P are not adjacent.

An example of the calculation of a canonical code is shown in Fig. 5 for a subgraph P.

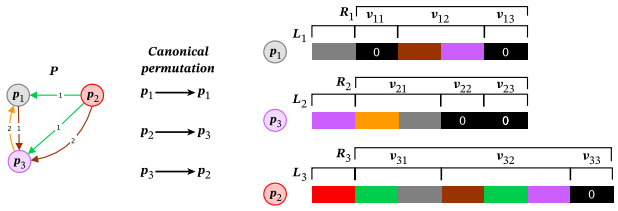


Figure 5: Computation of the canonical code of subgraph P. First, the permutation of P's nodes that puts P into a canonical form is calculated. Following the ordering of nodes in the permutation, the canonical code of P is built, by concatenating node and edge labels. Colored rectangles represent labels of nodes and edges, while black rectangles with "0" indicate the absence of edges connecting two nodes.

4.4 Handling Optional Parameters

CMiner can be easily adapted to handle the optional parameters.

If a list of templates T is given as input, CMiner first identifies occurrences of each template in all database graphs using MultiGraphMatch [29], a subgraph matching algorithm for multigraphs. Then, the support of each template is calculated, and all frequent templates are pushed into the stack $\mathcal{S}$ before starting the mining process.

If parameter m is set, then only frequent subgraphs with at least m nodes are added to the set of final solutions Sol.

If parameter M is specified, node extensions are performed only if a subgraph contains fewer than M nodes.

If one or more node and/or edge labels should be excluded during mining (parameters $\mathcal{L}^-$ and $\mathcal{T}^-$), all nodes and edges containing these labels are removed from the database's graphs (along with the edges whose endpoints are the removed nodes) before starting the mining process.

If a minimum frequency threshold $minFreq$ is set, then each time a node or edge extension is applied, CMiner checks if the frequency F of the newly generated subgraph P is above the threshold in each graph G of the database. If $F < minFreq$ in a graph G, then P is deemed as not occurring in G. This means that the support of P is calculated considering only the database graphs in which P occurs at least $minFreq$ times.

5 Complexity

To derive the complexity of CMiner, we will rely on the following two lemmas:

Lemma 5.1. *Let $G = (V, E)$ be a graph with maximum node degree Δ. Then the number of connected induced subgraphs of size M is at most $O(|V| \cdot (c\Delta)^{M-1})$, for some constant c.*

Lemma 5.2. *Let $c := \big((M-1)!\big)^{1/(M-1)}$. Then $c = \Theta(M)$.*

The proofs are provided in the appendix **??**.

Number of candidate patterns. Let $|\mathcal{DB}|$ be the number of input graphs, $|V|$ denote the total number of nodes across the database, $|E|$ the total number of edges, and Δ the maximum node degree observed in the database. In the worst case of dense inputs, every non-empty node subset is connected and the number of subgraphs up to M nodes grows as $2^{\min\{M, |V|_{\max}\}}$, where $|V|_{\max}$ is the largest graph in the database. By Lemma 5.1, when Δ is bounded, the number of connected induced subgraphs of size up to M over the whole database is $O\big(|V|\,(c\Delta)^{M-1}\big)$, i.e., the exponential factor depends on Δ rather than on $|V|_{\max}$, which yields a much smaller search space when $\Delta \ll |V|_{\max}$. Using Lemma 5.2, we take $c = \Theta(M)$ and write $(c\Delta)^{M-1} = \Theta((M\Delta)^{M-1})$.

5.0.1 *Time complexity.*

Cost per subgraph. For a subgraph P of size $k \le M$, the main costs are related to NET, EET and canonical labeling. For each graph, the construction of the NET scans the frontier incident to the nodes of P *for every occurrence of* P. For a single occurrence in graph G_i, the frontier's size is bounded by $\min\{|E_i|, k\Delta\}$ and each frontier edge is processed once with word-level updates; thus the per-occurrence cost is $O(\min\{|E_i|, k\Delta\})$. Summing over all occurrences across the database gives the NET construction cost $T_{\text{NET}}(P)$

$$\sum_{i=1}^{|\mathcal{DB}|} \text{occ}_i(P) \cdot \min\{|E_i|, k\Delta\} \;\le\; \text{occ}(P) \cdot \min\{|E|, |\mathcal{DB}|\,k\Delta\} = T_{\text{NET}}(P),$$

After a valid node extension, the EET enumerates only candidate edges between nodes already in the current subgraph P. For a single occurrence, the number of pairs is at most $k(k-1)/2$, each checked in $O(1)$ at the word level, therefore the EET construction costs $T_{\text{EET}}(P) = O\big(\text{occ}(P) \cdot k^2\big)$ over the whole database. The canonical labeling of each new subgraph is performed with the `BLISS` algorithm [25]. While canonical labeling is exponential in the worst case and graph isomorphism is only known to admit a quasi-polynomial bound [1], BLISS is highly efficient in practice due to partition refinement and automorphism group pruning. Given that our subgraphs have size at most M, we denote the cost by

$$T_{\text{can}}(P) = O\big(\text{occ}(P) \cdot T_{\text{can}}(k)\big) \le O\big(\text{occ}(P) \cdot T_{\text{can}}(M)\big).$$

Overall time complexity. With the per-subgraph costs $T_{\text{NET}}(P)$, $T_{\text{EET}}(P)$, and $T_{\text{can}}(P)$, the total runtime is

$$T_{\text{DB}} = O\left(\sum_{P:\,|P|\le M} \left(T_{\text{NET}}(P) + T_{\text{EET}}(P) + T_{\text{can}}(P)\right) \right). \quad (1)$$

By Lemma 5.1, $\sum_P \text{occ}(P) = O\big(|V|\,(c\Delta)^{M-1}\big) = \Theta\big(|V|\,(M\Delta)^{M-1}\big)$, hence

$$T_{\text{DB}} \;=\; O\Big(|V|\,(M\Delta)^{M-1}\cdot\big(\min\{|E|,\,|\mathcal{DB}|\,M\Delta\}+M^2+T_{\text{can}}(M)\big)\Big).$$

Sparse case. When the database is sparse and Δ is small, the term $\min\{|E|,\,|\mathcal{DB}|\,M\Delta\}$ collapses to $|E|$, yielding

$$T_{\text{DB}}^{(\text{sparse})} = O\Big(|V|\,(M\Delta)^{M-1}\cdot\big(|E|+M^2+T_{\text{can}}(M)\big)\Big).$$

Dense case. For dense inputs, with Δ large (up to $\Delta \approx |V|_{\max}-1$), the minimum is attained by $|\mathcal{DB}|\,M\Delta$, giving

$$T_{\text{DB}}^{(\text{dense})} = O\Big(|V|\,(M\Delta)^{M-1}\cdot\big(|\mathcal{DB}|\,M\Delta+M^2+T_{\text{can}}(M)\big)\Big).$$

In the latter regime, $(M\Delta)^{M-1}$ recovers the classical worst-case growth, while in the sparse regime the runtime is near-linear in $|E|$ per exponential layer $(M\Delta)^{M-1}$ of the search space.

