Mining Tractable Sets of Graph Patterns with the Minimum Description Length Principle

Francesco Bariatti

DOCTORAT BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par
Francesco BARIATTI

« Mining Tractable Sets of Graph Patterns with the Minimum Description Length Principle »

Thèse présentée et soutenue à IRISA Rennes, le 23 Novembre 2021
Unité de recherche : UMR 6074 - IRISA

Rapporteurs avant soutenance :

Bruno CRÉMILLEUX	Professeur à l'Université de Caen Normandie
Jilles VREEKEN	Faculty au CISPA Helmholtz Center for Information Security Senior researcher au Max Planck Institute for Informatics Professeur à l'Université du Saarland (Allemagne)

Composition du Jury :

Président :	Alexandre TERMIER	Professeur à l'Université de Rennes 1
Examinateurs :	Matthijs van LEEUWEN	Maitre de conférences à l'Université de Leiden (Pays Bas)
	Nathalie PERNELLE	Professeur à l'Université Sorbonne Paris Nord
	Arnaud SOULET	Maitre de conférences à l'Université de Tours
	Alexandre TERMIER	Professeur à l'Université de Rennes 1
Dir. de thèse :	Sébastien FERRÉ	Professeur à l'Université de Rennes 1
Co-encadrante :	Peggy CELLIER	Maitre de conférence à l'INSA Rennes

Acknowledgements

First and foremost, I would like to thank my supervisors, Peggy and Sébastien, for their incredible work during the course of this PhD. They are the main reason that made this PhD so amazing for me. From a research point of view, they were always available to help me and give me feedback on my work ; and from a career point of view, they gave me important advice for my future as a researcher, and are still doing so at the time of writing these acknowledgements. Finally, I want to thank them for being great colleagues, and for all the wonderful moments spent chatting together during coffee breaks or at the beginning of our meetings.

I would like to thank my family, and especially my partner Léna, who helped me trough all the stressful moments that occurred during the course of this PhD. In particular, I want to thank her for the time that she spent reassuring me after I made a dozen re-takes of the same ten-minute video for my first presentation at an international conference, and for her courage for agreeing to listen to a rehearsal of my PhD defense late in the evening, even though she is not at all a computer scientist and she did not understand most of it. A PhD is not always easy, and her presence during the difficult moments was crucial.

I would like to thank all my colleagues for the time spent together. I experienced a great work environment, full of interesting discussions, board games, and good moments in general.

I would like to thank all the members of the defense committee for accepting to spend their time reading this thesis, participating in the PhD defense, and just taking interest in my work. And also for their kind comments, interesting (and sometimes very challenging) questions, and important feedback during the defense.

And finally, I would like to thank you, the reader, for taking interest in this thesis. I hope that you find it as interesting as I found it while writing, and that you have a good time reading it.

Francesco

Résumé en Français

De nos jours, de nombreux domaines produisent de grandes quantités de données structurées, représentant des informations spécifiques au domaine. En particulier, il est courant de trouver des données structurées sous la forme d'un ensemble d'entités, reliées entre elles par des relations. Par exemple, en chimie et en biologie, les molécules peuvent être exprimées comme des atomes reliés par des liaisons ; en linguistique, les phrases peuvent être exprimées comme des mots reliés par des relations de dépendance ; dans le web sémantique, les informations relatives à un domaine peuvent être exprimées sous forme d'entités nommées reliées par des relations sémantiques. Ces données sont souvent représentées sous forme de graphes : des structures de données où les "sommets" (les entités) sont interconnectés par des "arêtes" (les relations). Les sommets et arêtes peuvent aussi être étiquetés, afin de leur attacher des attributs qui précisent leur propriétés et leur donner une signification sémantique. Ces données peuvent révéler de la connaissance utile à l'utilisateur qui les analyse. Par exemple, en biologie, cela pourrait intéresser l'utilisateur de trouver quelles sont les caractéristiques communes d'une famille de molécules actives contre une certaine maladie, afin de produire un nouveau médicament [BB02] ; en linguistique, la structure de différents textes peut être comparée afin de mettre en évidence les différences dans le style d'écriture.

Cependant, l'analyse de données par un humain devient de plus en plus difficile à mesure que la taille du jeu de données augmente. Dans la pratique, il n'est pas rare de trouver des jeux de données dont les graphes comportent des millions ou des milliards de sommets reliés par autant d'arêtes. De plus, contrairement à d'autres types de données (par exemple les séquences), les graphes n'ont pas de représentation canonique évidente, ce qui signifie qu'un même graphe peut être représenté de plusieurs manières, équivalentes, rendant ainsi son interprétation plus complexe. Pour toutes ces raisons, plus la taille des données est importante, plus la tâche devient difficile pour un utilisateur humain. Afin d'aider les utilisateurs, des approches automatisées sont nécessaires pour rendre les données plus faciles à traiter.

Les approches de *fouille de motifs* [AIS93] aident l'utilisateur à s'attaquer à cette tâche en extrayant des structures locales —appelées *motifs*— à partir des données. Ainsi, au

lieu d'avoir à analyser l'intégralité des données, l'utilisateur peut se concentrer sur ces petites structures et déduire de la connaissance à partir de celles-ci. Comme les motifs représentent de petites parties des données, il est plus facile de les analyser que d'analyser l'intégralité des données. Des nombreuses approches de fouille de motifs ont été proposées pour traiter des données et des motifs de type graphe [RP15 ; JCZ13]. Cependant, un problème courant dans la fouille de motifs est le problème dit de *l'explosion du nombre de motifs*. Même sur de petits jeux de données, l'ensemble des motifs extraits par les approches de fouille de motifs à tendance à être de très grande taille. Puisqu'un motif est une sous-structure présente dans les données, il peut potentiellement y avoir autant de motifs qu'il y a de sous-structures dans les données. Dans ce cas, la fouille de motifs n'est d'aucune utilité pour l'utilisateur, car l'analyse de la grande quantité de motifs extraits devient une tâche aussi difficile que celle de l'analyse des données initiales.

Afin de réduire le nombre de motifs extraits par les approches de fouille de motifs, un critère de sélection classiquement appliqué est de n'autoriser que les motifs qui apparaissent "fréquemment" dans les données [Agr+96]. La logique est la suivante : si un motif apparaît très fréquemment, il est alors *caractéristique* des données, et en tant que tel, il peut s'avérer utile pour l'analyste qui cherche à extraire de la connaissance. Cependant, ce simple critère de fréquence n'est pas suffisant pour réduire efficacement le nombre de motifs, car avec des seuils de fréquence assez bas, le nombre de motifs extraits reste trop important. C'est pourquoi d'autres approches ont été proposées pour résoudre ce problème : l'utilisation de représentations condensées pour réduire le nombre de motifs affichés à l'utilisateur [Pas+99 ; Bay98] ; l'intégration de contraintes dans le processus de fouille [Ng+98] ; l'utilisation de mesures d'intérêt autres que la fréquence pour sélectionner les motifs [GH06] ; et l'échantillonnage aléatoire de l'espace des motifs [To96 ; DL17]. Un autre problème qui accompagne fréquemment l'explosion du nombre de motifs est que l'ensemble des motifs extraits peut contenir de nombreuses redondances, qui en l'agrandissant augmentent la charge de travail de l'analyste, sans pour autant apporter de nouvelles informations. Idéalement, afin d'éviter cette redondance, le choix d'un motif donné ne devrait pas seulement prendre en compte le motif lui-même, mais aussi les autres motifs sélectionnés. En suivant cette idée, des approches ont été proposées qui évaluent des *ensembles de motifs* [RZ07].

Plus récemment, des approches ont été proposées qui utilisent le principe *Minimum Description Length* (MDL) [Grü07 ; Ris78] pour générer et sélectionner des ensembles de motifs suffisamment *petits* pour permettre une analyse humaine et suffisamment *descriptifs*

des données pour permettre d'en extraire de la connaissance significative. Le principe MDL provient du domaine de la théorie de l'information et est souvent résumé par la formule suivante : "le modèle qui décrit le mieux les données est celui qui les compresse le plus", ce qui signifie qu'un modèle adapté aux données devrait permettre de les décrire avec une quantité minimale d'informations par rapport à un modèle qui n'est pas adapté. Dans le domaine de la fouille de motifs, le principe MDL a été appliqué au problème de la sélection de motifs en traitant les ensembles de motifs comme des "modèles" qui sont utilisés pour encoder les données. Ainsi, le principe MDL permet d'attribuer à un ensemble de motifs une valeur numérique exprimant si l'ensemble est "descriptif" par rapport aux données, et donc permettant de comparer différents ensembles de motifs. Il a été montré que les approches utilisant le principe MDL sont capables de sélectionner des ensembles de motifs qui sont descriptifs des données, mais aussi suffisamment petits pour permettre une analyse humaine [Gal20]. Le principe MDL a été utilisé pour des tâches de fouille de motifs sur de nombreux types de données : données transactionnelles [VLS11 ; SV12], bases de données relationnelles [KS09], séquences [TV12 ; Gal+19], et matrices [FL20]. Quelques approches existent également pour les graphes [Kou+15 ; CH93 ; Bel+20 ; BR20].

Dans cette thèse, nous proposons des approches qui utilisent le principe MDL afin de générer et sélectionner des *petits ensembles* de motifs *descriptifs* de type graphe à partir de données de type graphe. Ceci afin d'aider les analystes humains à extraire de la connaissance significative des données. La fouille de motifs dans les graphes présente non seulement les défis habituels de la fouille de motifs —tels qu'un grand espace de recherche qui nécessite une stratégie d'exploration efficace— mais présente également des défis spécifiques dus à la nature des graphes. En premier lieu, détecter si un motif est présent dans les données (et combien de fois il est présent) est un problème NP-complet [For96]. Dans la pratique, la plupart du temps cela n'a pas beaucoup d'impact, car si les données sont suffisamment diverses, de nombreuses occurrences possibles peuvent être exclues. Cependant, dans certains cas cela pose un sérieux problème pour la vitesse des approches de fouille. Deuxièmement, les données de type graphe ont une composante structurelle importante. Savoir qu'un motif est *présent* dans les données n'est pas suffisant. Savoir *comment le motif se connecte au reste des données* révèle également de la connaissance importante sur les données. Conserver ces informations exactes sur les liens entre les occurences de motif est particulièrement important pour une approche basée sur le principe MDL, car l'application de ce principe exige que les informations soient encodées sans aucune perte. Un autre défi réside dans le fait que les graphes peuvent se présenter sous de nombreuses formes

(étiquetés/non étiquetés, dirigés/non dirigés, simples/multigraphes/hypergraphes), ce qui ajoute à la complexité globale de la tâche. Il s'agit d'un défi particulièrement significatif lors de la conception d'approches génériques, qui font le moins d'hypothèses possible sur les données en entrée. En dernier, pour que les approches de fouille de motifs soient les plus faciles possible à utiliser, il est souhaitable qu'elles limitent le nombre de paramètres qui doivent être réglés lors de leur exécution.

Contributions

Les contributions de cette thèse sont les suivantes :

1. Nous instancions le principe MDL dans un contexte de fouille de motifs de graphes. Nous proposons des mesures —appelées *longueurs de description* dans le contexte MDL— pour évaluer des ensembles de motifs de type graphe extraits de données de type graphe. Nous montrons, au travers d'expériences, que les ensembles de motifs sélectionnés selon ces mesures sont descriptifs des données et permettent d'en extraire de la connaissance.
2. Nous introduisons la notion de *ports*, qui permet de décrire les données comme une composition d'occurrences de motifs. Les ports permettent d'encoder les données avec un ensemble d'occurrences de motifs sans aucune perte d'information. Nous montrons également par des expériences que les ports permettent une analyse pertinente tant du point de vue des données que des motifs.
3. Nous proposons des approches pour extraire *un petit ensemble de taille humaine* de motifs *descriptifs* à partir de données de type graphe, en utilisant notre concept de ports et nos définitions MDL. Nous proposons des algorithmes heuristiques pour chacune de ces approches, permettant de produire des résultats en un temps raisonnable.

   (a) La première approche proposée effectue une sélection sur un ensemble de motifs donné en entrée de la méthode. Nous montrons qu'elle est capable de réduire de grands ensembles de motifs à de plus petites versions de taille humaine.

   (b) La deuxième approche proposée *entrelace la génération de motifs avec leur sélection*. Cela libère l'approche de toute dépendance à l'égard d'approches externes de génération de motifs, et améliore considérablement les performances

par rapport à l'approche précédente. L'approche devient également *sans paramètres*, puisque seules les données sont nécessaires pour générer et sélectionner des ensembles de motifs pertinents. De plus, cette approche est *anytime*, car elle peut produire un ensemble de motifs à n'importe quel moment de son exécution.

(c) Enfin, nous proposons une troisième approche qui étend ces techniques aux *graphes de connaissance*, utilisés dans le web sémantique. Nous discutons des différences entre les graphes de connaissances et les graphes habituellement utilisés dans la fouille de motifs, et les adaptations nécessaires au bon fonctionnement de l'approche. Cette approche peut également être facilement étendue aux *hypergraphes*, une généralisation du concept de graphe.

4. Nous proposons un outil pour *visualiser* les résultats de nos approches de manière *interactive*, permettant à l'utilisateur de les manipuler afin de mieux les comprendre.
5. Nous évaluons toutes nos contributions expérimentalement sur des jeux de données provenant de différents domaines.

Table of Contents

Chapter 1

Introduction

Nowadays, large quantities of structured data can be found in many fields, encoding information about their respective domains. In particular, it is common to find data structured as a set of entities, connected to each other via relationships. For instance, in chemistry and biology molecules can be expressed as atoms connected by bonds; in linguistics sentences can be expressed as words connected by dependency relationships; in the semantic web the information about a domain can be expressed as real-world entities connected by semantic relationships. Such data is often represented as *graphs*, data structures where "vertices" (the entities) are inter-connected via "edges" (the relationships). Labels can also be added to the vertices and edges to express their attributes and give them semantic meaning. Such data can reveal useful knowledge to the user that analyzes it. For example in biology the user may be interested in finding what are the common features in a family of molecules that are active against a certain disease, in order to produce a new drug [BB02]; and in linguistics the structure of different texts can be compared in order to highlight differences in style.

However, human analysis becomes increasingly more difficult as the size of a dataset increases. In practice, it is not unusual to find datasets where the graphs have millions or billions of vertices connected by as many edges, or large collections of hundreds of graphs. Also, contrary to other types of data (e.g. sequences) graphs do not have a natural canonical representation, meaning that a same graph can be represented in several different —and equivalent— ways, which makes the task more complex. Because of all these reasons, as the data gets bigger, the task becomes more daunting for human analysts. In order to help humans, automated approaches are needed that make the data more tractable.

Pattern mining approaches [AIS93] help the user tackle the task by extracting local structures —called *patterns*— from the data. Therefore, instead of the user having to analyze all of the data at once, they can focus on those smaller structures and infer knowledge from them. Since the patterns represent small parts of the data, the task of analyzing them

is easier than of analyzing the whole of the data. Many pattern mining approaches have been proposed to tackle graph data and patterns [RP15; JCZ13]. However, a well-known problem in pattern mining is the so-called problem of *pattern explosion*. Even on small datasets, the set of patterns that are extracted by pattern mining approaches can be very large in size. Since a pattern is a structure that is present in the data, there can potentially be as many patterns as there are sub-structures of the data. In those circumstances, pattern mining does not help the user, as analyzing the large amount of patterns produced is no less difficult a task than analyzing the initial data.

In order to reduce the number of extracted patterns , a classic selection criterion that is applied is to only allow patterns that appear "frequently enough" in the data, so as to leave out outliers [Agr+96]. The rationale being that patterns that appear very frequently are *characteristic* of the data, and as such they should be a useful tool for the analysts seeking to extract knowledge. But this simple frequency criterion is not sufficient, as it can still allow too many patterns for a low enough frequency threshold. Therefore other approaches have been proposed that tackle this problem by: using condensed representations to summarize the patterns to the user [Pas+99; Bay98]; integrating constraints in the mining process [Ng+98]; using different interest measures than the frequency to select patterns [GH06]; and random sampling of the pattern space [To96; DL17]. Another problem that frequently accompanies the large amount of extracted patterns is that the set of extracted patterns may contain many redundancies, which enlarge it and thus increase the workload of the analyst without bringing much new information. Ideally, in order to avoid this redundancy, the choice of selecting a given pattern should not only take into account the pattern itself, but also the other patterns that are selected. Therefore, approaches have been proposed that instead of selecting each pattern individually, evaluate *sets of patterns* as a whole [RZ07].

More recently, approaches have been proposed that use the *Minimum Description Length* (MDL) principle [Grü07; Ris78] in order to generate and select sets of patterns that are *small* enough so as to allow human analysis and *descriptive* enough of the data so as to allow the extraction of meaningful knowledge from it. The MDL principle comes from the domain of information theory and is often summarized as "the model that describes the data the best is the one that compresses the data the best", meaning that a model that is suited for some data should allow to describe it with a minimum amount of information, w.r.t. a model that is not suited for it. In the domain of pattern mining, the MDL principle has been applied to the problem of pattern selection by treating sets of

patterns as "models" that can be used to describe the data. In this way, the principle expresses a way to numerically evaluate the "descriptiveness" of a set of patterns w.r.t. some data, and thus to compare different sets of patterns. It has been shown that approaches using the MDL principle are able to select sets of patterns that are descriptive of the data but also small enough that human analysis is possible [Gal20]. The MDL principle has been used in pattern mining tasks on many kinds of data: transactional data [VLS11; SV12], relational databases [KS09], sequences [TV12; Gal+19], and matrices [FL20]. A few approaches also exist for graphs [Kou+15; CH93; Bel+20; BR20].

In this thesis we propose approaches that use the Minimum Description Length principle in order to generate and select *small and descriptive* sets of graph patterns from graph data, so as to help human analysts retrieve meaningful knowledge from such data. Pattern mining on graphs not only presents the standard challenges of pattern mining —such as a large search space that requires an efficient exploration strategy— but also presents specific challenges due to the nature of graphs. First of all, detecting if a pattern is present in the data (and how many occurrences it has) is an NP-complete problem [For96]. In practice, most of the time this is not very impactful, because if the data is diverse enough, many possible occurrences can be ruled out. However there can be some edge cases in which this presents a serious problem to the runtime of the pattern mining approaches. Second, graph data has an important structural component. Knowing that a pattern *is present* in the data is not enough. Knowing *how the pattern connects* to the rest of the data can also reveal important knowledge about the data. Keeping this exact information about pattern connections is especially important for a MDL-based approach, as applying the MDL principle requires that information is encoded without any loss. Another challenge is that graphs can appear in many shapes and forms (labeled/unlabeled, directed/undirected, simple/multigraphs/hypergraphs), which add to the overall combinatorial complexity of the task. This is particularly a challenge when designing generic approaches that make as little assumptions about the input data as possible. Finally, for pattern mining approaches to be the easily usable by users, it is desirable that they limit the number of parameters that need to be tuned when executing them.

1.1 Contributions

The contributions of this thesis are the following:

1. We instantiate the MDL principle in a graph pattern mining context. We propose

measures —called *description lengths* in a MDL context— to evaluate sets of graph patterns extracted from graph data. We show through experiments that pattern sets selected following these measures are descriptive of the data and highlight knowledge about it.

2. We introduce the notion of *ports*, which allows to describe graph data as a composition of graph pattern occurrences. Ports allow to encode the data using a set of pattern occurrences without any loss of information. We also show through experiments that they allow for an interesting analysis both from the point of view of the data and of the patterns.

3. We propose approaches to extract a *small, human-sized* set of *descriptive* graph patterns from graph data, using our concept of ports and MDL definitions. We propose heuristic algorithms for all these approaches, allowing them to find results in reasonable time.

   (a) The first proposed approach performs a *selection* on an already-extracted set of patterns. We show that it is able to reduce large sets of patterns to smaller, human-sized versions.

   (b) The second proposed approach *interleaves the pattern generation with the selection*. This frees the approach from any dependency on external pattern generation approaches, and greatly enhances the performances w.r.t. the previous approach. It also makes the approach *parameterless*, as only the data is needed for it to generate and select relevant patterns. Additionally, this approach is *anytime*, as it can yield a set of patterns at any point during its execution.

   (c) Finally, we propose a third approach that extends those techniques to *knowledge graphs*, used in the semantic web. We discuss the differences between knowledge graphs and the graphs usually used in graph pattern mining, and the adaptations that are needed for it to run successfully. This approach can also easily be extended to *hypergraphs*, a generalization of the concept of graphs.

4. We propose a tool for *visualizing* the results of our approaches in an *interactive* manner, allowing the user to manipulate them so as to better understand them.

5. We evaluate all our contributions experimentally on real-life datasets from different domains.

1.2 Thesis Outline

The remaining of this thesis is organised as follows. **Chapter 2** presents some concepts that are used extensively in the rest of the thesis. It formally defines the graph data structure and presents the associated notions of isomorphism, knowledge graphs, and hypergraphs. It also presents in more details the Minimum Description Length principle. In **Chapter 3** we present a state of the art of pattern mining and particularly graph pattern mining, the problem of pattern explosion, and MDL-based pattern mining. **Chapter 4** presents our first proposal —called GraphMDL— for selecting a small and descriptive subset of patterns from a given (large) set of patterns. For that we define a MDL measure that is suited for graph pattern mining and we show its effectiveness at this task. This chapter also presents the notion of *ports* that is used to encode the data with a set of patterns without loss of information. We also present in Section 4.5 GraphMDL Visualizer, an *interactive visualization tool* capable of visualizing the results of our approaches, so as to help the user better understand the extracted patterns and how they appear in the data. In **Chapter 5** we present a new *anytime and parameterless* approach —called GraphMDL+— that tackles some drawbacks of GraphMDL. In particular, in this chapter we introduce a way to generate patterns that are suited for the MDL-based selection. As a result this new approach is much faster than the previous one and generates patterns of higher quality. Because this approach can generate its own patterns, it does not depend on an external pattern generation approach, giving it more freedom about the patterns it generates. In **Chapter 6** we extend GraphMDL+ to handle knowledge graphs. In order to implement this extension —called KG-MDL— we discuss the differences between knowledge graphs and usual pattern mining graphs, and the challenges that result from these differences. We evaluate this approach experimentally on real-life knowledge graphs, showing that the generated patterns can give useful insights about both the knowledge graphs and the underlying schema that describes its structure. Finally, in **Chapter 7** we summarize our contributions, discuss them and present perspectives and research axis for future work.

Chapter 2

Background Knowledge

In this chapter we present notions that are extensively used in the rest of the thesis. Section 2.1 presents the concept of graphs —the type of data used throughout this thesis— and its associated notions. Section 2.2 presents the Minimum Description Length (MDL) principle. This principle comes from the domain of information theory and it is that we use in order to guide the search for relevant patterns from the data.

2.1 Graphs

Graphs are data structures that allow to represent a set of entities (the "vertices") and the relations that connect them (the "edges"). Graphs are generic enough to be able to represent a wide range of real-life data: social networks, cristalline structures, sentences in text, source code structure, and molecules, to cite only a few. Labeled graphs allow to represent information even more precisely by associating labels to those entities and relations, such that the graph not only represents a structure, but also properties of the elements in the structure. In this section we first present the usual definition of labeled graphs found in the literature, then we explore its limitations and give a new definition that we use throughout this thesis. Then, we present two concepts related to graphs —isomorphism and automorphism— which have a central role in graph applications compared to applications using simpler data structures such as sets or sequences. Then, we present the concept of Knowledge Graphs (KGs), used in the Semantic Web, and the differences that they have with usual graphs. Finally, we present the concept of hypergraphs, a generalization of the concept of graphs.

2.1.1 Labeled Graphs

In the literature, labeled graphs are usually defined as follows:

Definition 1 (Labeled graphs, usual definition) *A* labeled graph $G = (V, E, l)$ *over*

a label set $\mathcal{L}$ is a data structure composed of a set of vertices V*, a set of* edges $E \subseteq V \times V$*, and a* labeling function $l \in (V \cup E) \to \mathcal{L}$ *that associates a label to vertices and edges.*
G is undirected *if E is symmetric (i.e. $(u, v) \in E \iff (v, u) \in E$) and* directed *otherwise.*
G is simple *if E is irreflexive (i.e. $(u, u) \notin E$, no self-loops).*

While this definition allows to express a wide range of data, it has some limitations. First, vertices have exactly one label. While it is possible to have a vertex without labels by creating a special "no label" label, it is certainly not possible to easily express multiple attributes of a vertex. This forbids representations such as a family tree in which a person can be both female and young, unless one creates the label "female and young" and similar labels for any other combination of attributes. Second, this definition forbids the presence of multiple edges between a pair of vertices (graphs where this happen are usually called *multigraphs*). This limits the expressivity of the graph, forbidding multiple relations between entities, such as a social graph in which some people are both friends and colleagues. For this reason we use in this thesis a more general definition, which allows vertices to have any number of labels, and pairs of vertices to be connected by any number of edges. Additionally, this definition is easily extended to hypergraphs (presented in Section 2.1.4), which are graphs whose "edges" (called hyper-edges) can connect any number of vertices.

Definition 2 (Labeled graphs) *A* labeled graph $G = (V, V_{\mathcal{L}}, E_{\mathcal{L}})$ *over a set of* label symbols $\mathcal{L}$ *is a data structure composed of a set of* vertices V*, a set of* vertex labels $V_{\mathcal{L}} \subseteq V \times \mathcal{L}$*, and a set of* edge labels —*or* labeled edges— $E_{\mathcal{L}} \subseteq V \times V \times \mathcal{L}$*.*

For the sake of conciseness, in this thesis we call elements of $V_{\mathcal{L}}$ and $E_{\mathcal{L}}$ *labels*, and labeled graphs just *graphs*. We call elements of $\mathcal{L}$ *label symbols*, and they correspond to all the possible pieces of information that can be associated to vertices and edges. Conversely, labels correspond to occurrences of label symbols.

Definition 3 (Directed graphs, multigraphs) *Let $G = (V, V_{\mathcal{L}}, E_{\mathcal{L}})$ be a labeled graph.*
G is undirected *if $(u, v, l) \in E_{\mathcal{L}} \iff (v, u, l) \in E_{\mathcal{L}}$, otherwise it is* directed*.*
G is a multigraph *if there exist $(u, v, l) \in E_{\mathcal{L}}, (u, v, l') \in E_{\mathcal{L}}$ such that $l \neq l'$.*
G is simple *if it is not a multigraph and for all $v \in V$, $(v, v, l) \notin E_{\mathcal{L}}$ (i.e. no loops exist).*

Fig. 2.1, 2.2, and 2.3 show example graphs following these definitions. Fig. 2.1 shows an undirected simple graph with 8 vertices, 9 vertex labels, and 7 edge labels, from a set

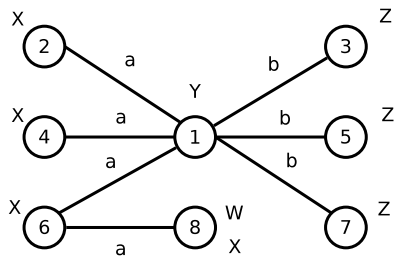


Figure 2.1 – A labeled undirected simple graph

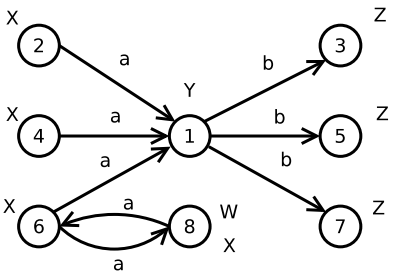


Figure 2.2 – A labeled directed simple graph

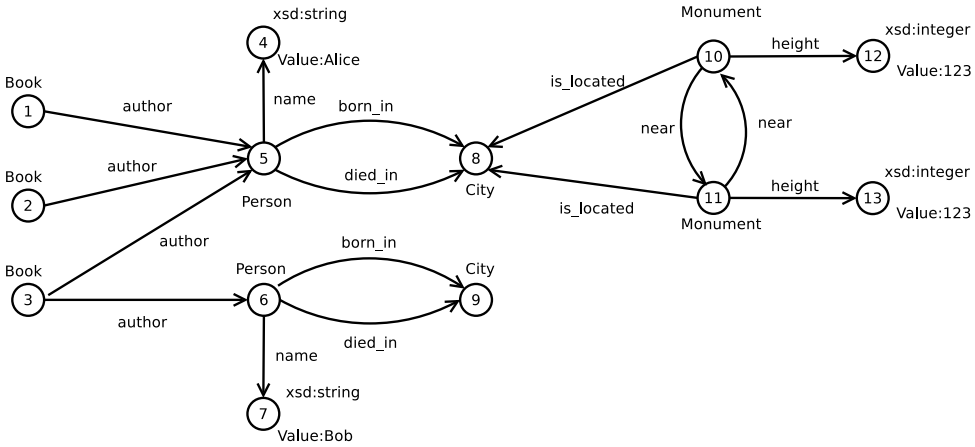


Figure 2.3 – A labeled directed multigraph.

of 6 label symbols $\{W, X, Y, Z, a, b\}$ (4 vertex label symbols and 2 edge label symbols). Fig. 2.2 shows a similar example of a directed simple graph with 8 vertices, 9 vertex labels, and 8 edge labels, from the same set of 6 label symbols. Fig. 2.3 shows a directed multigraph with 13 vertices, 17 vertex labels, and 16 edge labels, from a set of 16 label symbols (9 vertex label symbols and 7 edge label symbols). This last graph is a multigraph because there are multiple edges starting from vertex 5 and ending in vertex 8 (and from vertex 6 to vertex 9).

Note that in the domain of graph mining, most approaches are developed for simple undirected graphs, only few of them having adaptations to simple directed graphs as well [JCZ13]. On the other hand, in the domain of knowledge graphs (see Section 2.1.3) directed multigraphs are considered. The works presented in this thesis consider both directed and undirected simple graphs in Chapters 4 and 5, and multigraphs in Chapter 6.

In graph pattern mining, we find two fundamental notions: the notion of *pattern* and the

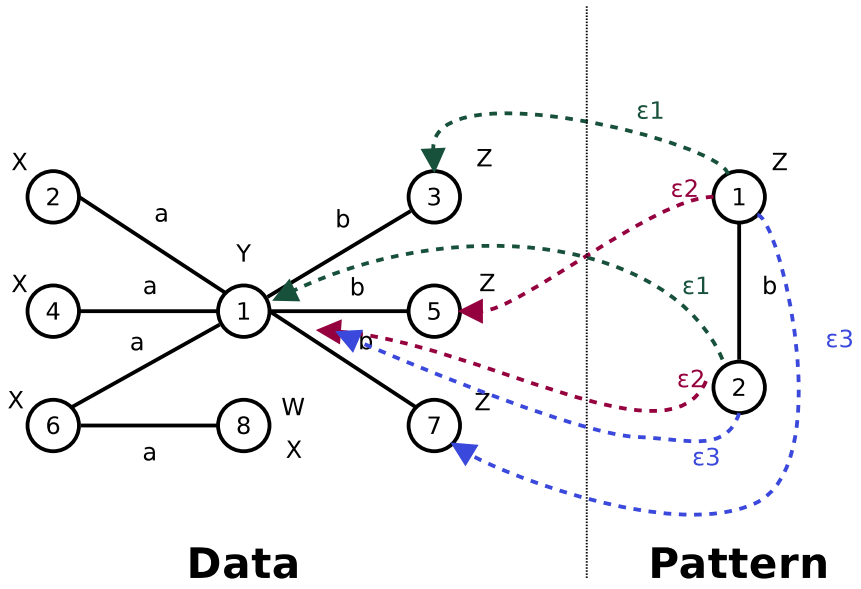


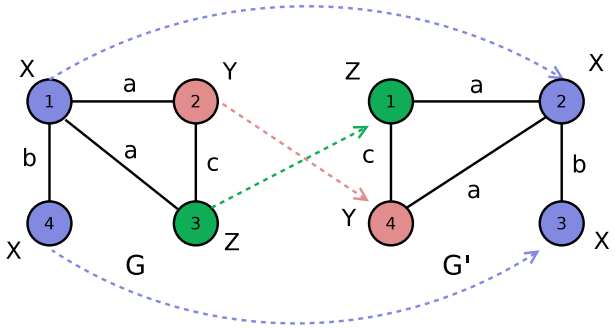
Figure 2.4 – The three embeddings of a pattern (right) in the graph of Fig. 2.1 (left).

notion of *pattern embedding*, or *pattern occurrence*.

Definition 4 (Pattern embeddings) *Let G^P (the "pattern") and G^D (the "data") be labeled graphs. An* embedding *—or* occurrence*— of G^P in G^D is an injective function $\varepsilon \in V^P \to V^D$ such that:*
(1) $(v, l) \in V^P_{\mathcal{L}} \implies (\varepsilon(v), l) \in V^D_{\mathcal{L}}$
(2) $(v_1, v_2, l) \in E^P_{\mathcal{L}} \implies (\varepsilon(v_1), \varepsilon(v_2), l) \in E^D_{\mathcal{L}}$.

Informally, a pattern embedding maps the vertices of the "pattern" to some of the vertices of the "data". It conveys the information that the structure of the pattern can be found within the data, at the specific region defined by those vertices. Of course, the data can be more complex than the pattern, having additional vertices and edges around —or within— the embedding. But the embedding tells us that in that region of the data we can find *at least the whole structure* of the pattern. As an illustration, Fig. 2.4 shows the three embeddings $\varepsilon_1, \varepsilon_2, \varepsilon_3$ of a two-vertices graph pattern (on the right) into the graph of Fig. 2.1 (on the left).

In graph theory, this definition of an embedding is called a *subgraph isomorphism*. We present the notion of isomorphism hereafter.



v	1	2	3	4
$f(v)$	2	4	1	3

Figure 2.5 – Two isomorphic graphs G and G' and the isomorphism that relates them.

2.1.2 Isomorphisms and Automorphisms

The notions of isomorphism and automorphism are core notions when dealing with graphs, which differentiate graphs from other simpler data structures, such as sets or sequences. When dealing with graphs, it is not trivial to check whether two given graphs that seem different are actually different structures, or merely different representations of the same structure. This is not the case with sets or sequences, where a simple lexicographic comparison suffices to decide.

Definition 5 (Graph isomorphism) *Let* $G = (V, V_{\mathcal{L}}, E_{\mathcal{L}})$ *and* $G' = (V', V'_{\mathcal{L}}, E'_{\mathcal{L}})$ *be graphs. An* isomorphism *between* G *and* G' *is a bijective function* $f \in V \to V'$ *such that:*
(1) $(v, l) \in V_{\mathcal{L}} \iff (f(v), l) \in V'_{\mathcal{L}}$
(2) $(u, v, l) \in E_{\mathcal{L}} \iff (f(u), f(v), l) \in E'_{\mathcal{L}}$
If such an isomorphism exists, G *and* G' *are* isomorphic.

Informally, two graphs are isomorphic if they have the same structure and they only differ in the way that their vertices are identified, e.g. the vertex called 1 in the first graph is called 2 in the second one and so on. An isomorphism maps each vertex of the first graph to the corresponding vertex of the second graph. It makes the fact that the two graphs have the same structure visible by giving a mapping that, when applied, transforms the first graph in the second. Fig. 2.5 shows two isomorphic graphs and the isomorphism that relates them. Note that there could be many isomorphisms between two given graphs. Pattern embeddings are a special case of isomorphism, called *subgraph isomorphism*, where the pattern is isomorphic to a part of the data.

Detecting graph isomorphisms is not an easy task. Interestingly, no known polynomial-time solution exists for the general case, but the problem is not known to be NP-complete either [For96]. Polynomial-time solutions exist for certain classes of graphs, e.g. trees

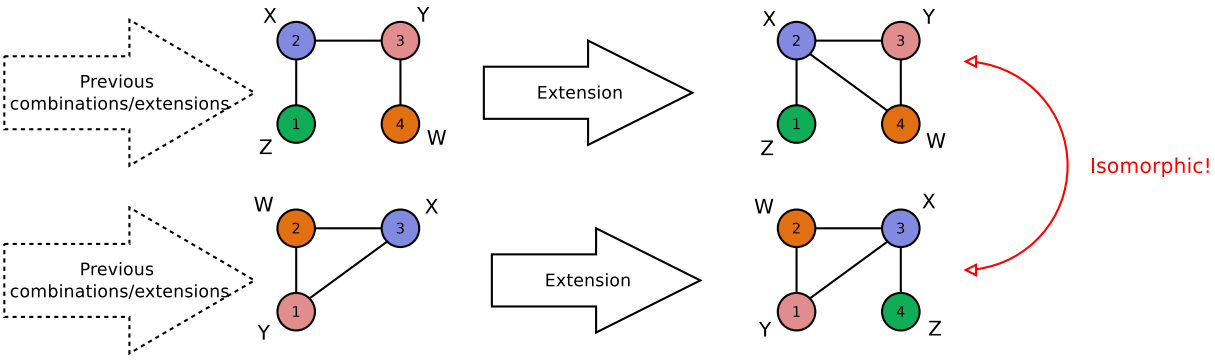


Figure 2.6 – Two different graphs that have been generated via different means are extended in different ways. However, the resulting graphs are isomorphic.

(acyclic graphs) or planar graphs. Finding subgraph isomorphisms (i.e. pattern embeddings), however, is known to be NP-complete. In practice, the difficulty of the problem tends to be not too much of an obstacle when dealing with labeled graphs. Indeed, if the labels of the graph(s) are diverse enough, most candidate isomorphisms can be eliminated quickly and the search can be terminated in reasonable time.

For graph pattern mining approaches, graph isomorphism is a fundamental notion. Even if a *single* isomorphism computation can be done reasonably quickly, the presence of isomorphisms in graphs and their detection can allow important optimisations that make the difference between a usable and unusable approach. Take as an example a pattern mining approach that generates larger patterns by combining and/or extending smaller patterns. Fig. 2.6 presents a situation where two different patterns, each generated from a different sequence of combinations/extensions and each extended differently, actually lead to two isomorphic larger patterns. If the approach did not detect that those two graphs are isomorphic, it would continue expanding both independently. That would lead to performing useless computation as those two search branches are —by definition— equivalent. Detecting (and pruning) equivalent search branches is fundamental for an efficient algorithm, and the difficulty of this detection marks a difference between graphs and other data types, e.g. sets, where that detection is trivial.

Canonical certificates. Most graph pattern mining approaches perform isomorphism checks using what is called a *canonical certificate* or *canonical label*[1] [JCZ13]. A *graph*

1. Not to be confused with vertex and edge labels.

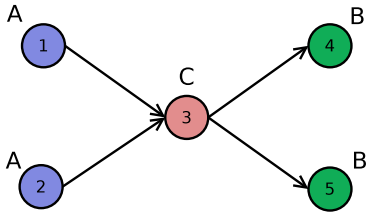
certificate (not necessarily "canonical") is a function C such that for two graphs G and G', $C(G) = C(G') \implies G$ *isomorphic* G' (the converse is not necessarily true however). A trivial example of a graph certificate is the concatenation of V, $V_{\mathcal{L}}$ and $E_{\mathcal{L}}$, which corresponds to a listing of all vertices, vertex labels and edges of the graphs: if two graphs have the exact same list of elements, they represent the same structure. Each graph can have many graph certificates. For example in the case of the aforementioned trivial certificate, if we start listing from a different vertex, we generate a different certificate. The power of graph certificates becomes evident when an ordering is defined for them. Then, the smallest graph certificate possible for a graph is defined as being the *canonical certificate* or *canonical label* of the graph [2]. A canonical certificate is a function $Canon$ such that for two graphs G and G', G *isomorphic* $G' \iff Canon(G) = Canon(G')$. The canonical certificate of a graph can be costly to compute since it is equivalent to an isomorphism check, but allows to compare graphs by a simple string comparison on the certificates. If two graphs have the same canonical certificate, they are definitely the same. And if they are the same, they *must* have the same canonical certificate.

The main reason why canonical certificates are used in graph pattern mining is that they allow to easily perform search space pruning. In a situation similar to the one depicted in Fig. 2.6, a certificate is computed from each expansion, and only the one where the certificate corresponds to the canonical certificate for the graph is expanded further. This allows to only ever continue the exploration of *one* of the isomorphic forms [3]. Another advantage is that with this method each expansion can be evaluated individually: if the certificate associated to the expansion corresponds to the canonical certificate for the graph, the search continues, otherwise it stops. Without the certificate, all expansions would need to be compared pairwise using an isomorphism check to know if two of them represent the same graph structure. This allows to perform depth-first searches in which only the current expansion is kept in memory, greatly reducing memory usage.

Graph automorphisms. *Automorphisms* are isomorphisms of a graph onto itself such that $V_{\mathcal{L}}$ and $E_{\mathcal{L}}$ are preserved. Fig. 2.7 shows an example of a graph that has 4 automorphisms γ_1 to γ_4 (γ_1 being the trivial identity automorphism). Each automorphism can be seen as a permutation of the vertices of the graph: for example, γ_2 represents the permu-

2. Some approaches define the largest certificate (w.r.t. some certificate ordering) as the canonical one, instead of the smallest. The concept stays the same.

3. This optimisation only works if the search algorithm guarantees that it will eventually test all expansions, so that exploring one isomorphic form or another is equivalent.



V	1	2	3	4	5
γ_1	1	2	3	4	5
γ_2	2	1	3	4	5
γ_3	1	2	3	5	4
γ_4	2	1	3	5	4

Figure 2.7 – A graph and its automorphisms.

tation of the two leftmost vertices of the graph. An interesting property of automorphisms is that the composition of two automorphisms is an automorphism itself: for example, γ_4 corresponds to the composition of γ_2 and γ_3. From an abstract algebra point of view automorphisms form a group, and are often studied using notions from group theory [KSM12]. Automorphisms are interesting for approaches that deal with graphs, because they highlight symmetries in the data. For example in Fig. 2.7, we know that talking about vertex 1 or vertex 2 is equivalent, since they can be freely exchanged without impacting the structure of the graph. We will further discuss the impact of the notions of automorphism and isomorphism in Chapter 5, where we present an approach that generates patterns, and thus needs to differentiate their isomorphic forms.

2.1.3 Knowledge Graphs

Knowledge Graphs (KGs) are, as the name implies, data structures used to represent knowledge. They are used in the Semantic Web domain as a mean to make information available in a way that is both interpretable by humans and machines [AVH04]. Nowadays, a huge quantity of data is made available as knowledge graphs. Notable examples are DBpedia [Leh+15], containing billions of pieces of information extracted semi-automatically from the various Wikimedia projects (e.g. Wikipedia); The Mondial database [May99], containing geographical facts about the world; the Google "Knowledge Graph" [4], used to enhance their search results with infoboxes. Many more knowledge graphs exist, with an ever-increasing amount of data available in this form. As we will treat knowledge graphs in Chapter 6 of this thesis, we give here a quick presentation of their structure and usual representations.