Our theoretical bounds use Δ as a worst–case upper bound. This is intrinsically pessimistic, since few hubs can inflate Δ while the typical frontier growth is governed by the *average* connectivity. In practice, it is more informative to replace Δ with the *mean degree* or with an *effective degree* that ignores top quantiles of the degree distribution.

5.0.2 *Space complexity.*

Tables NET and EET. We analyze memory *per single subgraph* P of size $k \le M$. Let w denote the machine word size. For each table $\mathsf{X} \in \{\text{NET}, \text{EET}\}$ we decompose the space as

$$S_{\mathsf{X}}(P) \;=\; \#\text{rows}_{\mathsf{X}}(P) \;\times\; \text{row\_size}_{\mathsf{X}}(P).$$

Thus, for both NET and EET we will (i) bound the *number of rows* generated by all occurrences of P in $\mathcal{DB}$, and (ii) bound the *space per row* given our packed, multi–hot encoding. We will use $\lceil \cdot/w \rceil$ to count the number of machine words needed to store a bitset.

Space for the NET. One NET row encodes a node–extension configuration $n_{\text{ext}} = (p, L_{\text{end}}, T_{\text{in}}, T_{\text{out}}, Loc)$. Hence the total number of rows depends on (i) the number of subgraph nodes $|V(P)| = k \le M$, (ii) the possible endpoint–label sets on the frontier ($L_{\text{end}} \subseteq \mathcal{L}$), and (iii) the distinct multi–edge configurations between pivot and endpoint observed in $\mathcal{DB}$ that are at most $2^{2\Delta}$, 2Δ because we consider both directions:

$$\text{rows}_{\text{NET}}(P) \;=\; O\big(M \cdot 2^{\,|\mathcal{L}|+2\Delta}\big).$$

Each row stores: (i) the pivot identifier p (constant space); (ii) the endpoint label set $L_{\text{end}} \subseteq \mathcal{L}$ as a multi–hot bitset requiring $\lceil |\mathcal{L}|/w \rceil$ words; (iii) the edge–label vectors T_{in} and T_{out}. Since we work on multigraphs, multiple parallel edges with the same label are possible; our encoding allocates one bit per *edge instance*. However, the total number of incident edges that can appear between p and the endpoint across a single configuration is bounded by the degree of the pivot $deg(p) = O\big(\Delta\big)$, hence $|T_{\text{in}}| + |T_{\text{out}}| \le \Delta$. Packing these bits uses $O(\lceil \Delta/w \rceil)$ words; (iv) the location field Loc, if materialized as a compressed bitmap over the $\text{occ}(P)$ occurrences, it needs $\lceil \text{occ}(P)/w \rceil$ words. Therefore the row footprint is:

$$\text{row\_size}_{\text{NET}}(P) \;=\; O\Big(\big\lceil \tfrac{|\mathcal{L}|}{w} \big\rceil + \big\lceil \tfrac{\Delta}{w} \big\rceil + \big\lceil \tfrac{\text{occ}(P)}{w} \big\rceil\Big)$$

Space for the EET. One EET row encodes an edge–extension configuration $e_{\text{ext}} = (s, d, T_{\text{in}}, T_{\text{out}}, Loc)$ found while scanning all occurrences of P in $\mathcal{DB}$. The total number of rows depends on (i) the number of unordered node pairs in the subgraph, $\binom{|V(P)|}{2}$, and (ii) the distinct multi–edge configurations between s and d observed in $\mathcal{DB}$. With at most Δ parallel edges per direction, again, the edge-side choices contribute $2^{2\Delta}$ overall. Therefore,

$$\text{rows}_{\text{EET}}(P) \;=\; O\left(\binom{|V(P)|}{2} \cdot 2^{\,2\Delta}\right) \;=\; O\big(M^2 \cdot 2^{\,2\Delta}\big).$$

Each row stores: (i) the pair (s, d) (constant space); (ii) the edge–label vectors T_{in} and T_{out} as multi–hot bitsets over *edge instances* between s and d. In a multigraph, multiple parallel edges with the same label are possible; our encoding allocates one bit per edge instance. For one configuration, the total number of bits set is bounded by the out–/in–degrees of the pair, hence $|T_{\text{in}}| + |T_{\text{out}}| \le 2\Delta$ and packing uses $O(\lceil \Delta/w \rceil)$ words; (iii) the location field Loc, which, if materialized as a compressed bitmap over the $\text{occ}(P)$ occurrences, needs $\lceil \text{occ}(P)/w \rceil$ words. Therefore, the row footprint is

$$\text{row\_size}_{\text{EET}}(P) \;=\; O\Big(\big\lceil \tfrac{\Delta}{w} \big\rceil + \big\lceil \tfrac{\text{occ}(P)}{w} \big\rceil\Big).$$

Space for canonical codes. Let $L_{\text{code}}(k)$ be the length (in machine words) of the canonical code of a k-node subgraph. Considering packed bitsets, each node contributes a bitset over $\mathcal{L}$, so $k \cdot \lceil |\mathcal{L}|/w \rceil$ words, while each ordered pair (i, j) we store v_{ij}. If i and j are adjacent, v_{ij} contains the bitset of edge labels (over $\mathcal{T}$) and the node-label payload of j; otherwise $v_{ij} = 0$. In the dense worst case there are k^2 entries, each costing $\lceil |\mathcal{T}|/w \rceil + \lceil |\mathcal{L}|/w \rceil$ words. Therefore, storing one code per generated subgraph costs

$$S_{\text{codes}} = O\Big(|V|\,(M\Delta)^{M-1}\cdot M^2 \cdot \big\lceil \tfrac{|\mathcal{T}|+|\mathcal{L}|}{w} \big\rceil\Big).$$

Stack. We adopt *depth–first* subgraph growth because it minimizes peak memory. With DFS, at most one subgraph per depth level is kept. Let $S_{\text{pat}}(k)$ be the memory footprint of a k-node subgraph:

$$S_{\text{pat}}(k) = O\Big(k^2 \cdot \Big\lceil \tfrac{|\mathcal{T}|+|\mathcal{L}|}{w} \Big\rceil\Big).$$

The stack height is $\le M$, hence

$$S_{\text{stack}}^{\text{DFS}} = O\Big(\sum_{k=1}^{M} S_{\text{pat}}(k)\Big) = O\Big(M \cdot S_{\text{pat}}(M)\Big) = O\Big(M^3 \cdot \Big\lceil \tfrac{|\mathcal{T}|+|\mathcal{L}|}{w} \Big\rceil\Big).$$