Knowledge graphs follow the RDF data model [5]. Several concrete representations of

4. `https://www.blog.google/products/search/introducing-knowledge-graph-things-not/`
5. `https://www.w3.org/TR/rdf11-concepts/`

KGs exist, the most common being RDF/XML [6] and Turtle [7]. We use the latter in this thesis, as it is made to be easily readable by humans. The RDF model defines a KG as composed of *entities*, connected by *properties* detailing the relationships between the entities. Knowledge is expressed as triples $(entity, property, entity)$. Each triple represents a relationship between the two entities, with the semantic of the relationship given by the choice of property. For example the triple *(Alice in Wonderland, author, Lewis Carroll)* can represent the knowledge *"Alice in Wonderland has been written by Lewis Carroll"*. The first entity ("Alice in Wonderland") is called the *subject*, the second one ("Lewis Carroll") the *object* and the property relating them the *predicate*. There are three kinds of entities in knowledge graphs:

- URIs (e.g. `http://dbpedia.org/resource/Lewis_Carroll`). Each URI represents a "thing", i.e. something that exists in the world. It can be something concrete such as a person or document (e.g. Lewis Carroll or Alice in Wonderland), as well as something abstract such as a concept (e.g. Freedom). URIs are the fundamental blocks of KGs. In the Turtle format, URIs are placed between chevrons, such as `<http://dbpedia.org/resource/Lewis_Carroll>`. However we will often shorten them in examples to make them more readable, e.g. `<Lewis_Carroll>`. Note that the Turtle format also allows to define *prefixes* to represent URIs more concisely. For example if we declare the `dbr:` prefix to correspond to `http://dbpedia.org/resource/`, we can represent `http://dbpedia.org/resource/Lewis_Carroll` as `dbr:Lewis_Carroll`.
- Literals (e.g. "1832-01-27"). Literals represents values such as numbers, dates, names, etc. that can be associated to entities. As such, literals can only appear as objects of triples, e.g. `<Lewis_Carroll> <birthDate> "1832-01-27"`. Literals can have a *datatype* (indicated in Turtle by `^^`) describing their semantic. For example in `<Lewis_Carroll> <birthDate> "1832-01-27"^^<date>` we know that the string "1832-01-27" has type `<date>` and as such it has a specific semantic interpretation (e.g. the 27th January 1832). Literals can also have a *language tag* (indicated in Turtle by `@` followed by a language identifier), associating a certain language to the value of the literal. For example in `<Alice's_Adventures_in_Wonderland> <title> "Les Aventures d'Alice au pays des merveilles"@fr`, we know that the literal corresponds to the french title of the book. Note that if a literal has a

6. `https://www.w3.org/TR/rdf-syntax-grammar/`
7. `https://www.w3.org/TeamSubmission/turtle/`

```
@prefix rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:   <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd:    <http://www.w3.org/2001/XMLSchema#>.
@prefix dbr:    <http://dbpedia.org/resource/>.
@prefix dbo:    <http://dbpedia.org/ontology/>.

dbr:Lewis_Carroll       rdf:type          dbo:Person ;
                        rdfs:label        "Lewis Carroll" ;
                        dbo:birthDate     "1832-01-27"^^xsd:date ;
                        dbo:notableWork   dbr:Alice's_Adventures_in_Wonderland .

dbr:Alice's_Adventures_in_Wonderland    dbo:literaryGenre     dbr:Fantasy ;
                                        dbo:releaseDate       "1865-11-26"^^xsd:date.
```

Figure 2.8 – An example knowledge graph, represented in Turtle format, showcasing the notions presented in this section.

language tag, it has the implied datatype of a language-tagged string [8] and as such it can not have both a type specification and a language tag.

- Blank nodes. Blank nodes are entities that have no intrinsic name and as such they are not represented by URIs. They are usually used to encode complex relationships that can not be simply expressed as triples, such as more-than-binary relationships and sequences. In turtle they are encoded as `_:nodeID`, where `nodeID` is just an identifier allowing to recognize the node among multiple triples. Such identifiers are private to the KG, so a knowledge graph can not reference the blank nodes of another graph, unlike URIs.

While entities can be either URIs, literals or blank nodes, predicates can only be URIs. Fig 2.8 shows an example KG, showcasing the aforementioned notions. Note that in Turtle, when a triple has the same subject as the previous one, we can end the previous triple with a semicolon and omit the subject of the following one, to make the representation easier to read. For example `dbr:Lewis_Carroll` is expressed only once even though it is the subject of four triples.

Entity types and lists. When describing knowledge, there are some notions that will be needed very often and in almost all KGs. RDF defines some predicates such that they can be used in all KGs. Since their semantic is known by all users, they favour exchanges between different KGs, without needing every KG to re-define common notions. One such predicate is `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`, often shortened to

8. Corresponding to URL `http://www.w3.org/1999/02/22-rdf-syntax-ns#langString`.

```
@prefix ex:     <http://example.org/Francesco's_thesis#>.
@prefix rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix dbr:    <http://dbpedia.org/resource/>.
@prefix dbo:    <http://dbpedia.org/ontology/>.

# The following
ex:Francesco_Bariatti   a               dbo:Person ;
                        ex:livedIn      (dbr:Milan dbr:Rennes).

# Is equivalent to
ex:Francesco_Bariatti   rdf:type        dbo:Person ;
                        ex:livedIn      _:blankNode1 .
_:blankNode1            rdf:first       dbr:Milan ;
                        rdf:rest        _:blankNode2 .
_:blankNode2            rdf:first       dbr:Rennes ;
                        rdf:rest        rdf:nil .
```

Figure 2.9 – Two equivalent Turtle representations of an example KG, showcasing the `rdf:type` predicate and the RDF list encoding.

`rdf:type`. It is used, as the name implies, to state what the type (or class) of an entity is: a person, a book, a cute cat, ... An entity can have multiple types, for example in the DBpedia KG the city of Milan is both a "Location" and a "Settlement". Since this predicate is so common, in Turtle it can be shortened to `a`.

It is also usual in KGs to have the need to encode ordered lists of things, e.g. the authors of a book. RDF defines an encoding in which each element of the list is represented by a blank node, with a predicate `rdf:first` pointing to the actual entity that represents the element, and a predicate `rdf:rest` pointing to the blank node representing the next element of the list. The end of the list is marked by making the `rdf:rest` predicate point to the `rdf:nil` entity instead of a new blank node. In Turtle, lists can be shortened as space-separated entities between parentheses, which imply the full encoding.

Fig. 2.9 shows an example KG which has a `rdf:type` predicate and a list of entities. Both the turtle-shortened list representation and the full representation are given.

Ontologies. In a knowledge graph, everything that is represented by a URI can be the subject of a triple. This means that it is possible to make meta-KGs containing triples that describe the vocabulary (predicates, types, etc) used in another KG. These meta-KGs are called *schemas* or *ontologies*. For example the DBpedia ontology (containing URIs with base `http://dbpedia.org/ontology/`, often shortened to `dbo:`) contains triples such as `dbo:birthDate rdfs:domain dbo:Person` and `dbo:birthDate rdfs:range xsd:date`, which tell us that the subjects of triples having predicate `dbo:birthDate` are of type

`dbo:Person`, and their objects are dates.

Many ontologies exist for a variety of domains. It is a good practice when publishing a knowledge graph to also publish its ontology, so that it can be easily interpreted. Of course, there also exist ontologies describing vocabularies that can be used to describe ontologies. The most common are RDFS [9] and OWL [10].

Graph representation of KGs. Knowledge graphs are canonically represented as sets of triples $(subject, predicate, object)$. However, they can also have a graph representation, which is often used as a mean of visualizing them. The usual conversion of a KG into a graph —as defined in RDF– is straightforward: every entity becomes a vertex, and every triple becomes a directed edge connecting the vertex corresponding to the subject to the vertex corresponding to the object. Since there can be multiple triples with the same subject and object, the resulting graph is a multigraph. This representation has some limitations when used in a graph pattern mining context, that we discuss in Chapter 6, Section 6.1. The generated graphs tend not to correspond to what is usually expected in that context, where the same knowledge would be represented differently. Therefore in order to use them in a graph pattern mining approach, some adaptations must be made.

2.1.4 Hypergraphs

Hypergraphs are a generalization of the concept of graph. Where a graph has vertex labels —which apply to 1 vertex— and edge labels —which apply to 2 vertices—, a hypergraph has labeled hyper-edges that apply to any number of vertices.

Definition 6 (Hypergraphs) *A* labeled hypergraph $HG = (V, E_{\mathcal{L}})$ *over a set of label symbols* $\mathcal{L}$ *is a data structure composed of a set of* vertices V*, and a set of* labeled hyper-edges $E_{\mathcal{L}} \subseteq \bigcup_{n>0}(V^n \times \mathcal{L})$*. The number* n *of vertices in a given hyper-edge is called the* arity *of the hyper-edge.*

Unordered hypergraphs can also be defined if the order of the vertices in hyper-edges does not matter. Labeled graphs (that we presented previously) correspond to hypergraphs where the hyper-edges have a maximum arity of 2. However in labeled graphs the set of "hyper-edges" is split in two, yielding the set of vertex labels (arity 1) and the set of

9. `https://www.w3.org/TR/rdf-schema/`
10. `https://www.w3.org/OWL/`

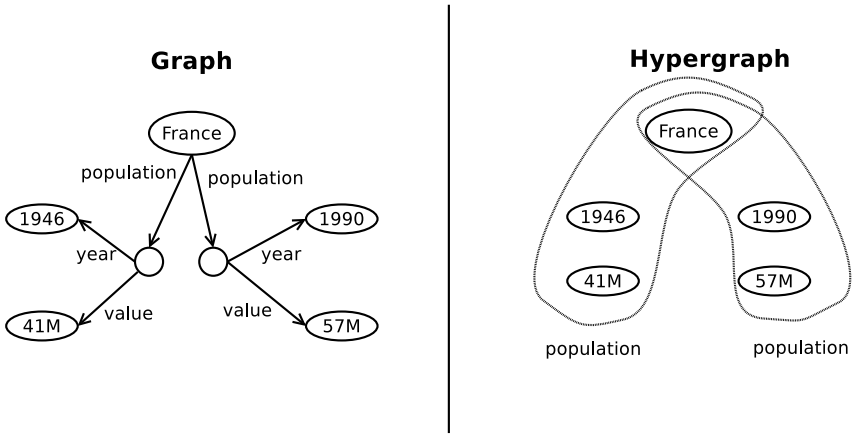


Figure 2.10 – An example on how the same information can be represented differently with a graph and a hypergraphs.

edge labels (arity 2). Therefore, hypergraphs can be used to represent graph data just by joining the set of vertex and edge labels.

An advantage of hypergraphs is that they allow to represent some data, such as more-than-binary relationships, in a more elegant and concise way w.r.t. what would be done with graphs. For example, in Fig. 2.10 we show how the information about the population census of a country can be represented in graphs and hypergraphs. Because of the need to tie a certain population count with not only the country but also its year, the graph representation introduces anonymous vertices, whose only goal is to allow to tie those three pieces of information together two by two (i.e. using edges). On the other hand, the hypergraph representation can use a single hyper-edge of arity 3 to represent this ternary relationship.

In practice, few data is available as hypergraphs in comparison with labeled graphs. This is especially true with the ever-growing amount of data that is available as knowledge graphs (which are multigraphs and not hypergraphs). So while hypergraphs are an interesting abstraction of the concept of graphs, in practice it is much more common to find data encoded using the graph representation shown on the left of Fig. 2.10. In this thesis we mention hypergraphs in Chapter 6. The approach that we present in that chapter uses definitions that are not restricted to edges with an arity ≤ 2, and thus could be applied to hypergraphs.

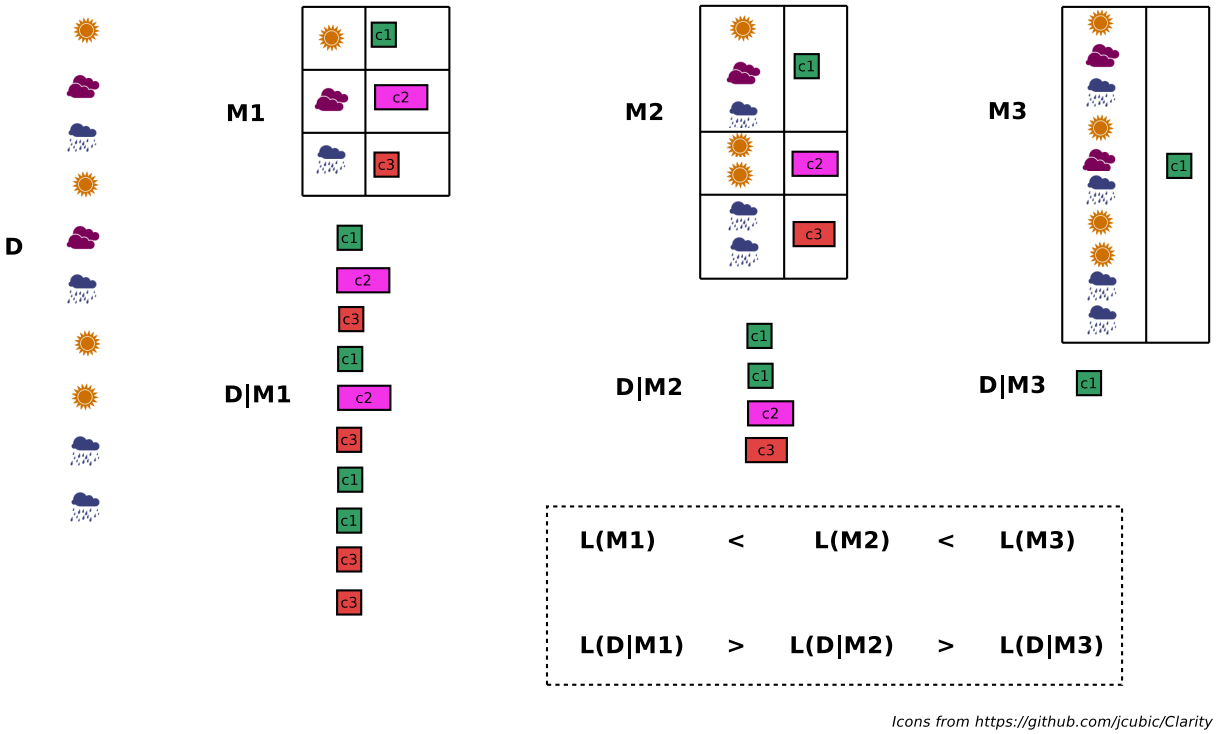


Figure 2.11 – Some data D and how using different models to encode it can affect the MDL description length. Colored squares represent codes.

2.2 The MDL Principle

The *Minimum Description Length* (MDL) principle [Grü07; Ris78] comes from the domain of information theory. It allows to compare models —from a family of models— and identify the one that best describes some data. The MDL principle is often informally presented with the sentence "the model that describes the data the best is the one that compresses the data the best", meaning that a model that fits some data well should allow to describe it with a minimum amount of information, w.r.t. a model that does not fit the data. Formally, the MDL principle states the following:

Definition 7 (MDL principle) *Given a family of models $\mathcal{M}$ and some data D, the best model $M \in \mathcal{M}$ to describe the data is the one that minimizes the* description length *$L(M, D) = L(M) + L(D|M)$, where $L(M)$ is the description length of the model, and $L(D|M)$ is the description length of the data encoded with the model.*

Fig. 2.11 shows an example where some data D (a sequence of sun, cloud, or rain symbols) is encoded using different models M_1, M_2, M_3. M_1 is the simplest of the three

models. It simply associates a code to each symbol (sun, cloud or rain), which can then be used to describe the data. Because of its simplicity, few information is needed to describe this model, and its description length $L(M_1)$ is thus the smallest. However, describing the data with this model is lengthy, since a code is used for each occurrence of each symbol. Thus, the length $L(D|M_1)$ of the data encoded with this model is the highest. M_2 is a slightly more complex model. It associates codes to *sequences of symbols*, so that multiple symbols can be represented with a single code. As such, the description length $L(M_2)$ of this model is higher than the length of model M_1, as more information is needed to describe it. However, this increased model complexity allows to represent the data much more concisely, needing only four codes to represent the whole data. Thus, the description length $L(D|M_2)$ of the data encoded with M_2 is lower than for M_1. M_3 is the extreme case in which the model is simply the data itself. The length $L(D|M_3)$ of the data encoded with the model is the lowest, because all the information is stored in the model itself. However the length $L(M_3)$ of the model is the highest. The MDL principle states that the best model is the one that minimizes the sum of the length of the model $L(M)$ and the length of the data encoded with the model $L(D|M)$. It takes into account both, so as to select a model that manages to express the data with a relatively small description, but it is not too over-engineered w.r.t. the data (such as the extreme case M_3). In practice, this allows to select models that are *descriptive* of the data.

The MDL principle is generic, i.e. not tied to some concrete form of data or model. Each approach using the MDL principle must devise its own definitions for the description lengths of the data and the model. However, there are some primitives that are commonly used for this [Lee01], listed hereafter [11]:

- An element x of a collection $\mathcal{X}$ with uniform distribution can be encoded with a code of length $\log(|\mathcal{X}|)$ bits.
- A choice of k elements from a set $\mathcal{X}$ can be encoded with a code of length $\log(\binom{|\mathcal{X}|}{k})$ bits.
- An element x appearing $\#x$ times in a collection $\mathcal{X}$ admits a code of length $-\log\left(\frac{\#x}{\sum_{x_i \in \mathcal{X}} \#x_i}\right)$ bits. This code is called a *prefix code* [12] and is optimal.
- An integer $n \in \mathbb{N}$ with no known upper bound can be encoded with a code of length $L_{\mathbb{N}}(n)$ bits, where $L_{\mathbb{N}}$ is a *universal integer encoding*. The most commonly used universal integer encodings are the Elias encodings [Eli75].

11. In these formulas, log denotes the base 2 logarithm.
12. Confusingly, prefix codes are also called *prefix-free* codes.

Note that when applying the MDL principle in practice, the actual codes are not needed, but only their length is computed. This simplifies the computation, as one only needs to compute the theoretical length of an encoding, without actually devising the encoding itself. In this thesis we use the terms *description length* and *code length* interchangeably, the act of associating a description length to an element is called *encoding*, and the reverse *decoding*.

Chapter 3

State of the Art

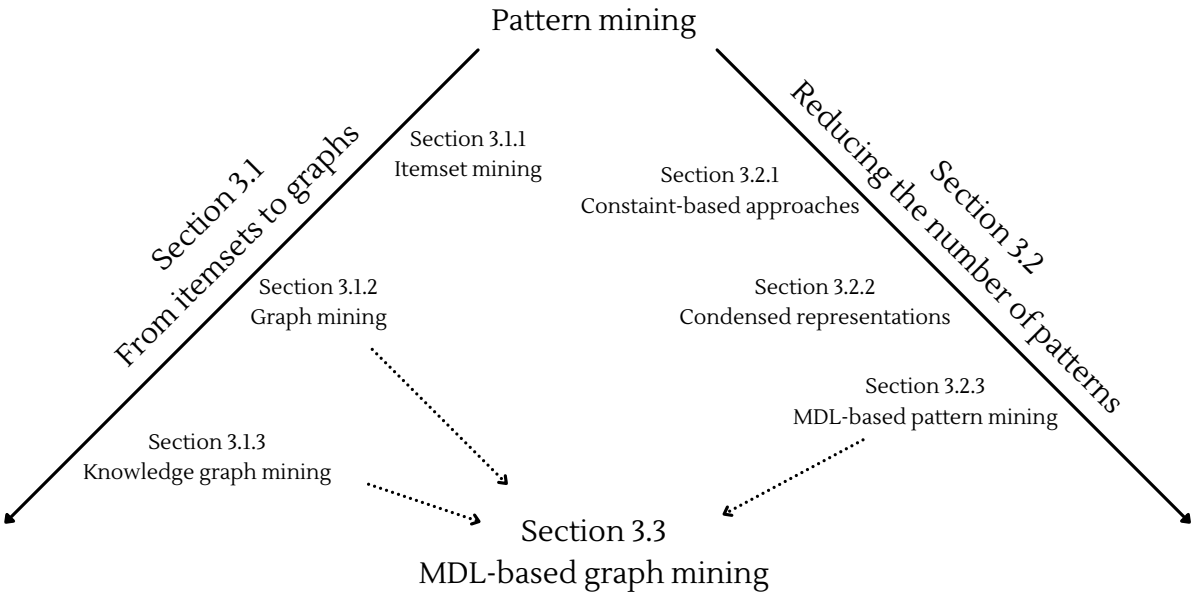


Figure 3.1 – The structure of this chapter, which follows two axes: the mining of graph patterns and the reduction of the number of mined patterns.

The works presented in this thesis are part of the *pattern mining* domain. They are at the intersection of two subdomains: *graph pattern mining* and *MDL-based pattern mining*. In pattern mining, the objective is to help users understand their data by extracting relevant *local structures* —called *patterns*— from it. In this way, the user does not have to analyze the whole data at once, but can instead focus on analyzing those (smaller) structures. Pattern mining is intrinsically an *exploratory* task, with the objective of helping the user explore large quantities of data and extract knowledge [FPSS96].

The structure of this chapter follows two axes, which are shown in Fig. 3.1. Pat-

Transaction	Items
t_1	{🍎, 🍋, 🥥}
t_2	{🍋, 🥥, 🐟}
t_3	{🍎, 🍋, 🥥}
t_4	{🍎, 🐟}
t_5	{🍎, 🍋, 🥥, 🐟}
t_6	{🍎}

Figure 3.2 – An example of a transactional dataset over a set of items $\mathcal{I} = \{$🍎, 🍋, 🥥, 🐟$\}$.

Frequent patterns for $minsup = 3$	
Pattern	support
{🍎}	5
{🍋}	4
{🥥}	4
{🐟}	3
{🍎, 🍋}	3
{🍎, 🥥}	3
{🍋, 🥥}	4
{🍎, 🍋, 🥥}	3

Figure 3.3 – Frequent patterns that can be extracted from the dataset of Fig. 3.2 with $minsup = 3$.

tern mining has classically been performed on transactional data, with *itemset* patterns [Agr+96]. In Section 3.1 we present this setting and core concepts of the pattern mining domain. Then, we present the differences that graph pattern mining has w.r.t. itemset pattern mining, and the specific adaptations and challenges that arise when dealing with graphs patterns and data.

A well-known problem in pattern mining is the so-called problem of *pattern explosion*. Even on small datasets, the set of patterns that can be extracted from the data can be very large in size. In Section 3.2 we present how this can be handled either via constraints and condensed representations, and by using the Minimum Description Length principle [1].

Finally, Section 3.3 ties those two axes together, by presenting how MDL-based approaches have been used to mine graph patterns.

3.1 From Itemsets to Graphs

3.1.1 Itemset Mining on Transactional Data

Classically, pattern mining has been performed on transactional data [AIS93; Agr+96; HPY00]. In this setting, the data D is a list of *transactions*, each transaction containing one

1. Other techniques exist (e.g. sampling). The ones that we present in detail are the ones that are prominent in graph mining.

or more *items* from a set of possible items $\mathcal{I}$. For example, in a retail dataset each transaction may correspond to a client's purchase and is composed of the products that the client purchased. Fig. 3.2 shows an example of a transactional dataset $D = \{t_1, t_2, t_3, t_4, t_5, t_6\}$ containing 6 transactions over a set of items $\mathcal{I} = \{$🍎, 🍋, 🍪, 🐟$\}$, where each transaction t_i corresponds to a subset of $\mathcal{I}$ (e.g. $t_1 =$ {🍎, 🍋, 🍪}). In this transactional setting, patterns are called *itemsets*, and correspond to subsets of $\mathcal{I}$. A pattern appears in a transaction if the pattern is a subset of the transaction. The number of transactions containing the pattern is called the *support* of the pattern.

Definition 8 (Itemset support) *Let D be some transactional data over a set of items $\mathcal{I}$. The* support *of an* itemset pattern *$P \subseteq \mathcal{I}$ is defined as $support(P) = |\{t \in D \mid P \subseteq t\}|$*

For example, in Fig. 3.2 the items 🍎 and 🍋 appear together in transactions t_1, t_3 and t_5, hence $support(\{$🍎, 🍋$\}) = 3$. [2]

Historically, the first itemset mining algorithms were developed to extract *frequent* patterns. A frequent pattern is a pattern whose support is greater than a user-supplied *minsup* parameter.

Definition 9 (Frequent patterns) *Let D be some transactional data over a set of items $\mathcal{I}$, support a support measure over D, and $minsup \in \mathbb{N}$ a user-supplied parameter. The set of* frequent itemset patterns *$\mathcal{F}$ is defined as $\mathcal{F} = \{P \subseteq \mathcal{I} \mid support(P) \geq minsup\}$.*

Fig. 3.3 shows all frequent itemsets that can be extracted from the data of Fig. 3.2 for $minsup = 3$, alongside their support. For example, {🍋, 🍪} is frequent because it has a support of 4 (it appears in t_1, t_2, t_3 and t_5), while {🍪, 🐟} is not, since it has a support of 2 (it appears only in t_2 and t_5).

By varying the value of *minsup*, the user can decide in how many transactions a pattern should appear to be "interesting". This allows to eliminate spurious patterns that appear only in few transactions, and focus on highly-relevant patterns.

An important property of the support measure is that it is *anti-monotonic*, i.e. the support of a pattern is always lower or equal to the support of its sub-patterns.

Definition 10 (Support anti-monotonicity) *Let support be a support measure. support is* anti-monotonic *if for all patterns P_1 and P_2, $P_2 \subseteq P_1 \implies support(P_1) \leq support(P_2)$.*

2. Some definitions of support divide the number of transactions containing the pattern by the total number of transactions in D, in order to normalize the support to a value between 0 and 1.

Data of Fig. 3.2 projected on {🍎}	
t_1	{🍋, 🍪}
t_3	{🍋, 🍪}
t_4	{🐟}
t_5	{🍋, 🍪, 🐟}
t_6	∅

Support of {🍎, ? } patterns	
Pattern	support
{🍎, 🍋}	3
{🍎, 🍪}	3
{🍎, 🐟}	2 (not frequent for $minsup = 3$)

Figure 3.4 – Projected database from the data of Fig. 3.2 for pattern {🍎}, and the patterns {🍎, ? } that can be deduced from it.

Corollary: *If P_1 is not frequent ($P_1 \notin \mathcal{F}$), then all patterns $P_3 \supseteq P_1$ are not frequent as well.*

For example, on the data of Fig. 3.2 the support of {🍎,🍋} is 3, while the supports of its subsets {🍎} and {🍋} are respectively 5 and 4.

On a set of items $\mathcal{I}$, there are possibly $2^{|\mathcal{I}|}$ itemsets. Enumerating them all to compute their support is infeasible on large datasets. The support anti-monotonicity is a valuable property that can be used to prune the search space and thus compute frequent itemsets in reasonable time. The Apriori algorithm [Agr+96] proposes a pruning strategy that relies on this property. The algorithm performs a breadth-first search, first computing all 1-item frequent itemsets, then all 2-item frequent itemsets, and so on. It generates candidate (k+1)-size itemsets by associating pairs of frequent k-size itemsets that differ in exactly one item (so that their union contains exactly k+1 items). A necessary condition for an itemset to be frequent is that all of its subsets are frequent as well. Thus, candidate (k+1)-size itemsets are pruned if any of their k-size subsets are not frequent. Only for the remaining candidates the support is then computed against the data (which is a time-consuming task on large datasets). Candidates with a support lower than *minsup* are pruned, and the operation starts again for (k+2)-size itemsets. The algorithm stops when the set of frequent k-size itemsets is empty, since it means that no (k+n)-size itemset can be frequent.

The Apriori algorithm has some drawbacks. First, its breadth-first search strategy requires storing all k-size patterns at each step, in order to generate (k+1)-size candidates. This can become a problem for larger datasets or small values of *minsup*, where so many patterns are generated that the memory capabilities of the machine are exceeded. Second, the (k+1)-size candidates that are generated are not guaranteed to be frequent, and computing their support requires scanning all the database, which can take a long time on large datasets. To address these drawbacks, *pattern-growth* algorithms have been proposed,

the most notable being FP-Growth [HPY00]. This algorithm uses a structure called the FP-tree (that we do not detail here) to rewrite the data in such a way that for any item, it is quick to access all the transactions that contain the item. The algorithm then uses the FP-tree to mine frequent patterns using a depth-first search. The algorithm starts with a frequent 1-item pattern, and uses the FP-tree to compute a *projected database*, i.e. a subset of the data formed by the transactions that contain the pattern. For example on the left of Fig. 3.4 we show the projected database for the data of Fig. 3.2 and the 1-item pattern {🍎}. Since transaction t_2 does not contain the pattern, it is not in the projected database. The pattern itself is not reported in the projected database's transactions as it is implied. The algorithm then computes all frequent 1-item patterns on the projected database. The reason is that if a pattern X is frequent on the projected database of pattern P, then $P \cup X$ is a frequent pattern on the original data. Then, the algorithm iterates again, computing projected databases for those new frequent patterns, and computing frequent items on those databases to generate larger frequent patterns, and so on. On the right of Fig. 3.4 we show the support of {🍎, ? } patterns, which can be deduced from the projected database of {🍎}. Thanks to the FP-tree and projected database structures, FP-Growth can compute frequent itemset patterns more efficiently than Apriori. Since it focuses on subsets of the data, it requires less scans of the complete data, making the computation quicker. Since the computation is done with a depth-first search, less patterns have to be stored in memory at a time, reducing the memory footprint.

Association rules. Association rules are structures of the form $X \to Y$, where X —the *body*— and Y —the *head*— are patterns. Such a rule is interpreted as "when X is present, often Y is present as well". Many pattern mining algorithms can not only extract frequent patterns, but *association rules* as well. This is the case for the aforementioned Apriori [Agr+96] and FP-Growth [HPY00]. This is because mining frequent patterns is often the first step in mining association rules.

3.1.2 Graph Mining

Graph pattern mining, or *graph mining* for short, is the subdomain of pattern mining in which the data is a graph (see Section 2.1). The extracted patterns are generally either graphs themselves, or association rules with a graph body. However, some approaches may extract simpler patterns (e.g. itemsets) from the graph data, to lower the task's complexity. The graph mining setting differs from the classical itemset setting because

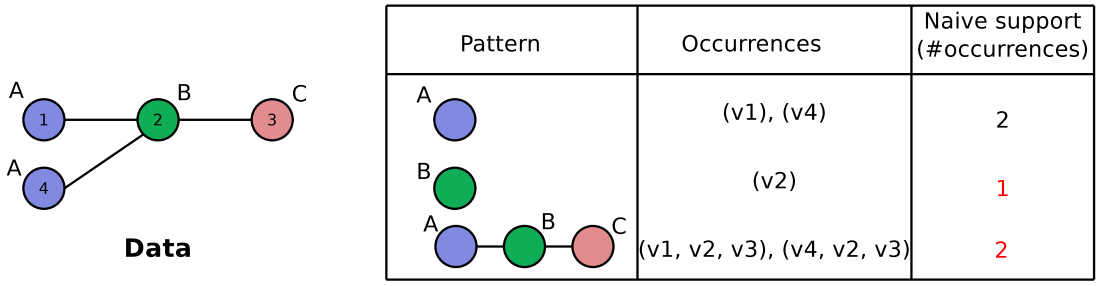


Figure 3.5 – Example showing that a naive support that counts the number of occurrences of a pattern is not anti-monotonic.

graphs are more complex structures than itemsets. Even a basic operation such as checking how many times a pattern appears in the data is NP-complete in graph mining because it needs to perform a subgraph isomorphism check (see Section 2.1.2).

Support in graphs. The definition of a support measure is not straightforward in graphs. At their core, the measures used on graphs have the same notion as the itemset support, i.e. they measure "how many times a pattern appears in the data". However, on graphs naively counting the number of occurrences of a pattern in the data (i.e. "how many times a pattern appear in the data") yields a measure that is *not* anti-monotonic. This can be seen in Fig. 3.5, where a naive support measure is defined as the number of occurrences of the pattern. In this example, pattern `A-B-C` has two occurrences (depending on the choice of the first vertex), but its sub-pattern `B` only has one (in the middle). Thus this naive measure is not anti-monotonic. Therefore, it does not allow for efficient search-space pruning. The actual support measure used by graph mining approaches depends on the type of graph data handled by the approach. There are two types of graph data: collections of graphs (e.g. a collection of molecules, a collection of sentences of a document), and single large graphs (e.g. a single large protein, a social network).

Most approaches [IWM00; KK01; BB02; YH02; HWP03; NK04; Zhu+07; YH03; Hua+04; YZH05; TVK10] accept the data as a collection of graphs. They define the support of a pattern as the number of graphs in the collection that contain the pattern. Since each graph of the collection is counted only once, independently of the number of occurrences that the pattern has in it, the measure is anti-monotonic.

Other approaches accept the data as a single graph [KK05; Kou+15; CH93]. In that case, the so-called "overlap-based support" or "minimum image based support" measures [BN08] are used instead. The *overlap-based support* is computed by creating an

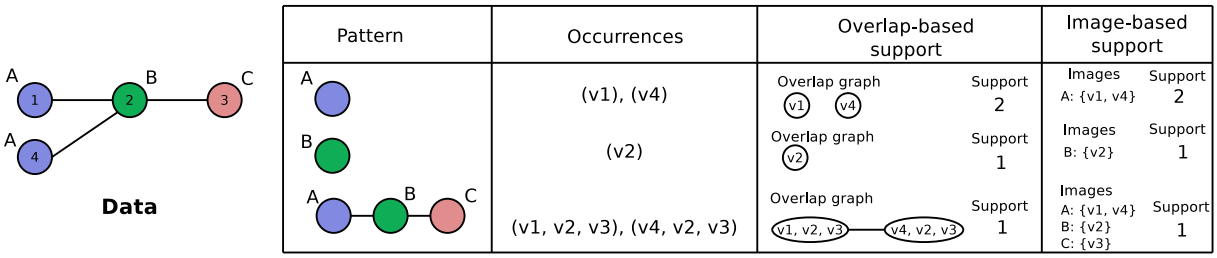


Figure 3.6 – Two anti-monotonic support measures that are used in graph mining when the data is a single graph.

"overlap graph" for the pattern: a graph where each vertex represents an occurrence of the pattern, and edges connect occurrences overlapping on at least one data edge [3]. The support of the pattern corresponds to the size of the Maximum Independent Set (MIS) that can be found on its overlap graph, i.e. the maximum number of occurrences of the pattern that can be chosen in the data such that no two occurrences overlap. This measure is anti-monotonic, however it requires computing the MIS of the overlap graph, which is a NP-complete computation [BN08]. The *image-based support* is computed by counting for each of the pattern's vertices the number of different data vertices to which it is mapped by the pattern's occurrences. The support is defined as the lowest of these values, i.e. the smallest number of data vertices corresponding to a pattern vertex. This measure is anti-monotonic and it does not need to perform a NP-complete computation. These two measures are illustrated in Fig. 3.6.

Rarely, approaches accept as input data both collections and single graphs [GSV06].

Apriori-based and pattern-growth approaches. Graph mining approaches can be divided in two families following the method that they use to generate patterns. Similarly to itemset mining, there are *Apriori-based* approaches and *pattern-growth* approaches [RP15; JCZ13].

Apriori-based approaches such as AGM [IWM00], FSG [KK01], DPMine [GSV06] and HSIGRAM [KK05] generate candidate patterns by merging smaller patterns using a breadth-first search. Pairs of frequent patterns of size k that have a common (k-1)-size core (i.e. they differ in only one element) are merged so as to generate candidate (k+1)-size patterns. This search strategy is the same for all Apriori-based approaches and they mainly differ in the way that they define a "k-size pattern": some count the number

3. A similar measure can be defined by taking into account vertex overlaps instead of edge overlaps.

of vertices, some the number of edges, and some use more sophisticated measures such as the number of disjoint paths. Apriori-based graph mining approaches have the same drawbacks that their itemset counterparts. First, the breadth-first search strategy requires to store all k-size patterns at each step. This becomes a problem in larger datasets or when the support threshold *minsup* is set to low values, since the number of patterns to store may be so large that the memory capabilities of the machine are exceeded. Second, these approaches generate many non-frequent candidate patterns, which consume computation time even if they are not shown to the user. Additionally, Apriori-based *graph* mining approaches have a drawback that their itemset counterparts do not have. Computing if two k-size patterns have a common (k-1)-size core is trivial on sets, but requires a NP-complete subgraph isomorphism check on graphs.

Pattern-growth approaches such as MoFa [BB02], gSpan [YH02] (the most cited and well-known graph mining approach), VSIGRAM [KK05], and Gaston [NK04] generate patterns by performing frequent extensions of smaller frequent patterns using a depth-first search. These approaches start with a small frequent pattern and look at the neighbourhood of its occurrences for possible extensions that appear enough times so that the pattern plus the extension give a new frequent pattern. For example gSpan starts with patterns with two vertices and a single edge connecting them, and each extension adds a single edge to the pattern, either between existing vertices or by adding a new vertex to the pattern. By using a depth-first search strategy, pattern-growth approaches avoid storing many candidates in memory, since only one branch of the search tree is expanded at a time. Also, since they only perform frequent extensions, they do not spend time computing the support of infrequent candidates. Those two points make them more efficient than Apriori-based approaches. An important point in the efficiency of pattern-growth approaches is to avoid generating multiple times the same patterns from different search branches. If that was not avoided, the algorithm would spend time exploring multiple times the same search space. While this can be trivially done on itemsets by defining an absolute order over the items [HPY00], it is particularly difficult to achieve on graphs since graphs can have many isomorphic forms. Because of isomorphisms, a graph pattern can be described in several different ways (see Section 2.1.2), which make the detection of equivalent search branches difficult. Most approaches use canonical graph certificates in order to manage this requirement. In [Wö+05], a quantitative comparison of the main existing pattern-growth approaches is done. It highlights the importance of performing search-space pruning by using graph certificates, and the importance to avoid as much as

possible NP-complete support computations.

Some approaches such as FFSM [HWP03] exist that are at the border between the Apriori-based and pattern-growth families. They perform both joining and extension operations in order to generate patterns. Because of that, in the literature they can be classified both as Apriori-based and pattern-growth. However, they are not much more efficient than pure pattern-growth approaches [Wö+05].

3.1.3 Knowledge Graph Mining

Some pattern-mining approaches have been proposed to target specifically knowledge graphs (see Section 2.1.3). Knowledge graphs tend to have some differences from the graphs classically encountered in graph mining. First, they are directed multigraphs, while usually graph mining approaches deal with simple graphs. Second, KGs datasets are usually composed of a single large graph, while most graph mining approaches expect a collection of graphs. Third, in real-life KGs, vertices tend to have a degree (i.e. the number of adjacent edges) that follows a power-law [GGD04]. This means that most vertices have few edges around them, while some "hub" vertices have a huge number of vertices around them. This is not the case in the graph datasets that are usually used in conjunction with classic graph mining approaches, which have more uniform degree distributions. Those differences make KG mining using classic graph mining approaches challenging, and explain why algorithms have been developed to treat them specifically.

Some approaches such as SWApriori [Ram+14], SWARM [BBL16] and the one proposed by Bobed et al. [Bob+20] convert the KGs to transactions to lower the complexity, then apply itemset-mining algorithms to the converted version. This simplifies the task, but of course limits the structure of the patterns that can be extracted. Most approaches, of which we can cite AMIE+ [Gal+15] and KGist [Bel+20] do not simply extract patterns but instead extract association rules whose heads or bodies (but not necessarily both) are graph patterns. The rules are often used in tasks such as link completion and error detection. The former consists in suggesting edges that are missing in the KG, and the latter in detecting edges that should not be in the KG (corresponding to wrong data). Their main disadvantage is that the patterns often have limited shapes, and that they are generated with a specific goal in mind (link completion or error detection). In general, we find in the literature that most pattern mining approaches for KGs are created with a specific task in mind, and do not perform other tasks, even when the adaptation would be easy [Pau17].

3.2 The Pattern Explosion Problem

Historically, the first pattern mining approaches have been *complete approaches*. These approaches extract *frequent patterns*, i.e. all patterns having a support greater than a user-supplied threshold *minsup*. While complete approaches are very efficient in mining many patterns in as little time as possible, they have a significant drawback that hinders their usage by a human analyst: the number of extracted patterns. In fact, the set of frequent patterns can get extremely large, even on medium-sized dataset. This problem is often called *the pattern explosion problem*. When the support threshold is lowered, the number of frequent patterns can increase exponentially. For low support values, it is not unusual to have hundreds of thousands —or even millions— of frequent patterns. For example in [YH03], it is noted that on a dataset of 422 graphs averaging 42 vertices in size, a support of 5% yield more than 10^6 frequent patterns. On the other hand, a high support value allows to keep the extracted patterns set more human-sized, but it may skip patterns that would be interesting for the analyst but are less frequent.

In this section we present some techniques that have been proposed in order to select a smaller and more manageable set of patterns, so as to make human analysis easier. First, we present approaches that integrate constraints into the mining process. Then, we present approaches that use condensed representations in order to reduce the amount of information shown to the user. Finally, we present approaches that use the Minimum Description Length principle (see Section 2.2) to select small sets of patterns. The techniques that we present in this section are the ones that are predominantly used to tackle the pattern explosion problem in graph pattern mining. However, other techniques exist to tackle this problem in the transactional setting, such as random sampling [To96; DL17], or using other measures than the support to select patterns [GH06]. Approaches have also been proposed that evaluate pattern sets instead of individual patterns [RZ07].

The techniques presented in this section are applied in a similar way to transactional and graph data. We use the transactional setting when presenting examples as it is generally easier to understand.

3.2.1 Constraint-based Approaches

In order to reduce the set of extracted patterns to a smaller size in a way that is in line with the user's expectations, *constraint-based approaches* [Ng+98] allow to include user-defined constraints into the mining process. On graphs, the most notable constraint-based

approach is the gPrune framework [Zhu+07]. Those approaches allow the user to state some constraints, and they only mine patterns that follow them. For example, the user may express that they are only interested in patterns where the vertices have many connections between them and thus reject patterns that have a sparse structure. If the constraints are well-selected, they can reduce the number of patterns and speed up the computation by pruning search branches. However, the benefits of such methods are limited [JCZ13]. First, they require the user to express constraints on the data, which implies having some knowledge about it. This is not suitable for exploratory settings, in which the data is not well-known in advance, and the user does not know which characteristics the extracted patterns should have. Second, the performance of the mining is tightly related to the performance of the constraint-checking function. If the user expresses constraints that take a long time to verify, they hinder the mining process more than help it.

3.2.2 Condensed representations

A drawback of the frequent pattern sets extracted by complete approaches is that they tend to contain a lot of redundancies, because when a pattern is frequent all of its sub-patterns are frequent as well. So if pattern {  } is in the set of frequent patterns, then so are {  }, {  }, {  }, {  }, {  } and {}. While it makes sense for all of those patterns to be in the set of frequent patterns —as they all are frequent— their presence can be deduced by the presence of pattern {  }, and as such they do not give additional information. On the contrary, they clutter the pattern set, making the work of the analyst more difficult (since they now have to sieve through 6 additional patterns). In order to tackle this problem, pattern mining approaches have been proposed to extract *maximal frequent patterns* or *closed patterns* [Bay98; Pas+99]. On graphs, the most common are SPIN [Hua+04] and MARGIN [TVK10] for maximal patterns, and CloseGraph [YH03] and SPLAT [YZH05] for closed patterns.

A pattern is a *maximal frequent pattern* if it is frequent and it is not part of any other frequent pattern. In other words, maximal patterns are the biggest possible patterns for a given support threshold.

Definition 11 (Maximal itemset patterns) *Let $\mathcal{F}$ be the set of all frequent patterns. The set of* maximal frequent patterns *is defined as* $\{X \in \mathcal{F} \mid \forall Y \supset X, Y \notin \mathcal{F}\}$

With graph patterns, the definition is the same as for itemsets, with the difference that the patterns are not compared by set inclusion, but by subgraph isomorphism. Maximal

Data

Transaction	Items
t_1	{🍎, 🍋, 🟤}
t_2	{🍋, 🟤, 🐟}
t_3	{🍎, 🍋, 🟤}
t_4	{🍎, 🐟}
t_5	{🍎, 🍋, 🟤, 🐟}
t_6	{🍎}

For $minsup = 3$

Frequent patterns $\mathcal{F}$: { {🍎}, {🍋}, {🟤}, {🐟}, {🍎, 🍋}, {🍎, 🟤}, {🍋, 🟤}, {🍎, 🍋, 🟤} }

Maximal patterns: { {🐟}, {🍎, 🍋, 🟤} }

Closed patterns: { {🍎}, {🐟}, {🍋, 🟤}, {🍎, 🍋, 🟤} }

Figure 3.7 – An example transactional dataset and the frequent, maximal and closed patterns that can be extracted from it for a support threshold $minsup = 3$.

patterns summarize the complete set of frequent patterns, since all frequent patterns can be deduced from them by considering all of the possible sub-patterns of each maximal pattern.