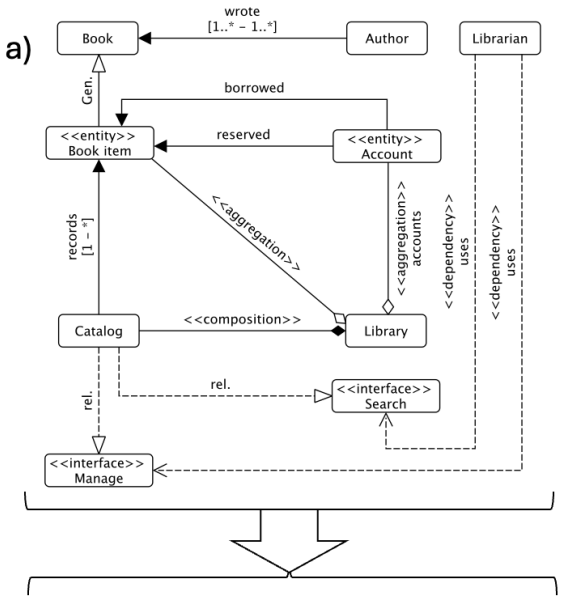
6 Validation

In this section, we present the experimental analysis designed to validate CMiner. The tests we run aim to satisfy the four requirements described in Section 3. The datasets, the output structures, and the scripts to generate the synthetic graphs are available at `https://github.com/SimoneAvellino/CMiner/`. Tests 1, 2, and 3 were conducted on a *Apple M2, 8GB RAM.* Test 4 was conducted on 30 cores CPU Intel(R) Xeon(R) Gold 6132 CPU, 256 GB of RAM and we used parallelized CMiner.

6.1 Test 1

The goal of this test is to demonstrate the ability of CMiner to accurately identify frequent subgraph structures across a set of directed multigraphs. It returns the correct frequency values for each structure, both globally (across all graphs) and locally (within each individual graph).

Research Questions. This experiment addresses research questions R1 and R2.



b)

```
t # 1
v 0 Librarian
v 1 Interface Search
v 2 Catalog
v 3 Library
v 4 Entity BookItem
v 5 Book
v 6 Author
v 7 Entity Account
v 8 Interface Manage
e 0 1 dependency
e 0 1 uses
e 0 8 dependency
e 0 8 uses
e 2 1 realization
e 2 3 composition
e 2 4 records
e 2 8 realization
e 3 2 composition
e 4 3 aggregation
e 4 5 Generalization
e 6 5 1*-1*
e 6 5 wrote
e 7 4 borrowed
e 7 4 reserved
s 20
f 200
```

Figure 6: Target structure to be found in Test 1. a) UML representation, b) CMiner output.

Setup. To ensure full control over the number and distribution of subgraph occurrences, we constructed a dataset of synthetic graphs, whose statistics are summarized in Table 1. Each graph was automatically generated using a custom script that takes as input the features that the graphs must have (e.g. number of nodes, number of edges, multi-labels) and allows the specification of a pre-defined support of a specific subgraph P and a pre-defined frequency of P in each graph of the database. In this experiment, we set 20 and 200 as support and frequency, respectively.

Figure 6a represents the conceptual modeling target structure to be found in the synthetic graphs.[6] The structure includes all the graph features we want to find and is a directed multigraph with multiple labels on nodes and edges.

Table 1: Statistics of the synthetic dataset used in Test 1. "Time" is the running time range (secs) of CMiner to retrieve the pattern of Figure 6 in the dataset.

Graphs	Nodes	Edges	Avg nodes	Avg edges	Time (secs)
20	18 - 1350	30 - 2250	166.5	281.9	10.4 - 14.3

[6]The pattern is inspired by the example provided at https://www.uml-diagrams.org/library-domain-uml-class-diagram-example.html

Results. The algorithm successfully identified the target structure within each graph, and the returned frequency values matched the ground truth, thereby confirming the expressivity and the accuracy of the mining process (as required by **R1** and **R2**). Figure 6b represents the output produced by CMiner for the UML structure depicted in Figure 6a. It contains the list of pattern nodes (lines starting with `v`) followed by the list of pattern edges (lines starting with `e`), and finally the support and the frequency of the structure (lines starting with `s` and `f`, respectively). Multiple labels associated to a node (e.g. "`v 1 Interface Search`") are listed in the same line, while multiple edges between two nodes (e.g. "`e 0 8 dependency`" and "`e 0 8 uses`") are reported in separate lines.

6.2 Test 2

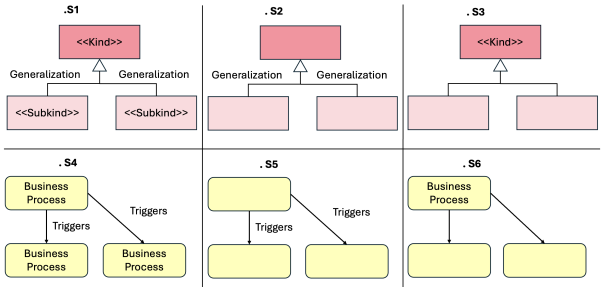


Figure 7: Structures used for template-guided mining of OntoUML and Archimate datasets in Test 2.

This test evaluates the impact of using **template-guided mining**, a key feature of CMiner which allows users to guide the mining process through template structures. The aim is to demonstrate that: i) CMiner is able to extend structures that are constrained to a starting template, ii) the use of templates can effectively reduce the number of irrelevant structures in the output and tailor the results to specific analytical goals.

Research Question. This experiment primarily addresses research question R3.

Setup. We used two datasets extracted from conceptual model repositories that encode conceptual models using widely known modeling languages: OntoUML [32][7] and ArchiMate [15].[8] Each dataset contains a collection of representative graphs of typical structures in their respective domains. A summary of their characteristics is provided in Table 2. We first performed a "blinded" trial on both datasets, in which CMiner was run without providing any input pattern. Next, we tested CMiner on the set of 6 template structures depicted in Fig. 7, which represent two different types of topologies with variable degree of specification of node and edge labels. More specifically, we ran CMiner to mine the OntoUML dataset starting from structures **S1**, **S2** and **S3** and the ArchiMate dataset starting from structures **S4**, **S5** and **S6**. In all trials, we set the support threshold $\sigma = 5$ and the maximum number of nodes of the frequent structures found by CMiner $M = 6$.