Maximal patterns allow to summarize the set of frequent patterns, but lose some information about the frequency of the sub-patterns. For example, let $P_1 = \{$🍎, 🍋, 🟤$\}$ be a maximal pattern. Therefore we can deduce that $P_2 = \{$🍋, 🟤$\}$ is a frequent pattern. However, we can not deduce whether 🍋 and 🟤 always appear together with 🍎, or if sometimes they appear without. In other words, we can not deduce whether *all* occurrences of P_2 are also occurrences of the larger pattern P_1, or not. Thus, we are missing some information about the frequency of P_2 w.r.t. the frequency of P_1. *Closed patterns* avoid this problem. A pattern is closed if it does not exist a larger pattern with the same support. Conversely, if a pattern is a closed pattern, its sub-patterns either have the same support or they are closed patterns themselves.

Definition 12 (Closed itemset pattern) *Let $\mathcal{F}$ be the set of all frequent patterns. The set of* closed patterns *is defined as* $\{X \in \mathcal{F} \mid \forall Y \supset X, support(Y) < support(X)\}$

With graph patterns, the definition is the same as for itemsets, with the difference that the patterns are not compared by set inclusion, but by subgraph isomorphism. Closed patterns are more powerful than maximal patterns as they convey all information of the set of frequent patterns without loss of information. Additionally, the set of maximal patterns is by definition a subset of the set of closed patterns.

Fig. 3.7 shows an example of transactional data and the frequent, maximal, and closed itemset patterns that can be extracted from it. Maximal and closed patterns can greatly reduce the number of patterns that are extracted from a dataset, by several orders of

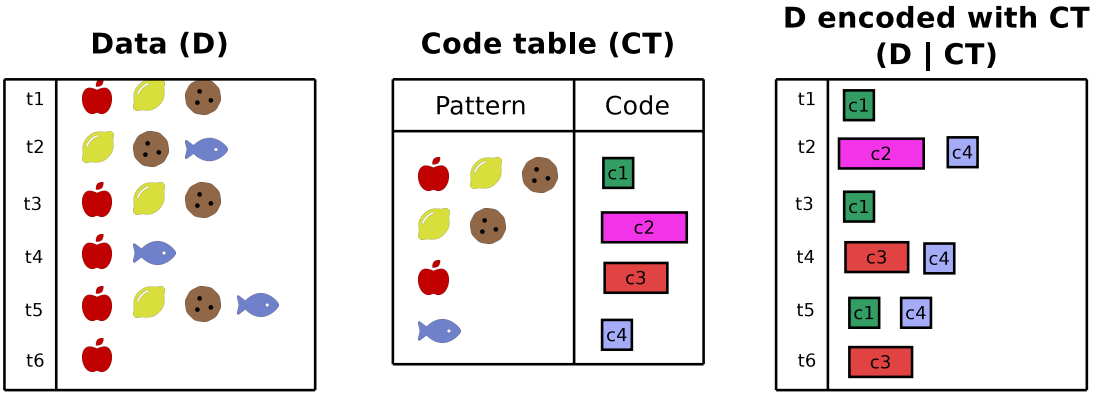


Figure 3.8 – Example of a Krimp's code table and how it is used to encode some data. The length of the code table's codes depends on how often they are used when encoding the data.

magnitude in some cases. However, the resulting set of patterns can still contain thousands of patterns or more [YH03]. Different approaches are thus needed in order to enable a human analysis.

3.2.3 MDL-based Pattern Mining

While frequency is surely an important characteristics for patterns (as frequent patterns highlight recurrent structures in the data), we have seen that using it as the only measure for pattern selection leads to sets of patterns that are too large and redundant for human analysis. While constraints and condensed representations such as maximal or closed patterns can reduce the number of patterns, they do not solve the problem. More recently, *MDL-based approaches* have been introduced that significantly reduce the number of extracted patterns and select patterns that are descriptive of the data, allowing a human to analyze it [Gal20]. In this section we present MDL-based approaches on the transactional setting, while in Section 3.3 we detail MDL-based approaches for graph mining. The most notable approaches in the transactional setting are Krimp [VLS11] and its successor Slim [SV12]. Krimp selects a small and descriptive subset of patterns from a set of patterns given as input, for example the set of frequent patterns generated by a complete approach. Slim does not require a set of patterns as input, as it generates patterns all the while selecting them. As a result, it only requires the data as input and it is more efficient than Krimp. These approaches use the Minimum Description Length principle (see Section 2.2) to evaluate sets of patterns, and thus guide the pattern selection.

An approach using the MDL principle (see Section 2.2) has to define a family of models $\mathcal{M}$, a way to encode the data using a given model $M \in \mathcal{M}$ $(D|M)$, and a way to measure the description length of the model and the encoded data ($L(M)$ and $L(D|M)$), so that the MDL description length measure $L(M, D) = L(M)+L(D|M)$ can be computed to evaluate models of $\mathcal{M}$. Krimp and Slim use as model a dictionary-like structure called a *Code Table* ($M = CT$). A code table contains a set of patterns, and codes associated to them. In order to encode the data using a code table ($D|M = D|CT$), Krimp replaces each transaction in the data with a sequence of pattern codes from the CT. For example a transaction {🍃, 🍪, 🐟} can be encoded with $code(\{🍃, 🍪\}) + code(\{🐟\})$. Fig. 3.8 shows an example data and code table, and the resulting encoded data. Note that when encoding the data, Krimp forbids patterns to overlap: so a transaction {🍃, 🍪, 🐟} could not be encoded by $code(\{🍃, 🍪\}) + code(\{🍪, 🐟\})$. Krimp decides which patterns to use to encode a transaction by going through the CT in order: the first pattern that can encode a part of the transaction is used, and following patterns can only encode the remaining items of the transaction. Therefore, the order of the patterns in the CT has an impact on the encoding. Krimp chooses this order based on a heuristic. Several heuristics have been proposed in [VLS11] and the one that performed the best experimentally —ordering patterns by descending size then descending support— was retained. Once the model and data encoding have been defined, Krimp defines formulas to compute their MDL description length ($L(M) = L(CT)$ and $L(D|M) = L(D|CT)$). This allows to compute the MDL description length measure $L(M, D) = L(CT, D) = L(CT) + L(D|CT)$, to evaluate a given code table. Thus, code tables can be compared numerically using this MDL description length value as a criterion to choose the one that best suits the data. Since each CT represents a set of patterns, Krimp has a way to choose between all possible sets of patterns. A naive way for Krimp to select the best subsets of the patterns given as input would be to create all possible CTs from all possible subsets of patterns, and evaluate them all using the MDL description length measure as a criterion to choose the best one. This is however not possible due to the huge number of possible subsets that can be generated (2^N possible subsets for N patterns given as input). Therefore, Krimp uses a greedy search strategy guided by heuristics. Starting from a code table composed by 1-item patterns, the algorithm tries to add one by one the patterns given as input (the order in which they are tried is given by a heuristic). If the addition of a new pattern improves the code table, the pattern is kept, otherwise it is discarded. This process is repeated until all input patterns have been tested. The patterns that are shown to the

user correspond to the patterns in the final code table after all input patterns have been tested.

Most MDL-based pattern mining approaches use dictionary-like structures similar to Krimp's code table [Gal20]. MDL-based approaches have been proposed to extract patterns from many kinds of data: relational databases [KS09], sequences [TV12; Gal+19], and matrices [FL20] to cite a few. The main advantage of MDL-based pattern mining approaches is that they evaluate sets of patterns *as a whole*. In more classic approaches each pattern is evaluated individually by some measure. Because of this, many similar patterns with similar values for that measure tend to be generated, yielding a lot of redundancies that can only be removed afterwards (e.g. with the maximal and closed patterns representations). On the contrary, in MDL-based approaches the evaluation of a set of patterns takes into account the interactions between the different patterns that compose it. This makes the approaches very powerful for human analysis, since humans do not analyze each of the extracted patterns in isolation. A second advantage of MDL-based approaches is the power to find a balance between the model complexity and its ability to describe the data. In the MDL description length measure that is used to evaluate a given model, both the length of the model and the length of the data encoded with the model appear. Because the length of the model appears, MDL-based approaches avoid generating models that are too complex. In [VLS11] it is shown that the selected sets of patterns are small but also descriptive of the data (a property evaluated through a classification task).

However, MDL-based approaches also have some disadvantages. A first problem is that the MDL principle is generic, i.e. not tied to some concrete form of data or model. Each approach based on the MDL principle must devise its own definitions for the description lengths of the data and the model, and the encoding of the data using the model. Designing the encoding gives a lot of power to the developer to tailor the approach to its own use case, but it also means that some biases can be introduced that favour some patterns over others. Also, the MDL principle expects that the encoding is designed to be lossless, meaning that one must be able to reconstruct the data from its encoded version. This is however not the case for all MDL-based approaches [Gal20]. A second problem that must be tackled is that the number of possible pattern sets is huge. In order to select a subset of patterns from a larger set, it is simply not feasible in reasonable time to evaluate all possible subsets. Therefore, MDL-based approaches use greedy search strategies guided by heuristics to explore the pattern space. Thus, when designing a MDL-based approach,

one must also design an efficient search algorithm in order to exploit it.

3.3 MDL-based Graph Pattern Mining

Few MDL-based approaches have been proposed to extract patterns from graph data. Among them, the most notable are SUBDUE [CH93], Vog [Kou+15], MeGS [Goe+16], KGist [Bel+20], and the approach proposed by Bloem and De Rooij [BR20].

SUBDUE [CH93] has been proposed to mine substructures from both labeled and unlabeled, directed and undirected graph data. It works in an iterative way. Each iteration, a pattern is generated from the data, using a MDL criterion to choose it. Then, each occurrence of the pattern is replaced by a single vertex. Edges that connected vertices in the occurrence to vertices in the rest of the data are rewired to connect to this new occurrence-vertex. Then, a new iteration starts and so on. SUBDUE has two main advantages. First, the extracted patterns can have any shape, they are not limited to pre-defined shapes. Second, the occurrence-vertices generated during an iteration can be used in following iterations, so that patterns can contain other patterns, generating a hierarchical view of patterns, which can give useful insights. However SUBDUE also has two major disadvantages. First, since the edges that connect a pattern occurrence to the rest of the data are rewired to a single occurrence-vertex, the information about which of the pattern vertices connected to a specific edge is lost. This is particularly noticeable when patterns include each other hierarchically. It is possible to know that Pattern A contains Pattern B, but not exactly *how* they are related. Also, MDL approaches are usually expected to use lossless encodings, in order to avoid information loss [Gal20]. The second disadvantage is that SUBDUE's MDL criterion is only used to evaluate one pattern at a time. Thus, the extracted patterns are not evaluated *as a set*, but each individually. This is in contrast with most MDL-based approaches that evaluate sets of patterns as once, and as such take into account interactions between patterns.

VoG and MeGS apply to unlabeled graphs (sometimes called *networks*). Their goal is to summarize the structure of a graph using a vocabulary of pre-defined structures (cliques, stars, bipartite cores, ...). These pre-defined structures are common structures observed in real-life networks. For example a star is a group of unconnected vertices all connected to a same central hub. The MDL principle is used to choose which structure to use to represent each group of vertices. Description length formulas are given to describe a group of vertices using the priors given by each structure: for example in order to describe

a star only the identities of the hub and the vertices that participate to the star are needed, as the edges between them are implied. Missing or excess edges are encoded by an error matrix. These description lengths are then used in order to choose which structures should be used to describe different parts of the data. As a result, the approaches can output a summary of the structure of the data as a composition of those different structures. The main difference between VoG and MeGS is that the former allows the structures to overlap on vertices (but not edges), while the latter does not. The main shortcoming of these approaches is their limitation to a predefined vocabulary of structures. While the vocabulary could be expanded, doing so requires a prior knowledge about the type of structures that can be present in the data.

KGist [Bel+20] is a rule-mining approach aimed specifically at knowledge graphs. Rules are expressed as rooted tree patterns (a subtype of graphs). The MDL model is a collection of rules, with associated information used to match them to specific parts of the data. The MDL principle that guides the search allows to select a set of rules that can be used to detect both recurrent structures in the data, but also missing and wrong edges in parts that do not follow the rules. This is particularly interesting in knowledge graphs, as many KGs contain missing information and errors, and the tasks of link prediction and error detection are prominent in the KG domain. The main advantage of KGist is that the rules that are extracted are not limited to pre-defined shapes. However, because of the way that the rules are defined, they can not express cycles, i.e. chains of the type "A goes to B which goes to C, which goes back to the same A". The rules can be matched to cycles in the data, but can not express that a cycle is required.

In [BR20], Bloem and De Rooij propose an approach to extract *motifs* from unlabeled graphs (networks). A motif is a structure that appear in the data more frequently than what would be expected from a random graph. Those structures are similar in the idea to patterns. Classically, selecting motifs requires generating many random graphs and counting the occurrences of a candidate motif in those random graphs w.r.t. the occurrences in the data, which takes a long time. However, by interpreting MDL encodings as probability distributions, the approach allows to find motifs without this generation step. The data is encoded by both a MDL encoding based on the candidate motif and a baseline MDL encoding that assumes the data to be a random graph. If the former gives a better description length, it means that the probability of the candidate is not what would be expected in a random graph, and as such it really is a motif.

Take-Home Message

The works in this thesis are part of the *pattern mining* domain and at the intersection of the *graph pattern mining* and *MDL-based pattern mining* sub-domains. Historically, pattern mining has been performed in a transactional setting, where patterns are sets of items that appear frequently together. Mining graph patterns require specific adaptations due to the nature of graphs: the support computation is not the same and depends on the type of the data, and many basic operations such as computing pattern occurrences become NP-complete. A well-known problem in pattern mining is the *pattern explosion problem*. That is, the number of frequent patterns tends to explode exponentially as the support threshold for a pattern to be considered frequent is lowered. This happens on transactional data as well as on graphs. In order to limit the number of patterns —to enable human analysis— approaches have been proposed that either use constraints of condensed representation to reduce the set of patterns that are shown to the user. However, this is often not enough, as the set of pattern remains large even on medium-sized datasets. More recently, approaches have been proposed that use the MDL principle to reduce the number of extracted patterns. Experiments show that they manage to extract small and descriptive sets of patterns. Few such approaches exist for mining graph patterns. Also, many of them are limited in the type of patterns that they extract. Therefore, there is a need for graph mining approaches that select a human-sized set of patterns, without putting limitations on the extracted patterns.

Chapter 4

Evaluating and Selecting Pattern Sets with the MDL Principle: GraphMDL

In this chapter we propose a *graph pattern mining* approach that uses the Minimum Description Length principle to tackle the *pattern explosion problem*. This problem is common in pattern mining, for approaches that mine *frequent patterns*. It denotes the fact that, as the threshold for a pattern to be considered frequent is lowered, the number of extracted patterns tends to increase exponentially. This poses a problem for human analysis, as —even on small and medium data— too many patterns are extracted for a human to be able to analyze them all.

The approach that we propose in this chapter —called GraphMDL— uses the MDL principle to select a small set of descriptive patterns from a set of candidate patterns generated on a labeled simple graph. Thus, patterns can be generated from graph data using classic graph mining algorithm, and then filtered with GraphMDL in order to generate a small set of patterns that allows human analysis. GraphMDL takes as input a labeled simple graph —the *data graph*— and a set of *candidate patterns* (labeled simple graphs as well), and returns as output a selected subset of them and a structure —the *rewritten graph*— that specifies how the data is decomposed using those patterns. The subset of candidates is selected with a greedy heuristic search using a MDL measure as a guide. An important characteristic of GraphMDL is that it evaluates pattern sets *as a whole*. Therefore when selecting a pattern set, the interactions between its patterns are taken into account. This marks a difference with other graph mining approaches such as SUBDUE [CH93] (which uses the MDL principle but evaluates patterns individually), or most constraint-based approaches [Zhu+07], which evaluate patterns independently one from another. GraphMDL also introduces the notion of *ports*, which allows to encode a data graph using a set of patterns in a lossless manner. This allows the user to exactly

know *how* the selected patterns interact with each other, and thus to better interpret the data. Thanks to the notion of ports, GraphMDL also does not impose any constraint on the shape of the patterns, nor on how their occurrences connect in the data graph. This is different from other MDL-based graph mining approaches such as VoG [Kou+15] and MeGS [Goe+16], which can only extract patterns with pre-defined shapes.

This chapter is organised as follows. Section 4.1 presents an overview of GraphMDL and gives a first definition of the concept of ports and rewritten graph. Section 4.2 formally defines all GraphMDL concepts from a MDL point of view and gives the formulas used to compute the MDL description length so as to evaluate pattern sets. Section 4.3 gives the search algorithm. Section 4.4 shows the experiments performed to evaluate GraphMDL. Finally, Section 4.5 presents GraphMDL Visualizer, a tool developed to allow the user to better understand GraphMDL results and perform an interactive analysis.

This chapter is based on work that has been published at conferences *Extraction et Gestion des Connaissances (EGC)* [BCF20a] and *Symposium on Intelligent Data Analysis (IDA)* [BCF20b]. Section 4.5 (GraphMDL Visualizer) is based on work published at the *Graph Embedding and Mining (GEM) workshop of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML-PKDD)* [BCF20c].

4.1 An Overview: Ports and Rewritten Graph

In order to use the MDL principle (see Section 2.2) one must define some elements: what is the data (D), what is a model (M), how a model is used to encode the data ($D|M$), and how to measure the description length of the model ($L(M)$) and the encoded data ($L(D|M)$). In this section we give an overview of the first three elements ($D, M, D|M$), and in the following section we give their formal definition, along with their description lengths.

The goal of GraphMDL is to select a set of patterns, which are labeled simple graphs, on some data, which is also a labeled simple graph. Therefore the *data* D is a labeled simple graph that we call the *data graph* G^D ($D = G^D$), and the *model* M a set $\mathcal{P}$ of labeled simple graphs, plus some additional information associated to each pattern that we detail in the next section. The intuition behind GraphMDL is that since the data and the patterns are both graphs, the data can be seen as a composition of occurrences of the patterns in the model. Encoding the data with the model ($D|M$) corresponds to making

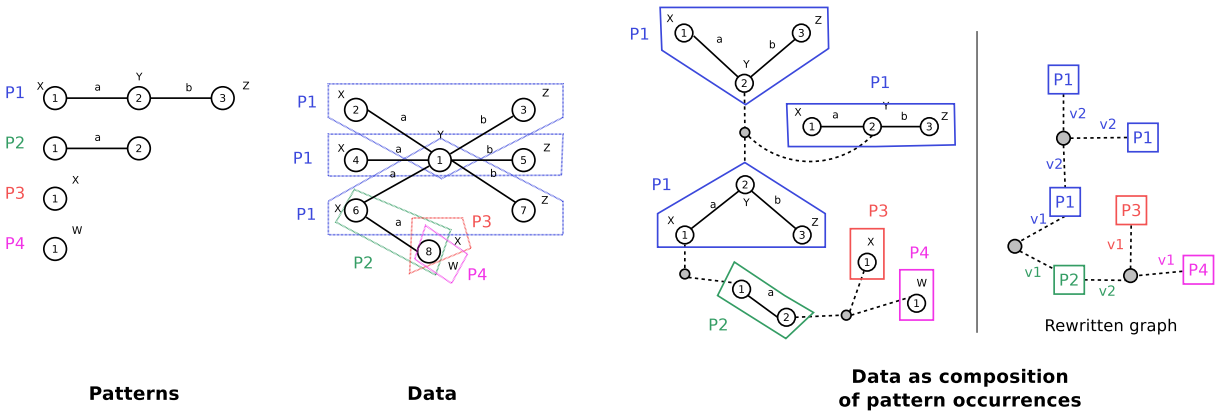


Figure 4.1 – How some labeled graph data can be represented as a composition of pattern occurrences. The rightmost representation corresponds to GraphMDL's *rewritten graph*. The second-to-last representation is given to better illustrate the concept.

this composition explicit. For that we use a structure that we call the *rewritten graph* G^R ($D|M = G^R$). Informally, a user analyzing the output of GraphMDL should be able to say "the data is composed of one occurrence of pattern P_1, connected to one occurrence of pattern P_2, which is itself connected to one occurrence of pattern P_3". More so, the user should be able to also say *how* those occurrences are connected: which vertices of each occurrence correspond to vertices of other occurrences. Ensuring both of those properties (which patterns there are and how they are connected) makes the encoding lossless, since the data can be perfectly reconstructed from the encoded data and the model. A lossless encoding is indeed a requirement in order to correctly use the MDL principle [Gal20].

Fig. 4.1 shows an example of how a data graph can be represented as a composition of pattern occurrences (i.e. a rewritten graph). We notice that vertices $1, 2, 3$ of the data graph (at the top) can correspond to an occurrence of pattern P_1. The same can be said for vertices $4, 1, 5$ (middle) and $6, 1, 7$ (bottom). Thus, vertices 1 to 7 of the data could be described as "three occurrences of pattern P_1". However, this information is not complete: not only we observe those three occurrences, we also observe that they all share their middle vertices, i.e. they all map vertex 2 of the pattern to vertex 1 of the data. Therefore, this part of the data is better described as "three occurrences of pattern P_1 which share their middle vertex". On the right of Fig. 4.1, this information is indicated in the rewritten graph by drawing a square for each of those pattern occurrences, and connecting all those squares to the same grey circle to represent the vertex that they all share. Those gray circles are the notion of *ports* that is at the heart of GraphMDL.

From the point of view of the data, ports are those data vertices that are used by more than one pattern occurrence. From the point of view of the pattern, ports correspond to those vertices that the pattern's occurrences share with other occurrences. That is to say, the vertices that are at the "interface" of the pattern, as opposed to non-port vertices which form the "internal structure" of the pattern. The main advantage of ports is allowing to represent the data in a lossless way, but they also allow freedom in the way that patterns can appear in the data. Any vertex of the pattern can potentially be a port, which means that a single pattern can have different vertices "at the interface" for each of its occurrences. For example, in Fig. 4.1 all occurrences of P_1 have their middle vertex as port, but one of them also has its first vertex as port (corresponding to data vertex 6, bottom left).