Results. Table 3 reports, for each trial, the number of solutions found by CMiner together with its running time. As expected, in the "blinded" test, the number of solutions found is significantly higher, especially with respect to the trials in which structure

[7]https://github.com/OntoUML/ontouml-models
[8]https://github.com/me-big-tuwien-ac-at/EAModelSet

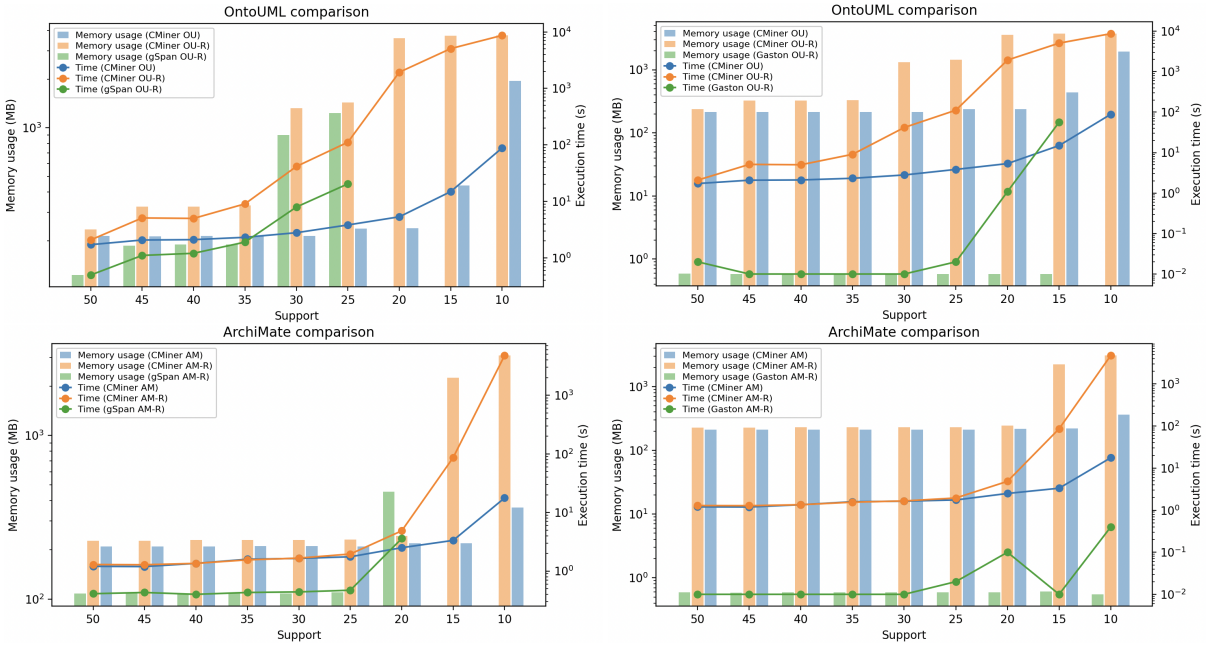


Figure 8: Memory usage and execution time of CMiner compared to gSpan and Gaston on OntoUML (top row) and ArchiMate (bottom row) datasets for the trials of Test 3 (as from Table 4).

Table 2: Statistics of datasets used in Test 2.

Metric	OntoUML	ArchiMate
Number of graphs	94	100
Average nodes	62.28	58.90
Average edges	91.30	68.07
Max nodes	1222	100
Max edges	1723	100
Min nodes	10	40
Min edges	9	32
Std. dev. of nodes	135.43	15.31
Std. dev. of edges	203.77	16.66

labels are fully specified (structures **S1** and **S4**). This confirms the key role of the template-guided mining feature in reducing the search space of the algorithm by making it easy the analysis of the most relevant outputs and extensions (as required by **R3**).

Table 3: Number of solutions found and running time (secs) of CMiner for the "blinded" test and the patterns S1-S6 of Figure 7 in the datasets of Test 2.

Trial	Solutions	Datasets	Time (secs)
Blinded	1378	OntoUML	303.16
S1	120	OntoUML	25.3
S2	675	OntoUML	151.5
S3	341	OntoUML	42.1
Blinded	2750	Archimate	74.21
S4	2	Archimate	0.28
S5	10	Archimate	0.47
S6	40	Archimate	3.85

6.3 Test 3

The third test evaluates the computational efficiency of CMiner in comparison with two widely used state-of-the-art algorithms, **gSpan** and **Gaston**, implemented in Python and C++, respectively. The comparison is based on *execution time*, *memory usage*, and the *number of solutions identified*.

Research Question. This experiment primarily addresses **R4**.

Setup. We considered the OntoUML and Archimate datasets already used in Experiment 2. Additionally, we generated modified versions of both datasets in which the relationships in the conceptual models were reified. Reification involves transforming edges representing relationships, such as "Generalisation" in Figure 6, into new nodes, which are then connected to the source and the target nodes of the original edge through new edges labeled as "source" and "target", respectively. This transformation simulates a scenario in which the structure of a graph must be adapted so that algorithms like gSpan and Gaston (which work only on undirected networks) can capture the directionality inherent in conceptual model graphs. Additionally, since Gaston does not work with string labels, we created an *ad hoc* version of the graphs with the labels mapped to numeric identifiers. This involved a dedicated step when interpreting Gaston's output, where we had to trace back the labels of each pattern to the source strings. Finally, we ran CMiner on both non-reified and reified networks and compared it with gSpan and Gaston, which were tested only on reified networks. Experiments were performed by varying the support threshold from 10 to 50.

Results. Figure 8 and Table 4 show memory usage and running time of CMiner on both non-reified and reified versions of OntoUML and Archimate datasets, and memory usage and running time of gSpan and Gaston on the reified versions of the two datasets. Accordingly, Table 4 reports the number of solutions found by both algorithms in these experiments.

At first glance, Gaston clearly outperforms the other methods. This result is unsurprising, given that the version of the algorithm we adopted is implemented in C++. However, a closer analysis shows that CMiner is competitive in several respects.

First, as the support threshold decreases, and thus the computational effort increases, CMiner becomes more competitive. In particular, when compared with gSpan, we observe that beyond a certain support threshold (25/20), gSpan runs into memory overflow, preventing it from producing any results. Gaston faces the same issue with the OntoUML dataset.

Second, CMiner avoids the need for reified graphs to capture the directionality of arcs, which leads to notable performance gains.

Third, CMiner provides the additional functionality of calculating the frequency of each output pattern within individual graphs. While this requires an extra step in the mining process and incurs significant additional memory usage, CMiner nevertheless never crashes in any experiment and, in several cases, continues to achieve performance close to that of its competitors. As a final note, Gaston tends to return a larger number of patterns. This is because when a cyclic subgraph is present, the corresponding tree subgraph is counted first, followed by the cyclic version of the same subgraph.

Table 4: CMiner vs. GSpan vs. Gaston in ArchiMate (AM) and OntoUML (OU) datasets considering both non-refied and refied (-R) versions. 'S' = solutions; 'T' = time; 'M' = memory; 'Of' = memory overflow.