Note that while we allow pattern occurrences to overlap on data *vertices* (which is the whole notion of ports), we do not allow them to overlap on *edges*. For example in Fig. 4.1, once we decide that an occurrence of P_1 is used to describe vertices 1, 2, 3, no other pattern occurrence can be chosen to describe the edges between vertices 1 and 2 and vertices 1 and 3. This is done for two reasons. First, when using the MDL principle one should avoid encoding redundant information as much as possible. Once an edge has been described by a pattern occurrence, we should avoid describing it again using additional pattern occurrences. Second, if we allowed edge overlaps, virtually all vertices in the data graph would become ports. For example in Fig. 4.1, all edges with label a could be occurrences of pattern P_2. As such, if we allowed edge overlaps, any pattern whose structure contains an edge with label a (such as P_1) would always overlap with an occurrence of pattern P_2, requiring additional port vertices where such overlaps appear. More generally, if we allowed overlaps, as long as both a pattern and any of its sub-patterns appeared in the model, all occurrences of the larger pattern would require additional ports as they would overlap with occurrences of the sub-pattern. This would create a large amount of ports and negate their utility as a way to discern between vertices that are internal to the pattern or at its interface. For those reasons, we forbid edge overlaps between pattern occurrences. As a consequence, the order in which patterns occurrences are considered can have an impact on the structure of the rewritten graph. We describe the order that we use when describing the GraphMDL algorithm in Section 4.3.

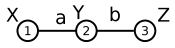
	G^P	C_P		$\lvert\pi_P\rvert$	π		C_π	
	Pattern structure	*Pattern usage*	Pattern code	*Pattern code length (bits)*	Port count	Port ID	*Port usage*	Port code	*Port code length (bits)*
P1		*3*	P1	*1*	2	v1 v2	*1* *3*	v1 v2	*2* *0.42*
P2		*1*	P2	*2.58*	2	v1 v2	*1* *1*	v1 v2	*1* *1*
P3		*1*	P3	*2.58*	1	v1	*1*		*0*
P4		*1*	P4	*2.58*	1	v1	*1*		*0*

Figure 4.2 – The CT associated to the data graph of Fig. 4.1. Values in light blue italic are for illustration only and are not part of the model.

4.2 MDL Description Lengths

In this section we formally define what the MDL model of GraphMDL is, along with the rewritten graph structure ($D|M$ in MDL terms), and the formulas used to compute their description lengths ($L(M)$ and $L(D|M)$ in MDL terms). All description length formulas can be found in Fig. 4.3 (for the model) and Fig. 4.5 (for the rewritten graph).

4.2.1 MDL Model: the Code Table

The MDL model used in GraphMDL is called the *Code Table* (CT). It has been inspired by the Krimp pattern mining approach [VLS11]. It is a table-like data structure, where each row represents a pattern and associated information needed for encoding the data with it. The differences with Krimp's code table are that the patterns are graphs, and that GraphMDL needs to store in it additional information about the ports of each pattern (which Krimp does not need because the itemset patterns it treats do not need ports).

Fig. 4.2 shows the CT associated to the data graph and patterns of Fig. 4.1. Every row of the CT is composed of three parts, and contains information about a pattern $P \in \mathcal{P}$. For instance the first row represents pattern P_1. The first part of a row is the graph G^P, which represents the structure of the pattern. The second part of a row is the code C_P associated to the pattern. This is a prefix code based on how many occurrences of this pattern are used in the rewritten graph. For instance pattern P_1 has 3 occurrences in the rewritten graph, and in total the rewritten graph contains 6 pattern occurrences, which gives a code of length $-\log(\frac{3}{6})$ bits $= 1$ bit. The third part of a row is the description of

$$L(M) = L(CT) = \sum_{\substack{P \in \mathcal{P} \\ \#occ(P) \neq 0}} \underbrace{L(G^P)}_{\text{structure}} + \underbrace{L(C_P)}_{\text{code (prefix code)}} + \underbrace{L(\Pi_P)}_{\text{ports}}$$

$$L(G^P) = \underbrace{L_{\mathbb{N}}(|V^P|)}_{\text{vertex count}} + \sum_{v \in V^P} [\ \underbrace{L_{V_\mathcal{L}}(v)}_{\text{vertex labels}} + \underbrace{L_{E_\mathcal{L}}(v)}_{\text{adjacent edges}}\ ]$$

$$L_{V_\mathcal{L}}(v) = \underbrace{L_{\mathbb{N}}(|\{(v,l) \in V^P_\mathcal{L}\}|)}_{\text{label count}} + \sum_{(v,l) \in V^P_\mathcal{L}} \underbrace{L_\mathcal{L}(l)}_{\text{label}}$$

$$L_{E_\mathcal{L}}(v) = \underbrace{\log(|V^P|)}_{\text{edge count}} + \sum_{(v,w,l) \in E^P_\mathcal{L}} [\underbrace{\log(|V^P|)}_{\text{destination}} + \underbrace{L_\mathcal{L}(l)}_{\text{label}}]$$

$$L(C_P) = -\log\left(\frac{\#occ(P)}{\sum_{P_i \in \mathcal{P}} \#occ(P_i)}\right)$$

where $\#occ$ is the number of occurrences in the rewritten graph

$$L(\Pi_P) = \underbrace{\log(|V^P| + 1)}_{\text{port count } |\Pi_P|} + \underbrace{\log(\binom{|V^P|}{|\Pi_P|})}_{\text{port ids}} + \sum_{\pi \in \Pi_P} \underbrace{L(C_\pi)}_{\text{port code (prefix code)}}$$

$$L(C_\pi) = -\log\left(\frac{\#occ(\pi)}{\sum_{\pi_i \in \Pi_P} \#occ(\pi_i)}\right)$$

where $\#occ$ is the number of occurrences in the rewritten graph

Figure 4.3 – MDL description length of the model used in GraphMDL. $\#occ$ represents the number of occurrences of an element (pattern or port) in the rewritten graph.

the ports Π_P of the pattern. This description is composed of: the number of ports $|\Pi_P|$; the identities $\pi \in \Pi_P$ of the ports (among all vertices of the pattern) ; and a prefix code C_π for each port, based on its usage in the rewritten graph w.r.t. all other ports of the pattern. For instance P_1 has 2 port vertices, its vertices 1 and 2, used respectively 1 and 3 times, giving respectively a code of 2 bits and 0.42 bits [1]. We call $\Pi = \bigcup_{P \in \mathcal{P}} \Pi_P$ the set of all ports of all patterns.

Description Length. The MDL model description length corresponds in GraphMDL to the description length of the Code Table, i.e. $L(M) = L(CT)$. The formulas pertaining to this description length can be found in Fig. 4.3. The description length of a code table ($L(CT)$) is the sum of the description lengths of its rows, skipping rows with unused patterns (since they do not contribute to the encoding). The description length of each row is the sum of the description length of its parts: the pattern's structure, the pattern's

1. Note that MDL approaches deal with *theoretical* code lengths, which do not need to be integers.

code, and the pattern's ports.

The structure of a pattern is a labeled graph $G^P = (V^P, V^P_{\mathcal{L}}, E^P_{\mathcal{L}})$. To encode it ($L(G^P)$) we first encode its vertex count. Since a pattern can have any number of vertices, we encode it with a universal integer encoding ($L_{\mathbb{N}}(|V^P|)$). Then, for each vertex $v \in V^P$ we encode its vertex labels and adjacent edges. To encode the labels of a vertex ($L_{V_{\mathcal{L}}}(v)$) we specify their number first using a universal integer encoding, then the label themselves ($L_{\mathcal{L}}(l)$, see below). To encode the edges around a vertex ($L_{E_{\mathcal{L}}}(v)$) we fist specify how many there are (between 0 and $|V^P| - 1$ for simple graphs), then for each one the other vertex to which it is connected (one of the $|V^P|$ pattern vertices) and its label ($L_{\mathcal{L}}(l)$). To encode vertex and edge labels ($L_{\mathcal{L}}(l)$) we use a prefix code based on the number of occurrences that the label symbol has in the data graph w.r.t. other label symbols of the same type (i.e. vertex label symbols and edge label symbols are counted separately). Note that since this code is based on the data graph it does not change from one CT to another. When working with undirected graphs, each edge is only encoded with one of its ends. The description length that we propose is independent of which of the two ends describes an edge, so an arbitrary decision is made (e.g. in our implementation we encode an edge (v, w) on v if $v < w$). The description length of this encoding for the pattern structure is also independent of vertex numbers, which means that it is invariant w.r.t. isomoprhisms.

The second part of a row is the pattern's prefix code C_P. Its length $L(C_P)$ is the length of a prefix code based on the number of occurrences of this pattern in the rewritten graph w.r.t. all pattern occurrences in the rewritten graph.

The last part of a row is the description of the pattern's ports ($L(\Pi_P)$). First, we encode how many of the pattern vertices are ports (between 0 and $|V^P|$). Then we specify which vertices are ports: if there are k ports, then there are $\binom{|V^P|}{k}$ possibilities. Finally, each port $\pi \in \Pi_P$ has a prefix code based on its usage w.r.t. all ports of the pattern. Note that since port information increases the description length, we expect our approach to select patterns with few ports. We consider this to be good, as it will favour models which give a data encoding where the different patterns only have few "interface points" (i.e. ports). In our opinion, such models are more easily interpretable by a human analyst than models where patterns are very interconnected through many ports, giving a "tangled" representation of the data.

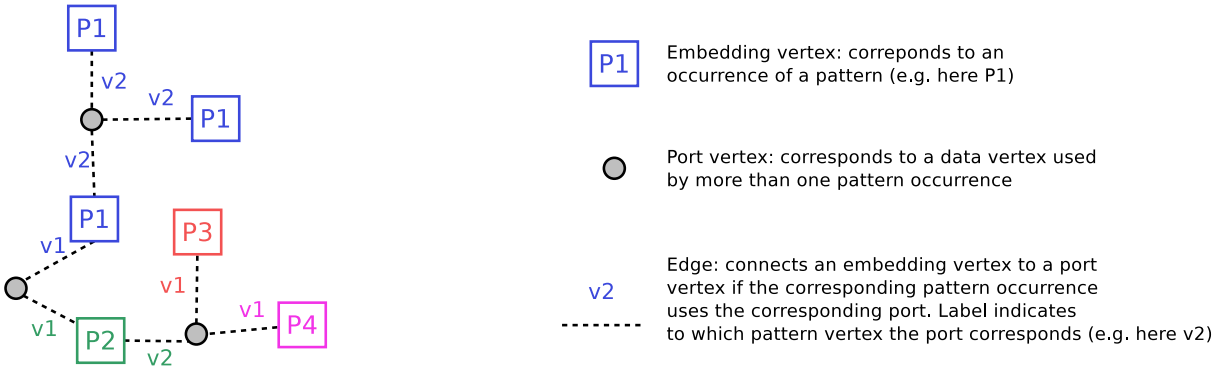


Figure 4.4 – The rewritten graph associated to the data graph of Fig. 4.1 and CT of Fig. 4.2.

4.2.2 Encoded Data: the Rewritten Graph

In Section 4.1 we presented how GraphMDL represents the data as a composition of pattern occurrences connected through ports. The *rewritten graph* structure formally describe that representation. It corresponds to the "data encoded with the model" in MDL terms ($D|M$ in formulas). A rewritten graph is a labeled simple graph $G^R = (V^R, V^R_{\mathcal{L}}, E^R_{\mathcal{L}})$, with some peculiarities:

1. The vertices can be separated in two distinct subsets: $V^R = V^R_{emb} \cup V^R_{port}$. We call V^R_{emb} *embedding vertices* and V^R_{port} *port vertices*.
2. Embedding vertices have exactly one label which represents a CT pattern, port vertices have no labels: $V^R_{\mathcal{L}} \in V^R_{emb} \to \mathcal{P}$.
3. Edges can exist between embedding vertices and port vertices, such edges have a label that represents a CT pattern port: $E^R_{\mathcal{L}} \subseteq V^R_{emb} \times V^R_{port} \times \Pi$.

Each *embedding vertex* $v_e \in V^R_{emb}$ represents a single pattern occurrence. Its label represents which of the CT patterns it instantiates. Each *port vertex* represents one port, i.e. a data vertex shared by multiple pattern occurrences. If a pattern occurrence uses a port, an edge exists in the rewritten graph from the corresponding embedding vertex to the corresponding port vertex. That edge is labeled with a label indicating which of the pattern's ports correspond to the port vertex.

An example of rewritten graph is shown in Fig. 4.4, which corresponds to the rewritten graph given by the CT of Fig. 4.2 over the data of Fig. 4.1. We represent embedding vertices as squares with their corresponding label written inside, port vertices as circles,

$$L(D|M) = L(G^R) = \underbrace{L_{\mathbb{N}}(|V^R_{port}|)}_{\text{port vertices count}} + \sum_{v_e \in V^R_{emb}} L_{emb}(v_e)$$

$$L_{emb}(v_e)\left(\text{with } P = V^R_{\mathcal{L}}(v_e)\right) = \underbrace{L(C_P)}_{\text{pattern prefix code}} + \underbrace{\log(|\Pi_P|+1)}_{\text{edge count}} + \sum_{(v,w,l)\in E^R_{\mathcal{L}}} \underbrace{\log(|V^R_{port}|)}_{\text{port vertex id}} + \underbrace{L(C_\pi)}_{\text{port prefix code}} .$$

Figure 4.5 – MDL description lengths used in GraphMDL for the encoded data. See Fig, 4.3 for the definition of $L(C_P)$ and $L(C_\pi)$.

and edges from embedding vertices to port vertices as dotted lines with their label alongside. For instance the top of the image shows three embedding vertices of pattern P_1 all connected to the same port vertex. For all of those three embeddings, the port vertex corresponds to vertex v_2 of the pattern.

Description Length. The MDL description length of the data encoded with the model ($L(D|M)$) corresponds in GraphMDL to the description length of the rewritten graph ($L(G^R)$). The formulas pertaining to this description length are shown in Fig. 4.5. To describe port vertices only their number $|V^R_{port}|$ is needed since they have no other distinguishing features. A universal integer encoding is used for that ($L_{\mathbb{N}}(|V^R_{port}|)$). Then each embedding vertex is encoded as well as the edges connecting it to port vertices ($L_{emb}(v_e)$). In order to encode an embedding vertex v_e, we must first encode which pattern P it represents (corresponding to its label $V^R_{\mathcal{L}}(v_e)$). The pattern prefix code C_P defined in the CT is used for that. Then, in order to encode its edges we must first indicate how many there are, i.e. how many ports this specific pattern occurrence has (between 0 and $|\Pi_P|$, i.e. all ports of the pattern). We then encode for each edge starting from v_e the port edge to which it is connected (one of the $|V^R_{port}|$ port vertices), and to which pattern port π it correspond (using the port code C_π defined in the CT).

4.3 Performing the Search: GraphMDL Algorithm

In the previous sections we presented the different MDL definitions that GraphMDL uses to evaluate code tables, i.e. sets of patterns. A naive algorithm for finding the best (in the MDL sense) pattern set could be to create a CT for every possible subset of candidates

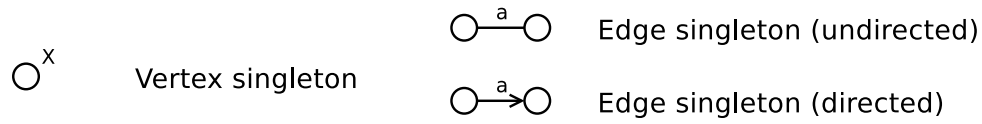


Figure 4.6 – GraphMDL singleton patterns. For the edge singleton, both the version used on undirected graphs and the one used on directed graphs are shown.

Algorithm 1 The GraphMDL algorithm.

Require: A data graph D, a set of candidate patterns $\mathcal{C}$

1: Compute initial singleton code table CT_0 from labels in D
2: $CT \leftarrow CT_0$
3: **while** $\mathcal{C} \neq \emptyset$ **do**
4: $\quad P \leftarrow$ SelectAndRemoveCandidate($\mathcal{C}$) ▷ Heuristic
5: $\quad CT' \leftarrow$ InsertInCT(P, CT) ▷ Heuristic
6: $\quad$ **if** $L(CT', D) < L(CT, D)$ **then**
7: $\quad\quad$ **if** pruning enabled **then**
8: $\quad\quad\quad CT \leftarrow$Pruning($CT, CT'$)
9: $\quad\quad$ **else**
10: $\quad\quad\quad CT \leftarrow CT'$
11: $\quad\quad$ **end if**
12: $\quad$ **end if**
13: **end while**
14: **return** CT

and retain the one yielding the smallest description length. However, such an approach is often infeasible because of the large amount of possible subsets. That is why GraphMDL performs a greedy heuristic search algorithm, inspired from the Krimp algorithm [VLS11].

The GraphMDL algorithm is presented in Alg. 1. It starts by computing an initial code table CT_0 containing a singleton pattern for each label of the data graph (line 1). Singletons are the most basic patterns that GraphMDL can handle. A vertex singleton is a pattern composed of a single vertex with a single label, while an edge singleton is a pattern composed of two unlabeled vertices connected by a single labeled edge. Fig. 4.6 shows an example of a singleton pattern of each type. Including all singleton patterns in the CT ensures that the data can always be encoded, since any label not covered by non-singleton patterns can always be covered using singletons. After that initialisation, the GraphMDL algorithm consists in a loop that treats all candidate patterns one after the other (lines 3-13). In each iteration of the loop, one candidate is selected from the set of candidates (line 4) and inserted in the CT (line 5). If the description length of this new CT is better than the previous one, the candidate is kept in the CT, otherwise it is

discarded (lines 6-12). When all candidates have been tested, the best CT that has been found is returned (line 14).

Like Krimp, GraphMDL implements an optional *post-acceptance pruning* mechanism (line 7-11). When a code table is found that improves the description length, GraphMDL can verify if removing some patterns from the CT can improve the description length. All patterns whose usage has diminished compared to the previous code table are considered for pruning one after the other (in increasing usage order). GraphMDL iterates through them and tries to remove them from the CT, and if the description length improves after their removal they are permanently removed (otherwise they are kept in the CT). If the usage of some patterns diminishes during the pruning process, they are also added to the list of pruning candidates. The intuition behind this mechanism is that patterns can become "obsolete" once a better pattern has been added to the CT, and removing them avoids cluttering the code table with old not-very-useful patterns, thus reducing the overall description length.

Two heuristics are used to guide the GraphMDL search:

- The SelectAndRemoveCandidate function (line 4) decides in which order candidate patterns are selected from the candidate pattern set $\mathcal{C}$. Since GraphMDL does not try all possible combinations of candidates but only tests the inclusion of each candidate once, the order in which candidates are added to the code table can have an impact over the explored solutions. We first order candidates by descending image based support [BN08] in the data graph, i.e. the pattern that appears the most frequently in the data will be chosen first. Ties are broken by descending label count (vertex labels plus edge labels). Further ties are broken with an arbitrary order defined on the patterns.
- The InsertInCT function (line 5) inserts a candidate pattern in the CT. The position of a pattern in the CT is important: when the rewritten graph is computed, all embeddings of the pattern at the top of the CT are considered first, followed by the embeddings of the second pattern, and so on. If a pattern embedding overlaps on an edge with another embedding that has already been included in the rewritten graph, it is not added to the rewritten graph (see Section 4.1). Therefore patterns at the top of the CT have more chances of being used in the rewritten graph. We insert patterns in the CT such that the patterns are first ordered by descending number of labels, then by descending image based support [BN08] in the data graph, with

Table 4.1 – Characteristics of the datasets used in the experiments.

Dataset	Graph count	$\|V\|$	$\|V_{\mathcal{L}}\|$	$\|E_{\mathcal{L}}\|/2$	Vertex label symbols	Edge label symbols
AIDS-CA	423	17k	17k	18k	21	3
AIDS-CM	1082	34k	34k	37k	26	3
Mutag	125	2k	2k	3k	7	4
PTC-FM	143	2k	2k	2k	18	4
PTC-FR	121	2k	2k	2k	19	4
PTC-MM	129	2k	2k	2k	20	4
PTC-MR	152	2k	2k	2k	18	4
UD-PUD-En	1000	21k	21k	20k	17	46

ties broken by an arbitrary order defined on the patterns.

We developed several versions of each heuristic (taking inspiration from Krimp heuristics), and experiments outlined the combination presented here as the most efficient.

4.4 Experiments

In order to evaluate our proposal, we developed a prototype of GraphMDL. The prototype was developed in Java 1.8 and is available as a git repository [2].

We conducted experiments on four different datasets (some of them having several sub-datasets). The AIDS [3] (AIDS-CA and AIDS-CM), Mutag [4], and PTC [5] (PTC-FM, PTC-FR, PTC-MM and PTC-MR) datasets are molecular datasets, where vertices are atoms and edges are bonds. Those datasets have few label symbols and many cycles. The UD-PUD-En dataset is from the Universal Dependencies project [6]. Each graph of the dataset represents a sentence: vertices are words, vertex labels are part-of-speech tags, and edge labels are dependency relationships. This dataset correspond to English sentences of the PUD corpus (which exists in many different languages). This dataset has many label symbols and no cycles. For datasets that have both a positive and a negative class, only the positive class was used for the evaluation. All of the datasets are collections of graphs. Since GraphMDL works on a single data graph instead of a collection, we aggregated collections into a single graph, each graph of the collection simply becoming a connected component in the single graph.

2. `https://gitlab.inria.fr/fbariatt/graphmdl`
3. `https://wiki.nci.nih.gov/display/NCIDTPdata/AIDS+Antiviral+Screen+Data`
4. `http://networkrepository.com/Mutag.php`
5. `http://networkrepository.com/PTC-FM.php` (replace FM with FR, MM or MR for the others)
6. `https://universaldependencies.org/`

The characteristics of the datasets that we use are presented in Table 4.1. For each dataset we report: the number of graphs in the collection, the total amount of vertices $|V|$, the total amount of vertex labels $|V_{\mathcal{L}}|$, the total amount of edges $|E_{\mathcal{L}}|/2$ with each undirected edge counted only in one direction, the number of vertex label symbols and the number of edge label symbols.