Performance Comparison on OntoUML (OU) Dataset

Support	CMiner OU			CMiner OU-R			gSpan OU-R			Gaston OU-R		
	S	T (s)	M (MB)	S	T (s)	M (MB)	S	T (s)	M (MB)	S	T (s)	M (MB)
50	1	1.72	214	9	2.09	236	9	0.5	123	9	0.02	0.59
45	3	2.08	213	14	5.11	325	14	1.11	187	14	0.01	0.58
40	3	2.11	214	18	5.01	325	18	1.21	190	18	0.01	0.59
35	5	2.32	213	30	9.08	331	30	1.91	190	30	0.01	0.58
30	8	2.80	214	47	41.67	1323	47	7.98	900	47	0.01	0.59
25	10	3.84	238	83	111.71	1433	83	20.48	1233	85	0.02	0.58
20	26	5.35	239	117	1936.46	3593	-	-	Of	216	1.09	0.58
15	55	14.94	441	344	5065.82	3715	-	-	Of	496	56.7	0.58
10	163	88.60	1952	993	8675.23	3728	-	-	Of	-	-	Of

Performance Comparison on ArchiMate (AM) Dataset

Support	CMiner AM			CMiner AM-R			gSpan AM-R			Gaston AM-R		
	S	T (s)	M (MB)	S	T (s)	M (MB)	S	T (s)	M (MB)	S	T (s)	M (MB)
50	0	1.20	210	1	1.29	227	1	0.41	108	1	0.01	0.59
45	0	1.19	210	1	1.28	227	1	0.43	108	3	0.01	0.58
40	0	1.35	210	2	1.35	230	2	0.40	107	6	0.01	0.59
35	2	1.59	211	12	1.55	230	12	0.43	108	14	0.01	0.59
30	2	1.64	211	16	1.66	230	16	0.44	108	21	0.01	0.59
25	3	1.75	210	28	1.95	231	29	0.47	110	34	0.02	0.59
20	12	2.50	219	67	4.85	244	68	3.58	453	78	0.10	0.59
15	31	3.33	220	189	86.23	2250	-	-	Of	201	0.01	0.61
10	114	17.64	363	911	4765.82	3082	-	-	Of	939	0.40	0.55

6.4 Test 4

In Test 4, we aim at evaluating the scalability of CMiner on big datasets, by considering the *number of solutions identified* and *memory usage*.

Research Questions. This experiment primarily addresses **R4**.

Setup. To evaluate scalability, we generated different databases of multigraphs with a varying number of edges, which was set according to a Gaussian distribution $N(\mu, \sigma)$ with $\mu \in \{1000, 3000, 5000\}$ and $\sigma = 150$. Multigraphs were extracted from a real-world social network (LSQB benchmark[9]) and generated using a controlled graph toolkit.[10] We generated databases with varying number of graphs $|\mathcal{DB}| \in \{10, 50, 100\}$. To this aim, we adopted a nested database design in which smaller datasets are contained within larger ones. This ensures that CMiner processes identical core subgraphs, allowing scalability tests to rely on consistent graph structures among different databases. The combination of all possible values of μ and $|\mathcal{DB}|$ led to 9 different experiments. In each experiment, we set a timeout of 30 minutes, searching for frequent subgraphs of 5 nodes with minimum support equal to 5%. The selected timeout allowed us to observe the results in a reasonable time frame. We plan to test the algorithm with a longer timeout in future work. Peak memory usage was monitored to assess resource efficiency under stress.

Results. In all 9 experiments CMiner reached the timeout before finishing. Figure 9 reports the number of solutions found $|\mathcal{S}|$ before the timeout and, for each value of $|\mathcal{DB}|$, the average peak memory usage across the different graph densities.

Memory consumption exhibits sub-linear growth: a $10x$ increase in database cardinality ($|\mathcal{DB}|$ from 10 to 100) results in only a 1.6× increase in mean RAM usage (from 5.0 GB to 8.1 GB). This confirms that CMiner's memory footprint is efficiently decoupled from the input volume. The number of solutions found before the timeout reflects the increasing computational cost of the mining task. As the size of the database grows to $|\mathcal{DB}| = 100$ and the average number of edges of each graph peaks at $\mu = 5k$, the search space expands significantly, leading to a reduced number of solutions found. However, this reduction is not exponential but inversely proportional to the growth of the number of graphs in the database: for instance a 10x increase of $|\mathcal{DB}|$ from 10 to 100, corresponds to a 10x decrease of $|\mathcal{S}|$. Therefore, CMiner proves effective even in these computationally intensive settings, successfully extracting hundreds of high-support patterns within the time limit, confirming its ability to navigate complex search spaces.

7 Discussion

The experiments assess how CMiner addresses the requirements outlined at the beginning of this article. Overall, the proposed approach proves to be competitive for several key reasons.

First, CMiner supports mining over multi-directed graphs with multiple labels. This capability is essential for preserving and tracing the diverse pieces of information typically embedded in conceptual models, as illustrated in Test 1. Beyond expressiveness, this feature enables the discovery of structure that may encode new interesting patterns or well-known modeling patterns, such as the one shown in Figure 10. These patterns can offer valuable insights, for instance, into how a modeling language like OntoUML is actually used in practice.

Second, CMiner supports a guided discovery process for recurring structures. Users can actively steer the mining task by formulating targeted queries, thereby focusing the analysis on specific structures of interest and reducing the volume of irrelevant results.

As demonstrated in Test 2, this mechanism allows the discovery process to be both more efficient and more aligned with the analyst's intent. Third, as shown in Tests 3 and 4, CMiner consistently produces results even when operating on large graphs, including scenarios characterized by low input support, where the mining task becomes inherently more computationally demanding.

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

[10] https://github.com/SimoneAvellino/GraphToolkit

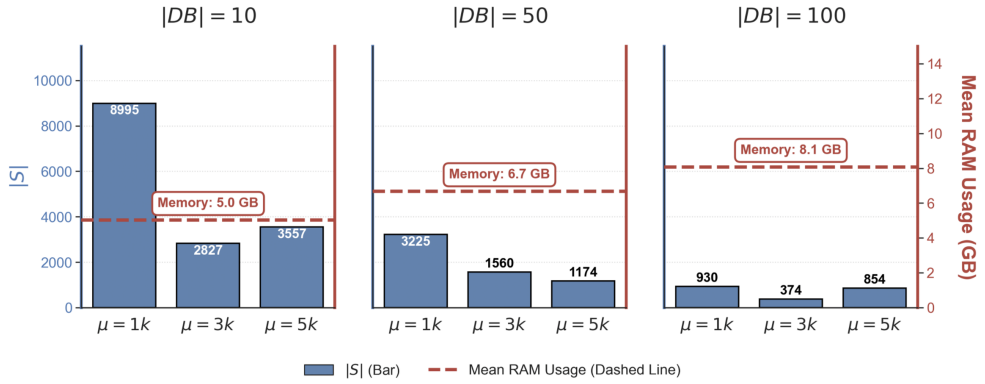


Figure 9: Number of solutions found $|\mathcal{S}|$ and memory usage of CMiner in the datasets of Test 4, on varying $\mathcal{DB}$, the number of graphs in the database, and μ, the average number of edges in each graph of the database.

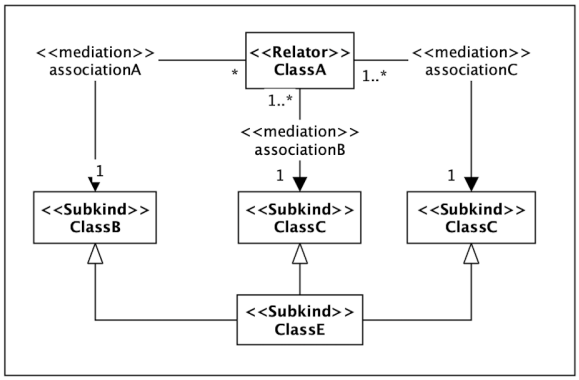


Figure 10: Example of a known OntoUML pattern (RelOver) [33] that can be discovered by CMiner.

It is worth emphasizing that our evaluation deliberately compares CMiner against two algorithms that represent established baselines in FSM research, one of which is implemented by the highly performant language C++. Despite this, CMiner remains highly competitive. In Test 3, when executed on a standard laptop, it never crashes and successfully reports two distinct notions of frequency for each discovered subgraph: frequency across the input graphs and frequency within individual graphs. Providing this additional information requires several extra processing steps compared to existing approaches. Likewise, the results of Test 4 further confirm that CMiner maintains both scalability and stability when applied to large inputs.

With respect to limitations, it is fair to note that CMiner, like any FSM algorithm, faces inherent challenges in terms of execution time. However, within the context considered in this work, execution time is not the primary concern. What matters most is the algorithm's ability to reliably and exhaustively identify all subgraphs that satisfy the required constraints. From this perspective, completeness and correctness take precedence over raw performance. In addition, the template-based guided search mechanism is explicitly intended to mitigate efficiency issues by constraining and reducing the search space.

Finally, returning to the motivation stated at the outset of this article, CMiner should be viewed as a tool for identifying recurring subgraphs in graphs that encode the rich semantics of conceptual models (possibly, both at the schema and instance levels). The meaning of nodes and relationships can be traced through the multiple labels associated with graph elements, preserving their conceptual context. The recognition of patterns, as understood within the conceptual modeling community, remains a subsequent and expert-driven step. In this sense, CMiner does not aim to replace expert interpretation, but rather to support it by systematically exposing candidate structures that may correspond to meaningful conceptual modeling patterns.

8 Related Work

In the last 20 years, several algorithms have been proposed to solve Frequent Subgraph Mining (FSM), spanning different domains that include conceptual graphs [7], typed linked data [40], chemical compound data [19], large disk-based databases [36] and data with differential privacy [37].

We preliminary scanned the literature about FSM algorithms proposed so far and finally identified a set of 11 FSM methods which were selected based on their popularity, similarity to our proposed solution, and the availability of an up-to-date implementation.

Early approaches such as **FSG** [27] and **CloseGraph** [39] are designed for undirected, single-labeled graphs in graph databases. **AGM** [23], is similar to those algorithms but also works for directed graphs. These approaches follow iterative strategies to find frequent subgraphs and do not support multigraphs or template guidance. **gSpan** [38] and **Gaston** [30] improve efficiency with pattern-growth techniques and are also limited to undirected, single-labeled graphs. **gSpan-H** [34] extends this approach to support directed and multi-labeled graphs but still lacks support for multigraphs, templates, and frequency within each single graph. Other algorithms focus on specialized contexts. **MOFA** [5] and **SEuS** [14] handle unlabeled graphs, with MOFA covering directed graphs and SEuS offering limited user interactivity for pattern discovery. **GERM** [3] analyzes evolving graph data but remains constrained to undirected, single-labeled graphs. **STREAMFSM** [31] adapts mining techniques to dynamic graphs by sampling affected regions over time. Distinct from frequency-based methods, **SUBDUE** [20] identifies patterns that compress the graph structure, and is one of the algorithms designed to work on single graphs.

Finally, our proposed method, CMiner, stands out by supporting both directed and undirected multigraphs, multi-labeled

nodes, optional input templates, and the ability to mine patterns from either single graphs or graph databases.

9 Conclusions

In this paper, we present a new frequent subgraph mining algorithm called CMiner. It implements several new features compared to the FSM algorithms in the literature while maintaining competitive performance. CMiner is able to extract pattern frequencies considering a set of graphs or each single graph. It can work on both directed and undirected multigraphs with multiple labels on nodes. It can be guided by input parameters that allow users to search only for patterns having specific characteristics. All these features were motivated by a scenario where FSM must be applied over complex graphs encoding conceptual models, to support the development of pattern catalogues for various conceptual modeling languages. The validation we provided aimed at demonstrating how CMiner addressed the predefined requirements. In this sense, a discussion of the quality of the patterns that can be found by CMiner and a user evaluation based on the usage of the algorithm in a concrete application scenario is out of the scope of this paper, and will be part of the future work.

As future steps, we plan to: *(i)* further optimize the performance of the algorithm; *(ii)* integrate it within a user interface; *(iii)* provide pattern visualization features in line with the notation of the relevant conceptual modeling languages; *(iv)* explore methods for clustering frequent patterns found by the algorithm and sort them by a degree of interest, based on requirements provided by the language experts themselves.

Declaration of generative AI. The authors declare that generative AI tools were used only in some cases for sentence polishing and to check grammatical errors.

Acknowledgments

Giovanni Micale and Alfredo Pulvirenti were funded by the 2024/2026 Research Plan of University of Catania Pia.ce.ri (IMAGINE project). The research of Mattia Fumagalli and Diego Calvanese has been partially supported by the HEU project CyclOps (GA n. 101135513), by the Province of Bolzano and FWF through project OnTeGra (DOI 10.55776/PIN8884924). By the Province of Bolzano and EU through projects ERDF-FESR 1078 CRIMA and ERDF-FESR 1047 AI-Lab, and by MUR through the PRIN project 2022XERWK9 S-PIC4CHU.

References

[1] László Babai. 2016. Graph Isomorphism in Quasipolynomial Time. arXiv preprint arXiv:1512.03547. https://arxiv.org/abs/1512.03547

[2] Dinesh Batra. 2005. Conceptual data modeling patterns: Representation and validation. *Journal of Database Management (JDM)* 16, 2 (2005), 84–106.

[3] Michele Berlingerio, Francesco Bonchi, Björn Bringmann, and Aristides Gionis. 2009. Mining Graph Evolution Rules. In *Machine Learning and Knowledge Discovery in Databases, European Conference, ECML PKDD 2009, Bled, Slovenia, September 7-11, 2009, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 5781)*, Wray L. Buntine, Marko Grobelnik, Dunja Mladenic, and John Shawe-Taylor (Eds.). Springer, 115–130. doi:10.1007/978-3-642-04180-8_25

[4] Pooyan Ramezani Besheli. 2018. The Pattern of Patterns: What is a pattern in conceptual modeling?. In *VMBO*. 99–106.