We generate candidate patterns using an implementation of the gSpan algorithm [YH02] found on the website of one of its authors [7]. This approach needs a *support* parameter: the lower the support, the less frequent the generated patterns can be (and therefore the more there are). We tested different values, going as low as possible while having the experiments terminate in less than 4 hours. Since this gSpan implementation only treats undirected graphs, we considered the UD-PUD-En dataset as being undirected (the other datasets are already undirected).

4.4.1 Quantitative Evaluation

Table 4.2 presents the results obtained when running GraphMDL on the different datasets. The first columns of the table report respectively: (1) the dataset ; (2) the gSpan support used for the experiment ; (3) the number of candidate patterns that gSpan generated for that support value ; and (4) the runtime of GraphMDL (gSpan runtime is negligible). Note that for low values of support, the number of candidate patterns explodes, which makes the runtime explode as well, since GraphMDL needs to treat all of them. This is the reason why on some datasets the experiments do not approach the 4h time limit: for all the lower values of support that we tested, the number of candidates was so high that GraphMDL ran for far longer than the time limit.

Number of selected patterns. The fifth column of Table 4.2 ($|CT_{GraphMDL}|$) reports the number of patterns in the best code table found by GraphMDL. We observe that our approach selects a significantly smaller set of patterns compared to the number of candidates. This is particularly remarkable for experiments ran with small support thresholds, where there are thousands less patterns in the CT than in the set of candidates. For example on the AIDS-CA dataset for a support threshold of 10%, gSpan generates more than $2 \cdot 10^4$ patterns, and GraphMDL selects only around 150 of them. We also note that when the support threshold decreases, the number of candidate patterns explodes, but the number of patterns in the CT does not increase by much. For example on the AIDS-CA

7. `https://sites.cs.ucsb.edu/~xyan/software/gSpan.htm`

Table 4.2 – Experimental results for different candidate sets. $L\%$ is the compression ratio $\frac{L(CT_{GraphMDL},D)}{L(CT_0,D)}$

Dataset	gSpan support	Candidate count	Runtime	$\lvert CT_{GraphMDL}\rvert$	$L\%$	Median label count	Median port count
AIDS-CA	20%	2190	12m	112	24.4%	9	3
AIDS-CA	15%	7862	39m	130	21.5%	9	4
AIDS-CA	10%	20590	1h52m	148	19.2%	11	3
AIDS-CM	20%	429	15m	110	28.8%	7	4
AIDS-CM	15%	774	21m	130	27.5%	9	4
AIDS-CM	10%	2049	42m	145	24.8%	9	4
AIDS-CM	5%	9937	2h50m	212	20.7%	9	4
Mutag	40%	656	4m	26	18.9%	15	4
Mutag	35%	1766	9m	29	17.6%	17	4
Mutag	30%	5395	1h13m	29	16.7%	16	4
Mutag	25%	15258	2h38m	29	15.4%	16	4
PTC-FM	5%	518	1m	57	26.9%	7	2
PTC-FM	3%	5096	5m	69	23.6%	7	3
PTC-FM	2.5%	16286	20m	69	23.4%	7	3
PTC-FM	1%	80598	1h42m	77	22.9%	7	2
PTC-FR	5%	1437	2m	57	27.3%	7	3
PTC-FR	3%	20254	28m	70	23.4%	7	2
PTC-MM	5%	1010	2m	62	26.6%	7	2
PTC-MM	3%	17646	29m	75	23.8%	7	2
PTC-MM	1%	78359	2h09m	75	23.7%	7	2
PTC-MR	5%	1783	3m	67	27.0%	7	3
PTC-MR	3%	16180	19m	78	23.1%	7	2
UD-PUD-En	10%	150	1m	165	39.5%	5	2
UD-PUD-En	5%	444	3m	250	34.4%	5	2
UD-PUD-En	1%	6006	21m	522	28.1%	5	2
UD-PUD-En	0.5%	21131	1h42mm	647	26.8%	5	2

dataset, as the support threshold goes from 20% to 10%, the number of patterns generated by gSpan goes from $2 \cdot 10^3$ to $2 \cdot 10^4$, while the size of the CT selected by GraphMDL only goes from around 100 patterns to around 150. We imagine that this is because the additional candidate patterns generated by gSpan contain a lot of redundancy that our approach avoids.

Description length. The sixth column of Table 4.2 reports the *compression ratio* $L\% = \frac{L(CT_{GraphMDL},D)}{L(CT_0,D)}$ between the description length given by the code table found by GraphMDL and the one given by the initial singleton-only code table CT_0. We observe that the description length given by the CTs found by our approach is between 15% and 39% of the one given by the initial baseline code table. This indicates that GraphMDL succeeds in finding regularities in the structure of the data that allow to compress it (w.r.t. CT_0 which is only based on the frequency of the label symbols in the data). We also observe

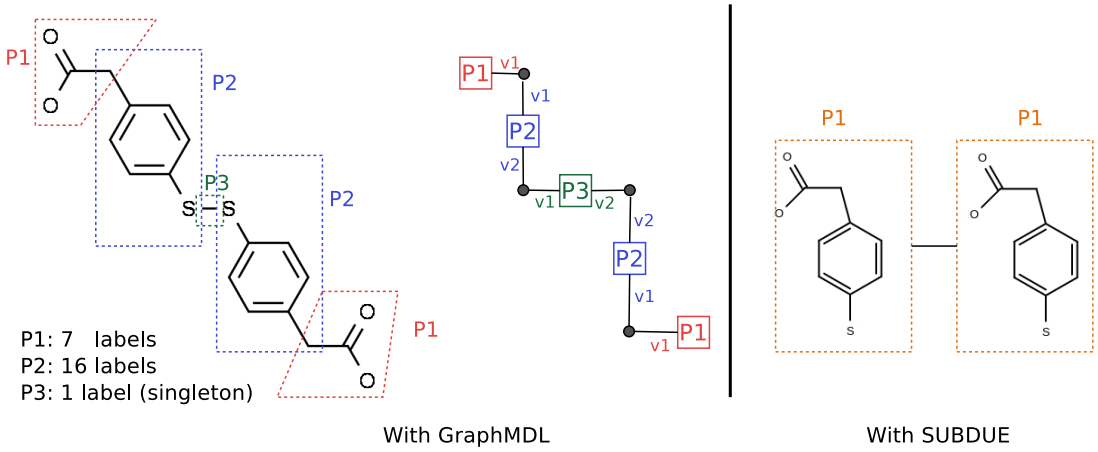


Figure 4.7 – How GraphMDL (left) and SUBDUE (right) encode one of AIDS-CM graphs.

that description lengths are smaller when the number of candidates is higher: this may be because with more candidates, there are more chances of finding "good" candidates that allow to better reduce description lengths.

Pattern characteristics. The last two columns of Table 4.2 report respectively the median number of labels in the selected patterns and the median number of ports. We observe that GraphMDL is not limited to small patterns, but can find patterns of non-trivial size. We also observe that most patterns have few ports, which shows that GraphMDL manages to find models in which the original graph is described as a set of components without many connections between them. We believe that a human will interpret such a model with more ease, as opposed to a model composed of "entangled" components.

4.4.2 Qualitative Evaluation

Rewritten graph interpretation. Fig. 4.7 shows how GraphMDL uses patterns selected on the AIDS-CM dataset to encode one of the graphs of the dataset. It illustrates the key idea behind our approach: find a set of patterns so that each one describes part of the data, and such that their occurrences connected via ports describe the whole data. We observe that GraphMDL can select and use both big patterns (such as P_2), describing big chunks of data, and small patterns (such as P_3, edge singleton), that can form bridges between pattern occurrences. Big patterns increase the description length of the CT, but describe more of the data in a single occurrence, whereas small patterns do the opposite. Following

Table 4.3 – Classification accuracies. Results of methods marked with * are from [RBB19].

Algorithm \ Dataset	AIDS-CA/CI	UD-PUD-En/De/Fr/It	Mutag	PTC-MR	PTC-FR
Baseline-Largest	50.01 ± 0.03	25.00 ± 0.00	66.50 ± 0.00	55.80 ± 0.00	65.50 ± 0.00
Baseline-Labels	50.91 ± 0.03	25.10 ± 0.00	66.50 ± 0.00	57.00 ± 0.00	65.80 ± 0.00
GraphMDL	71.61 ± 0.96	69.45 ± 0.27	80.79 ± 1.51	57.38 ± 1.68	62.70 ± 1.86
WL*	N/A	N/A	87.26 ± 1.42	63.12 ± 1.44	67.64 ± 0.74
P-WL*	N/A	N/A	86.10 ± 1.37	63.07 ± 1.68	67.30 ± 1.50
P-WL-C*	N/A	N/A	90.51 ± 1.34	64.02 ± 0.82	67.15 ± 1.09
RetGK*	N/A	N/A	90.30 ± 1.10	62.15 ± 1.60	67.80 ± 1.10

the MDL principle, our approach finds a good balance between the two types of patterns. It is interesting to note that pattern P_1 corresponds to the carboxylic acid functional group, common in organic chemistry. GraphMDL selected this pattern without any prior knowledge of chemistry, solely because it was interesting from a MDL point of view to have it in the CT (probably because it appears often as a "module" connected through one port to the rest of the data). The rewritten graph structure highlights well how each pattern is connected to the others in order to represent the data, which of its vertices are internal and which are at its borders, i.e. the "interface" to other patterns.

Comparison with SUBDUE. On the right of Fig. 4.7 we can observe the encoding found by SUBDUE on the same graph. The main disadvantage of SUBDUE is information loss: we can see that the data is composed of two occurrences of pattern P_1, but not how these two occurrences are connected. Thanks to the notion of ports, GraphMDL does not suffer from this problem: the user can exactly know which atoms lie at the boundary of each pattern occurrence.

Assessing patterns through classification. We showed in the previous experiments that GraphMDL manages to reduce the amount of patterns, and that the introduction of ports can give the user a better understanding about how the selected patterns compose the data. We now ask ourselves if the extracted patterns are *characteristic* of the data. To evaluate this aspect, we adopt the classification approach used by Krimp [VLS11]. We apply GraphMDL independently on each class of a multi-class dataset, and then use the resulting CTs to classify each graph: we encode it with each of the CTs, and classify it in the class whose CT yields the smallest description length $L(D|M)$. Since GraphMDL is not designed with the goal of classification in mind, we do not expect it to outperform existing classifiers. In particular, note that patterns are selected on each class independently of other classes and the fact that they will be used to differentiate between classes. Indeed, GraphMDL follows a descriptive approach whereas classifiers generally follow

a discriminative approach. Table 4.3 presents the results of this experiment. We compare GraphMDL with graph classification algorithms found in the literature [RBB19], and two baselines. Baseline-Largest classifies all graphs as belonging to the largest class. Baseline-Labels does the same, unless it finds that the graph to classify contains a label symbol that is unique to one of the classes. The AIDS-CA/CI dataset is composed of the CA class of the AIDS dataset and a same-size same-labels random sample from the CI class (corresponding to negative examples). The UD-PUD-En/De/Fr/It dataset is composed of the graphs from the UD-PUD-En dataset, corresponding to sentences from the English version of the PUD corpus, and their counterpart in German, French, and Italian. Each language is considered as a class, and all classes contain the same sentences, albeit in different languages. The other datasets [8] are from [RBB19]. We performed a 10-fold validation repeated 10 times and report average accuracies and standard deviations.

GraphMDL clearly outperforms the baseline on the AIDS, UD-PUD, and Mutag datasets, but is only comparable to the baseline for the PTC datasets. On Mutag, our approach is less accurate than other classifiers but closer to them than to the baseline. On the PTC datasets, we hypothesize that the learned descriptions are not discriminative w.r.t. the chosen classes, although they are characteristic enough to reduce description length. Nonetheless results are still better than random guessing (accuracy would be 50%). Despite those mitigated results, the experiment shows that the selected patterns are descriptive enough of their class that they have a positive impact in a graph classification task. An interesting point of GraphMDL classification that is specific to our approach is that it is explainable: the user can look at how the patterns of the two classes encode a graph (similarly to Fig. 4.7) and understand *why* one class is chosen over another.

4.5 The GraphMDL Visualizer Tool for Interactive Visualization of Patterns

The overarching goal of pattern mining approaches such as GraphMDL is to help users analyze data. However, they can only be truly helpful in this task if the users can actually understand their outputs. The user should be able to understand what the extracted patterns look like, how they appear in the data, and how they relate with other patterns. Thus, visualization techniques play a great role in pattern mining, bridging the

8. For concision, we do not report on PTC-{MM,FM}, they yield results similar to PTC-{MR,FR}

gap between the algorithms and the users [JK19].

In this section we present GraphMDL Visualizer, a tool that presents to the users an *interactive visualization* of GraphMDL results. We constructed GraphMDL Visualizer around the needs and expected behavior of users when analyzing GraphMDL results. The tool has different views, ranging from more general (distribution of pattern characteristics), to more specific (visualization of a specific pattern). It is also highly interactive, allowing the users to navigate between the different views and customize them through simple mouse clicks. Since GraphMDL introduces the notion of ports and rewritten graph, a custom tool is needed for visualizing its results in a way that can take into account those notions, and thus present all information contained in them.

GraphMDL Visualizer is available online at `http://graphmdl-viz.irisa.fr/`. The results of the experimental evaluation presented in Section 4.4 as well as the ones in Chapters 5 and 6 are also available on that page, and they can be visualized with the tool.

Implementation. GraphMDL Visualizer presents itself as an interactive web page that the user can open in their web browser. Web applications have a native support for user interactivity (e.g. reacting when the user clicks or moves the mouse), and can be used without needing a complex installation process by the user. The application has been developed using the Javascript framework VueJS [9], the Javascript data visualization library D3.js [10], and the CSS library Bootstrap [11]. The source code is available as a git repository [12].

In order to be able to transmit GraphMDL results to GraphMDL Visualizer through a web browser, we modified the GraphMDL implementation so that all of its results are given as a JSON file. This file can then simply be uploaded in the GraphMDL Visualizer page using the usual file upload mechanism of the web browser. This simplifies the user experience, since the user does not need to perform post-processing between running GraphMDL and visualizing its results. It also makes GraphMDL Visualizer potentially capable of visualizing results of other pattern mining approaches, as long as they are converted to the same format.

9. `https://vuejs.org/`
10. `https://d3js.org/`
11. `https://getbootstrap.com/`
12. `https://gitlab.inria.fr/fbariatt/graphmdl-visualizer`

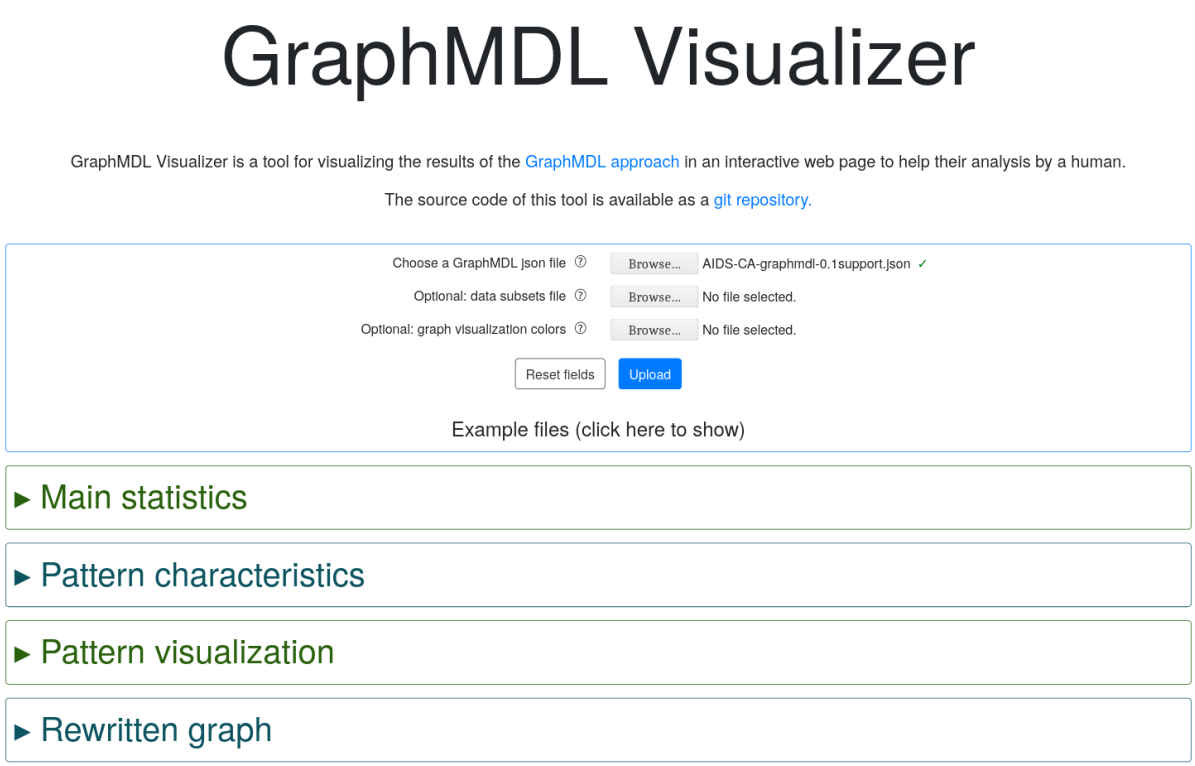


Figure 4.8 – The main structure of GraphMDL Visualizer after a user has uploaded a GraphMDL result file.

General structure. We identified 4 steps in the interaction between the user and the algorithm's output, and designed GraphMDL Visualizer around them.

Step 1. Broad view of statistical information about the patterns, to answer questions such as: *How many patterns have been extracted? Do the extracted patterns have many occurrences in the rewritten graph? How many vertices and edges do they have?*

Step 2. More precise view of the characteristics of the individual patterns, to answer questions such as: *What is the largest pattern? Which one has the most occurrences in the rewritten graph? How many vertices and edges do they have? Which of the pattern vertices are used as ports?* This allows the user to identify some patterns that interest them.

Step 3. Visualization of the structure of the pattern(s) identified during the previous step. Graphs are complex structures that are usually best understood when plotted graphically, thus the need for the visualization of a pattern's structure.

Step 4. Visualization of the rewritten graph, showing how the patterns are used to

describe the data, to answer questions such as: *Which are the neighbours of the pattern(s) selected at the previous step(s)? How a certain subset of the data is described in terms of pattern occurrences?*

During step 4, the user can identify new patterns that pique their interest, and navigates to a different step, or starts over with a different goal in mind. We found that this back-and-forth navigation was very typical, and therefore a good visualization should be interactive and allow the user to easily move from one view to another.

Fig. 4.8 shows the general structure of GraphMDL Visualizer after the user has uploaded a GraphMDL result file. As a running example in this section, we imagine that the user used the result file generated by running GraphMDL on the AIDS-CA dataset, with a gSpan support threshold of 10% (see Section 4.4). The application is composed of 4 main blocks, which corresponds to the 4 steps of the user interaction that we described above.

Step 1: main statistics. The "Main statistics" block, shown in Fig. 4.9, provides a general view of the patterns extracted by GraphMDL.

At the top, some general information is given, notably the number of patterns in GraphMDL's code table, the size of the rewritten graph (number of embedding vertices and port vertices), and MDL description lengths.

The bottom part of the block contains 5 plots. These plots represent the distribution of pattern occurrences in the rewritten graph (called "usage" in the visualization), the distribution of pattern sizes (vertex count, edge count, label count), and the distribution of the number of ports per pattern. An advantage of using an interactive web page as a visualization tool w.r.t. static images is that the user can zoom and drag those plots to highlight the areas that interest them the most. Hovering a bar of the bar plot also shows the exact height of the bar. Each plot also has a button to toggle between a bar plot or a box plot for representing the data. In this way the user can switch between the two visualizations, depending on whether they want to observe the exact distribution or its main characteristics (median, percentiles). For example in Fig. 4.9, we can observe in the first plot that 75% of the patterns appear 33 times or less in the rewritten graph but some of them appear up to 400 times. In the second, third and fourth plot we can observe that patterns have a median size of 6 vertices and 5 edges, or 11 labels. Finally in the last plot we observe that patterns tend to have few ports, but a small number of them have many ports.