[5] Christian Borgelt and Michael R. Berthold. 2002. Mining Molecular Fragments: Finding Relevant Substructures of Molecules. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002), 9-12 December 2002, Maebashi City, Japan*. IEEE Computer Society, 51–58. doi:10.1109/ICDM.2002.1183885

[6] Jose Antonio Fernandes de Macedo, Fabio Porto, Sergio Lifschitz, and Philippe Picouet. 2007. A conceptual data model language for the molecular biology domain. In *Twentieth IEEE International Symposium on Computer-Based Medical Systems (CBMS'07)*. IEEE, 231–236.

[7] Adam Faci, Marie-Jeanne Lesot, and Claire Laudy. 2021. cgSpan: Pattern Mining in Conceptual Graphs. In *Artificial Intelligence and Soft Computing*, Leszek Rutkowski, Rafał Scherer, Marcin Korytkowski, Witold Pedrycz, Ryszard Tadeusiewicz, and Jacek M. Zurada (Eds.). Springer, Cham, 149–158.

[8] Ricardo A. Falbo, Giancarlo Guizzardi, Aldo Gangemi, and Valentina Presutti. 2013. Ontology Patterns: Clarifying Concepts and Terminology. In *Proceedings of the 4th International Conference on Ontology and Semantic Web Patterns - Volume 1188* (Sydney, Australia) *(WOP)*. CEUR-WS.org, Aachen, DEU, 14â€“26.

[9] Aamir Farooq and Vadim Zaytsev. 2021. There Is More Than One Way to Zen Your Python. (2021), 68–82. doi:10.1145/3486608.3486909

[10] Philippe Fournier-Viger, Wensheng Gan, Youxi Wu, Mourad Nouioua, Wei Song, Tin Truong, and Hai Duong. 2022. Pattern mining: Current challenges and opportunities. In *International Conference on Database Systems for Advanced Applications*. Springer, 34–49.

[11] Mattia Fumagalli, Tiago Prince Sales, Pedro Paulo F Barcelos, Giovanni Micale, Philipp-Lorenz Glaser, Dominik Bork, Vadim Zaytsev, Diego Calvanese, and Giancarlo Guizzardi. 2025. Mining Frequent Structures in Conceptual Models: M. Fumagalli et al. *Software and systems modeling* (2025), 1–29.

[12] Mattia Fumagalli, Tiago Prince Sales, and Giancarlo Guizzardi. 2022. Pattern Discovery in Conceptual Models Using Frequent Itemset Mining. In *Conceptual Modeling. ER 2022*, Vol. 13607. Springer, 52–62. doi:10.1007/978-3-031-17995-2_4

[13] Aldo Gangemi and Valentina Presutti. 2009. Ontology design patterns. In *Handbook on ontologies*. Springer, 221–243.

[14] Shayan Ghazizadeh and Sudarshan S. Chawathe. 2002. SEuS: Structure Extraction Using Summaries. In *Discovery Science, 5th International Conference, DS 2002, Lübeck, Germany, November 24-26, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2534)*, Steffen Lange, Ken Satoh, and Carl H. Smith (Eds.). Springer, 71–85. doi:10.1007/3-540-36182-0_9

[15] Philipp-Lorenz Glaser, Emanuel Sallinger, and Dominik Bork. 2025. The extended EA ModelSet—a FAIR dataset for researching and reasoning enterprise architecture modeling practices. *Software and Systems Modeling* (2025), 1–19.

[16] Giancarlo Guizzardi. 2014. Ontological Patterns, Anti-patterns and Pattern Languages for Next-Generation Conceptual Modeling. In *ER 2014*. 13–27.

[17] Giancarlo Guizzardi, Gerd Wagner, João Paulo Andrade Almeida, and Renata SS Guizzardi. 2015. Towards Ontological Foundations for Conceptual Modeling: The Unified Foundational Ontology (UFO) Story. *Applied ontology* 10, 3-4 (2015), 259–271.

[18] Jiawei Han and Jian Pei. 2000. Mining frequent patterns by pattern-growth: methodology and implications. *ACM SIGKDD explorations newsletter* 2, 2 (2000), 14–20.

[19] Shuguo Han, Wee Keong Ng, and Yang Yu. 2007. FSP: Frequent Substructure Pattern Mining. In *2007 6th International Conference on Information, Communications & Signal Processing*. 1–5. doi:10.1109/ICICS.2007.4449818

[20] Lawrence B Holder, Diane J Cook, Surnjani Djoko, et al. 1994. Substucture Discovery in the SUBDUE System.. In *KDD workshop*. Washington, DC, USA, 169–180.

[21] John Hunt and John Hunt. 2013. Gang of four design patterns. *Scala Design Patterns: Patterns for Practical Reuse and Design* (2013), 135–136.

[22] M Idrees and MUG Khan. 2015. A review: conceptual data models for biological domain. *JAPS: Journal of Animal & Plant Sciences* 25, 2 (2015).

[23] Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. 2000. An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data. In *Principles of Data Mining and Knowledge Discovery, 4th European Conference, PKDD 2000, Lyon, France, September 13-16, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1910)*, Djamel A. Zighed, Henryk Jan Komorowski, and Jan M. Zytkow (Eds.). Springer, 13–23. doi:10.1007/3-540-45372-5_2

[24] Chuntao Jiang, Frans Coenen, and Michele Zito. 2013. A Survey of Frequent Subgraph Mining Algorithms. *The Knowledge Engineering Review* 28, 1 (2013), 75–105.

[25] Tommi Junttila and Petteri Kaski. 2007. Engineering an efficient canonical labeling tool for large and sparse graphs. In *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics*, David Applegate, Gerth Stølting Brodal, Daniel Panario, and Robert Sedgewick (Eds.). SIAM, 135–149. doi:10.1137/1.9781611972870.13

[26] Tommi Junttila and Petteri Kaski. 2011. Conflict Propagation and Component Recursion for Canonical Labeling. In *Theory and Practice of Algorithms in (Computer) Systems – First International ICST Conference, TAPAS 2011, Rome, Italy, April 18–20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6595)*, Alberto Marchetti-Spaccamela and Michael Segal (Eds.). Springer, 151–162. doi:10.1007/978-3-642-19754-3_16

[27] Michihiro Kuramochi and George Karypis. 2004. An Efficient Algorithm for Discovering Frequent Subgraphs. *IEEE Trans. Knowl. Data Eng.* 16, 9 (2004), 1038–1051. doi:10.1109/TKDE.2004.33

[28] Agnieszka Ławrynowicz, Jedrzej Potoniec, Michał Robaczyk, and Tania Tudorache. 2018. Discovery of Emerging Design Patterns in Ontologies Using Tree Mining. *Semantic web* 9, 4 (2018), 517–544.

[29] Giovanni Micale, Antonio Di Maria, Roberto Grasso, Vincenzo Bonnici, Alfredo Ferro, Dennis Shasha, Rosalba Giugno, and Alfredo Pulvirenti. 2025. MultiGraphMatch: a subgraph matching algorithm for multigraphs. *ACM Transactions on Knowledge Discovery from Data* 19, 5 (2025), 1–36.

[30] Siegfried Nijssen and Joost N. Kok. 2004. The Gaston Tool for Frequent Subgraph Mining. In *Proceedings of the 2nd International Workshop on Graph-Based Tools, GraBaTs 2004, Rome, Italy, October 2, 2004 (Electronic Notes in Theoretical Computer Science, Vol. 127)*, Tom Mens, Andy Schürr, and Gabriele Taentzer (Eds.). Elsevier, 77–87. doi:10.1016/J.ENTCS.2004.12.039

[31] Abhik Ray, Larry Holder, and Sutanay Choudhury. 2014. Frequent Subgraph Discovery in Large Attributed Streaming Graphs. In *Proceedings of the 3rd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications, BigMine 2014, New York City, USA, August 24, 2014 (JMLR Workshop and Conference Proceedings, Vol. 36)*, Wei Fan, Albert Bifet, Qiang Yang, and Philip S. Yu (Eds.). JMLR.org, 166–181. http://proceedings.mlr.press/v36/ray14.html

[32] Tiago Prince Sales, Pedro Paulo F Barcelos, Claudenir M Fonseca, Isadora Valle Souza, Elena Romanenko, César Henrique Bernabé, Luiz Olavo Bonino da Silva Santos, Mattia Fumagalli, Joshua Kritz, João Paulo A Almeida, et al. 2023. A FAIR Catalog of Ontology-driven Conceptual Models. *Data & Knowledge Engineering* 147 (2023), 102210. doi:10.1016/j.datak.2023.102210

[33] Tiago Prince Sales and Giancarlo Guizzardi. 2015. Ontological Anti-patterns: Empirically Uncovered Error-prone Structures in Ontology-driven Conceptual Models. *Data & Knowledge Engineering* 99 (2015), 72–104.

[34] Milind Sangle and Prof. S. A. Bhavsar. 2016. gSpan-H: An Iterative MapReduce Based Frequent Subgraph Mining Algorithm. *International Journal of Advance Research and Innovative Ideas in Education* 2, 5 (2016), 169–177. https://api.semanticscholar.org/CorpusID:65212949

[35] Marigianna Skouradaki, Vasilios Andrikopoulos, Oliver Kopp, and Frank Leymann. 2016. RoSE: Reoccurring Structures Detection in BPMN 2.0 Process Model Collections. In *On the Move to Meaningful Internet Systems: OTM 2016 Conferences*, Christophe Debruyne, Hervé Panetto, Robert Meersman, Tharam Dillon, eva Kühn, Declan O'Sullivan, and Claudio Agostino Ardagna (Eds.). Springer, Cham, 263–281.

[36] Chen Wang, Wei Wang, Jian Pei, Yongtai Zhu, and Baile Shi. 2004. Scalable Mining of Large Disk-Based Graph Databases. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Seattle, WA, USA) *(KDD)*. ACM, New York, NY, USA, 316â€“325. doi:10.1145/1014052.1014088

[37] Jiangna Xing and Xuebin Ma. 2021. DP-gSpan: A Pattern Growth-based Differentially Private Frequent Subgraph Mining Algorithm. In *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. 397–404. doi:10.1109/TrustCom53373.2021.00067

[38] Xifeng Yan and Jiawei Han. 2002. gSpan: Graph-Based Substructure Pattern Mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002), 9-12 December 2002, Maebashi City, Japan*. IEEE Computer Society, 721–724. doi:10.1109/ICDM.2002.1184038

[39] Xifeng Yan and Jiawei Han. 2003. CloseGraph: mining closed frequent graph patterns. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 24 - 27, 2003*, Lise Getoor, Ted E. Senator, Pedro M. Domingos, and Christos Faloutsos (Eds.). ACM, 286–295. doi:10.1145/956750.956784

[40] Xiang Zhang, Cuifang Zhao, Peng Wang, and Fengbo Zhou. 2012. Mining Link Patterns in Linked Data. In *Web-Age Information Management*, Hong Gao, Lipyeow Lim, Wei Wang, Chuan Li, and Lei Chen (Eds.). Springer, Berlin, Heidelberg, 83–94.

A Proofs

Lemma A.1. *Let $G = (V, E)$ be a graph with maximum node degree Δ. Then the number of connected induced subgraphs of size M is at most $O(|V| \cdot (c\Delta)^{M-1})$, for some constant c.*

Proof. Fix a size $k \leq M$ and a root node $r \in V$. A connected induced set of k nodes containing r can be constructed incrementally, by adding one node at a time from the boundary of the current set. Since each node has degree at most Δ, when the partial set has size t the boundary has at most Δt nodes. Hence the number of possible growth sequences of length $k-1$ is bounded by

$$\prod_{t=1}^{k-1} \Delta t \;=\; \Delta^{k-1}(k-1)!.$$

Different sequences may yield the same set, so this is also an upper bound on the number of distinct connected k-sets rooted at r. To control the factorial term, observe first that

$$(k-1)! \;=\; \Big((k-1)!\Big)^{\frac{k-1}{k-1}}.$$

Since the sequence $a_t := (t!)^{1/t}$ is increasing in t, for $k \leq M$ we then have

$$(k-1)! \;\leq\; \Big((M-1)!\Big)^{\frac{k-1}{M-1}} = \Big(\big((M-1)!\big)^{1/(M-1)}\Big)^{k-1}.$$

Setting $c := \big((M-1)!\big)^{1/(M-1)}$, which depends only on M, we obtain

$$\Delta^{k-1}(k-1)! \;\leq\; (c\Delta)^{k-1}.$$

Thus, the number of connected k-nodes sets rooted at r is at most $(c\Delta)^{k-1}$, and multiplying by $|V|$ for all choices of r yields $O\big(|V|\,(c\Delta)^{k-1}\big)$ sets. Since this bound increases with k, the total number of connected induced subgraphs of size up to M is $O\big(|V|\,(c\Delta)^{M-1}\big)$. □

Lemma A.2. *Let $c := \big((M-1)!\big)^{1/(M-1)}$. Then $c = \Theta(M)$.*

Proof. Set $n := M-1$, so that $c = (n!)^{1/n}$. Using Stirling's approximation,

$$n! \;\sim\; \sqrt{2\pi n}\left(\tfrac{n}{e}\right)^n \qquad \text{as } n \to \infty.$$

Taking n-th roots we obtain

$$c = (n!)^{1/n} \;\sim\; \Big(\sqrt{2\pi n}\Big)^{1/n} \cdot \frac{n}{e}.$$

The factor $\Big(\sqrt{2\pi n}\Big)^{1/n}$ tends to 1 because $\log(\sqrt{2\pi n})/n \to 0$. Therefore

$$c \;\sim\; \frac{n}{e} \;=\; \frac{M-1}{e}.$$

This shows that c grows linearly in M, i.e. $c = \Theta(M)$. □