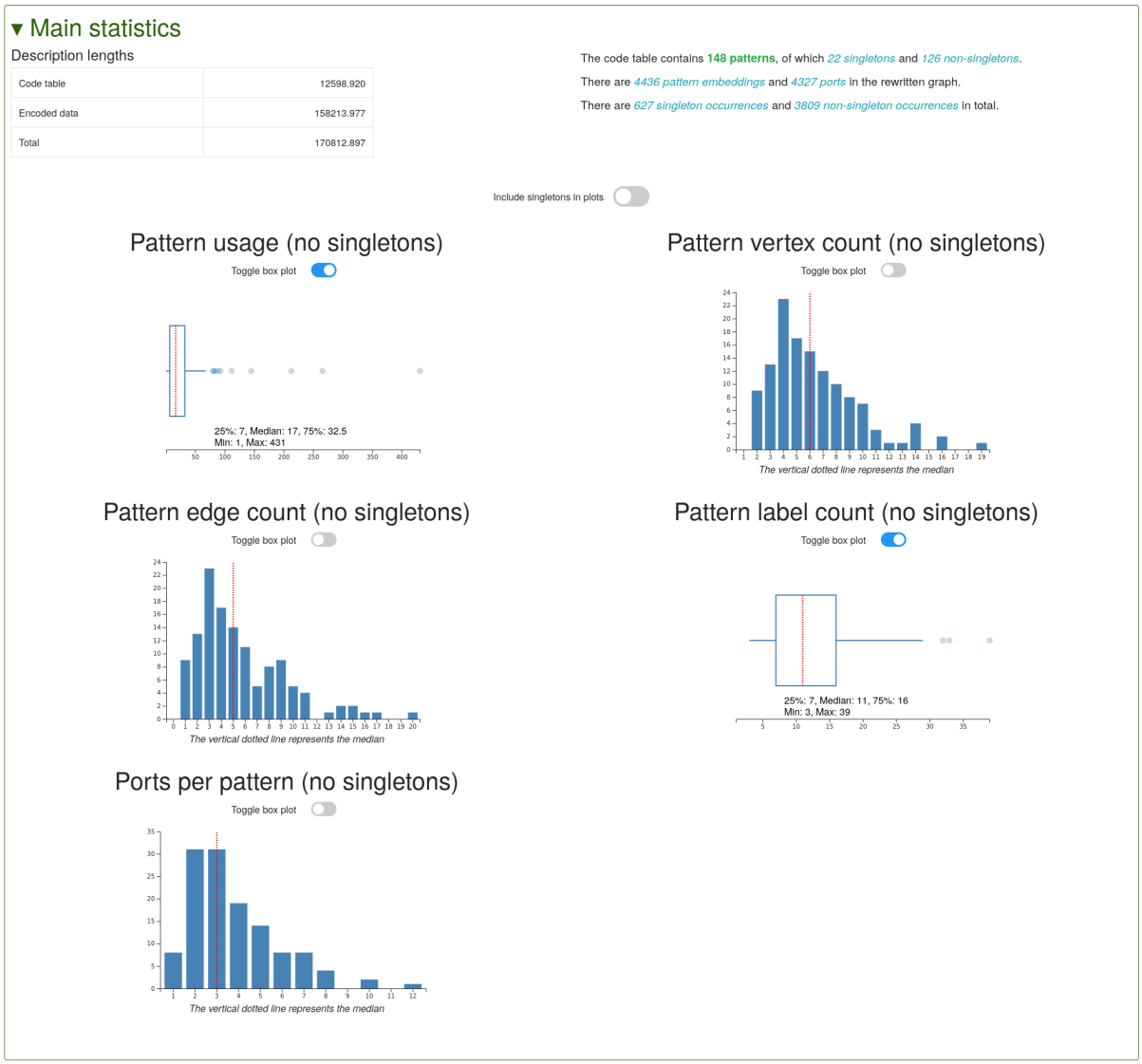
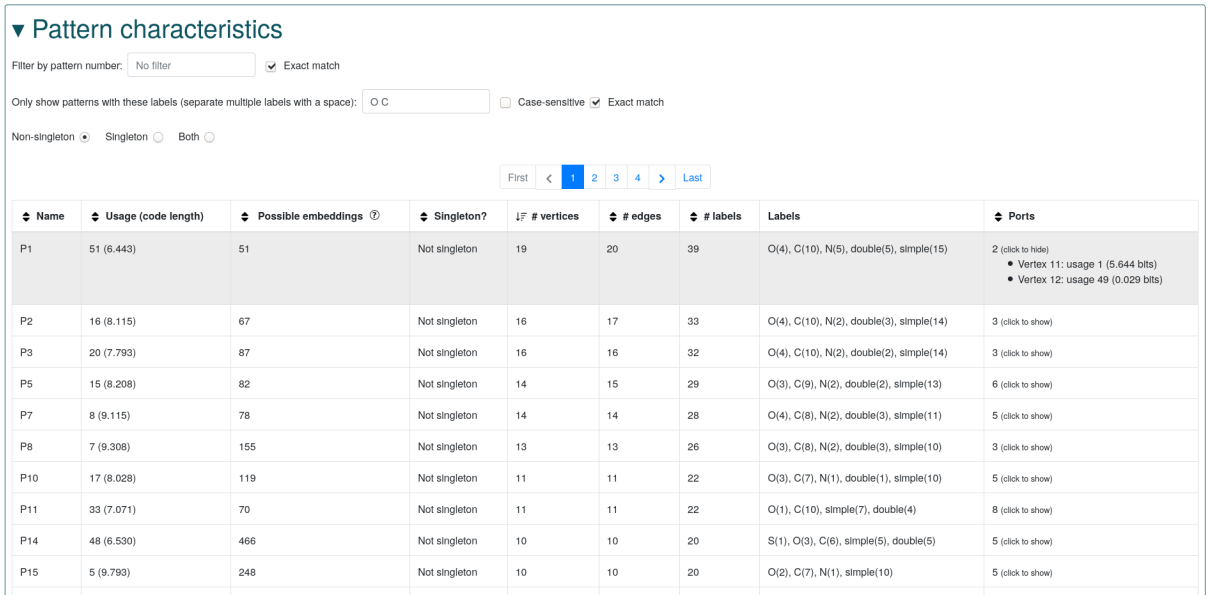


Figure 4.9 – The "Main statistics" block of GraphMDL Visualizer. This blocks shows the global characteristics of the selected patterns, without focusing on the individual patterns. In this image the user switched the first and fourth plots to box plots.



Name	Usage (code length)	Possible embeddings	Singleton?	# vertices	# edges	# labels	Labels	Ports
P1	51 (6.443)	51	Not singleton	19	20	39	O(4), C(10), N(5), double(5), simple(15)	2 (click to hide) • Vertex 11: usage 1 (5.644 bits) • Vertex 12: usage 49 (0.029 bits)
P2	16 (8.115)	67	Not singleton	16	17	33	O(4), C(10), N(2), double(3), simple(14)	3 (click to show)
P3	20 (7.793)	87	Not singleton	16	16	32	O(4), C(10), N(2), double(2), simple(14)	3 (click to show)
P5	15 (8.208)	82	Not singleton	14	15	29	O(3), C(9), N(2), double(2), simple(13)	6 (click to show)
P7	8 (9.115)	78	Not singleton	14	14	28	O(4), C(8), N(2), double(3), simple(11)	5 (click to show)
P8	7 (9.308)	155	Not singleton	13	13	26	O(3), C(8), N(2), double(3), simple(10)	3 (click to show)
P10	17 (8.028)	119	Not singleton	11	11	22	O(3), C(7), N(1), double(1), simple(10)	5 (click to show)
P11	33 (7.071)	70	Not singleton	11	11	22	O(1), C(10), simple(7), double(4)	8 (click to show)
P14	48 (6.530)	466	Not singleton	10	10	20	S(1), O(3), C(6), simple(5), double(5)	5 (click to show)
P15	5 (9.793)	248	Not singleton	10	10	20	O(2), C(7), N(1), simple(10)	5 (click to show)

Figure 4.10 – The "Pattern characteristics" block of GraphMDL Visualizer. This block shows characteristics for each pattern. Table rows can be sorted and filtered, and detailed port information can be hidden.

Step 2: pattern characteristics table. While the "Main statistics" block provides information about the general distributions of pattern characteristics, the "Pattern characteristics" block shown in Fig. 4.10 provides information about the characteristics of the individual patterns. It contains a table, each row of the table corresponding to one row of the GraphMDL code table. For each row (i.e. each pattern) it reports: the number of occurrences the pattern has in the rewritten graph and the resulting pattern's code length; the total possible occurrences of the pattern in the data (not limited to the ones in the rewritten graph); whether the pattern is a singleton pattern or not; the size of the pattern (number of vertices, edges, labels); a summary of the labels included in the pattern; and the pattern's ports.

The table is interactive. It is possible to filter the rows and to sort the table by any of its columns, thus offering different points of view. For example in Fig. 4.10, the user chose to only show non-singleton patterns containing at least an atom of Oxygen and an atom of Carbon. Then, the user sorted the table by number of vertices, so that the largest patterns would be on top. Thanks to this table, the user identified a pattern that interest them: pattern P_1. This pattern has 51 occurrences in the rewritten graph, has 19 vertices,

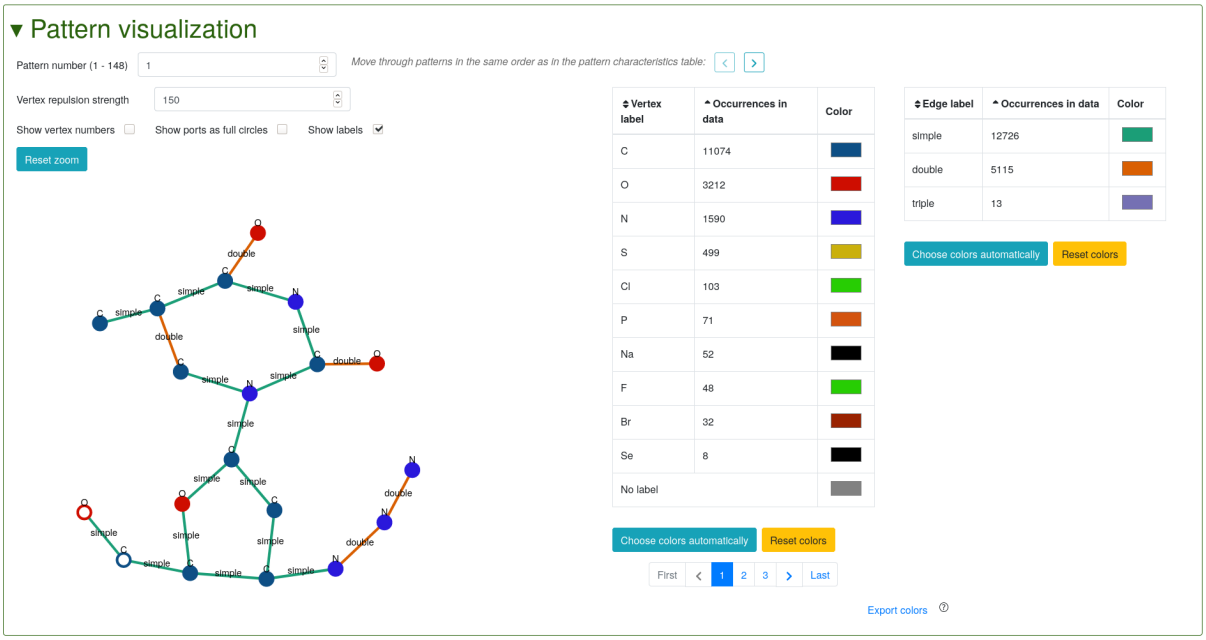


Figure 4.11 – The "Pattern visualization" block of GraphMDL Visualizer. The user can select a pattern, and the structure of the pattern will be shown. Position and colors of vertices and edges can be customised. Vertices represented as hollow circles are the pattern's ports.

20 edges and a total of 39 labels. It has two ports: its vertices 11 and 12, which are used 1 and 49 times respectively. A double-click on the pattern's row moves the visualization to the next block, which shows the pattern's structure.

Step 3: pattern visualization. Graph are complex data structures and —especially for large graphs— visualizing them allows the user to better understand them. The "Pattern visualization" block, shown in Fig. 4.11, allows the user to display the structure of the patterns selected by GraphMDL as interactive plots. The user can zoom on parts of the pattern, drag around vertices to rearrange them, and customize vertex and edge colors. There are many ways to plot a given graph, and it is not always easy to choose the layout that is the best for the user. Thus, by dragging vertices around the user can arrange the visualization to a layout that suits them. This interactivity greatly eases interpretation of the more complex patterns.

In Fig. 4.11 we can observe the structure of the pattern P_1 that the user selected when analyzing the pattern characteristics table during the previous step. It is a fairly complex

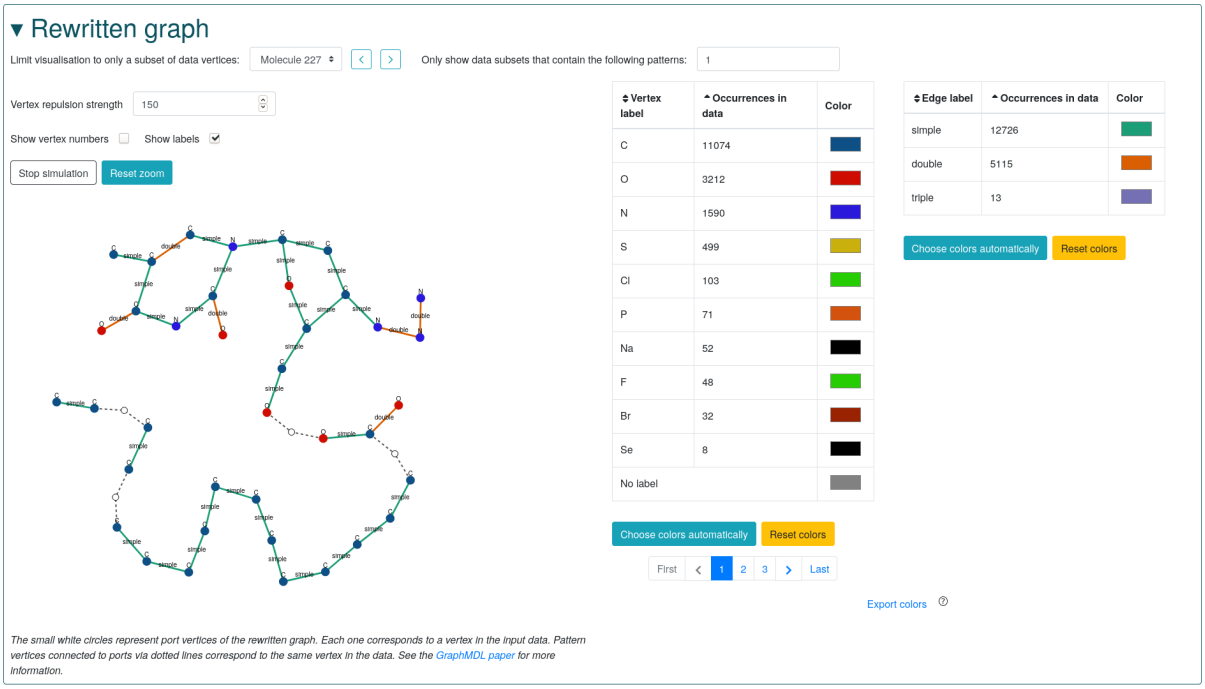


Figure 4.12 – The "Rewritten Graph Visualization" block of GraphMDL Visualizer. Here the user limited the visualization to only the data vertices corresponding to one of the molecules in the AIDS-CA dataset.

molecule with 19 vertices, which would be difficult to interpret without a plot. Thanks to the visualization we can also easily identify that the pattern can be connected to other patterns only via the two vertices at the bottom left, which are marked as being ports (ports are shown as hollow circles).

Step 4: rewritten graph visualization The "Rewritten graph" block, shown in Fig. 4.12, visualizes GraphMDL's rewritten graph. Since the data may be large, it would be very difficult to show the rewritten graph for the whole data at once: it would be difficult for the user to analyze and it would probably require too much computation for the web browser. Therefore, we implemented the possibility of limiting the visualization to only some subsets of the data. In order to do so, the user can upload an additional file describing some subsets of the data vertices to which the visualization can be limited. For example, since the AIDS-CA dataset is a collection of molecules, we created such a file allowing the rewritten graph of each molecule to be shown separately. We found that the visualization ran smoothly on graphs up to some hundreds of vertices and edges.

Fig. 4.12 shows the visualization of one of the molecules in the dataset, which contains pattern P_1. This visualization is similar to the pattern visualization block. In particular, the color customization of the two blocks are synced, so that the choices made when visualizing the patterns are maintained when visualizing the rewritten graph, and conversely. Thanks to this visualization, we can observe that the molecule shown in Fig. 4.12 is described by GraphMDL using 5 patterns, with pattern P_1 (at the top) describing the main bulk of the molecule.

By observing this visualization, the user will probably become interested in a new pattern (for example one of the other patterns that appear alongside P_1). A double-click on a pattern takes the user back to the pattern visualization block showing the structure of that new pattern. The user can also just go back to the pattern table or the main statistics block, and change some options in order to continue exploring GraphMDL results.

Take-Home Message

In this chapter we presented GraphMDL, a MDL-based graph pattern mining approach that selects a *small* and *descriptive* subset of graph patterns from a set of candidate graph patterns extracted from labeled graph data. The approach also outputs a structure called the *rewritten graph*, which is a description of the data encoded as a composition of pattern occurrences. The originality of the approach lies in the notion of *ports*, which guarantees that the data can be perfectly reconstructed from the extracted patterns and the rewritten graph. We proposed MDL description length definitions allowing to evaluate sets of patterns using the minimum description length principle. We proposed a greedy heuristic algorithm that uses our MDL definitions to select a subset of patterns from a set of patterns given as input. We evaluated GraphMDL experimentally, showing that it significantly reduces the amount of patterns w.r.t. classic graph mining approaches. Also, we show that the selected patterns can have complex shapes and are descriptive of the data.

In this chapter we also presented GraphMDL Visualizer, a tool that allows an *interactive visualization* of the results of GraphMDL, helping the user in their interpretation. The tool is available online and can be used to browse the experimental results presented in this thesis.

Chapter 5

Interleaving Pattern Generation and Selection: GraphMDL+

In Chapter 4 we presented the GraphMDL approach, which tackles the pattern explosion problem, i.e. the fact that —even on small datasets— classic graph mining approaches generate too many patterns for human analysis. GraphMDL tackles this problem by using an MDL criterion to select a small set of patterns out of a larger set of candidate patterns. In this way, our approach can be applied to the large set of patterns that is generated by a classic pattern mining approach (e.g. gSpan [YH02]), reducing it to a smaller set that can be processed by a human analyst. GraphMDL is a generate-then-select approach, meaning that first a large collection of candidate patterns is generated by an external graph mining approach, then GraphMDL iterates through all the candidates, evaluating each of them and deciding whether to include it in its output set. This two-phase method has two main drawbacks. First, the large number of candidate patterns that are generated from the data entails a high computational cost, especially for the MDL selection phase, which evaluates all candidates one by one. Thus, in order to make the data treatable in reasonable time, the user has no choice but to limit the number of candidate patterns that are generated by the external graph mining approach, prior to executing GraphMDL. As a consequence, there may exist some useful patterns that could be extracted from the data that are not even considered by our approach, since they are not in the limited set of candidates that is passed to it. In particular, since the external pattern mining approach has no knowledge of the MDL criterion used by GraphMDL, it is possible that it would discard patterns that are well-suited for the MDL criterion, all the while generating many patterns that are not much useful for it, and whose processing requires computation time without providing a significant gain. The second drawback is that GraphMDL only does a *selection* of the candidate patterns that it receives. That means that any output that it generates contains —by definition— only patterns that are in the candidate set. As a consequence, any bias or limitation of the

external pattern mining algorithm that is used to generate the candidate patterns will be transmitted through to GraphMDL. For instance, imagine that the external pattern mining algorithm only generates patterns that have a certain shape. In that case, even if a pattern with a different shape would be interesting from an MDL point of view, it would never be considered by GraphMDL.

In this chapter we present GraphMDL+, an approach that tackles the limitations of GraphMDL presented above. GraphMDL+ uses the same description length definitions as GraphMDL, but tightly interleaves pattern generation and pattern selection (instead of generating all candidate patterns beforehand). We show that thanks to this coupling between the generation step and the selection step, GraphMDL+ can generate better pattern sets than GraphMDL (according to the MDL criterion) and in significantly less time. In addition, GraphMDL+ is completely detached from external graph mining approaches, thus it does not suffer from their limitations about the nature of graphs (e.g. directed vs undirected) or patterns (e.g. shapes of patterns). As a consequence, GraphMDL+ is a parameter-less and anytime approach, it can output a descriptive set of patterns whenever the user decides so. In MDL-based pattern mining, GraphMDL+ is to GraphMDL on graph mining as Slim [SV12] is to Krimp [VLS11] on itemset mining.

This chapter is organised as follows. Section 5.1 presents the intuition behind GraphMDL+ and its way of generating candidate patterns. Section 5.2 formally presents the data structure used to describe candidate patterns. Section 5.3 presents a heuristic used to rank all the candidate patterns generated at a given step, in order to choose which one to consider first for inclusion in a code table. In Section 5.4 we put all the previous notions together and present the algorithm used to generate and select candidate patterns. Section 5.5 makes a focus on the notion of graph isomorphism and automorphism and their impact in the context of GraphMDL+. Finally, Section 5.6 presents the experiments performed to evaluate GraphMDL+ and to compare it to GraphMDL.

This chapter is based on work that has been published at the *ACM Symposium on Applied Computing (SAC)* [BCF21].

5.1 The Intuition Behind GraphMDL+

The most substantial difference between GraphMDL and GraphMDL+ is the ability for the latter to generate candidate patterns by itself. This generation is constructed

around GraphMDL's notion of rewritten graph. Given a code table, the rewritten graph represents how the approach uses the patterns of the code table to encode the data. Our intuition is to inspect the rewritten graph and observe which patterns are frequently neighbours of each other. Two patterns are neighbours if some of their embeddings share at least one port vertex in the rewritten graph. If two patterns are frequently neighbours, we merge them to create a larger pattern that will be able to cover their common occurrences. The shared port(s) could then become internal vertices of the larger pattern, and hence not ports anymore. The rationale behind this intuition is that if pattern A and pattern B frequently appear sharing a given port, maybe their occurrences indicate the presence of a larger pattern $A \cup B$, where the shared port is an internal vertex of that pattern.

When two patterns are merged, the resulting pattern is added to the code table and this new code table is again used to encode the data. If the MDL description length obtained with the new code table is better (i.e. lower), the new pattern is kept. Thanks to this approach, we can start with small (i.e. very general) patterns in the code table, and make our way up to generate larger (i.e. more specific) patterns. In particular, we can start with singleton patterns —the smallest patterns possible— so as to ensure that any pattern can potentially be generated, since any pattern graph can be constructed from connected singleton components. Larger patterns are interesting because they allow to describe more of the data in a single occurrence, and can highlight complex components. At the same time, they have less occurrences and they increase the description length of the code table. The MDL principle acts as a gatekeeper, guaranteeing a balance between the complexity of the patterns and of the encoded data. In conclusion, the GraphMDL+ algorithm can be summarised as follows:

1. Compute the rewritten graph for the current code table (use a singleton-only code table the first time).

2. Find neighbouring patterns (i.e. that share ports) and merge them two by two to create new patterns (see Section 5.2).

3. Try to add one candidate to the code table. If the description length improves, start again with this new CT, otherwise try the next candidate (see Section 5.3 for explanations about candidates order).

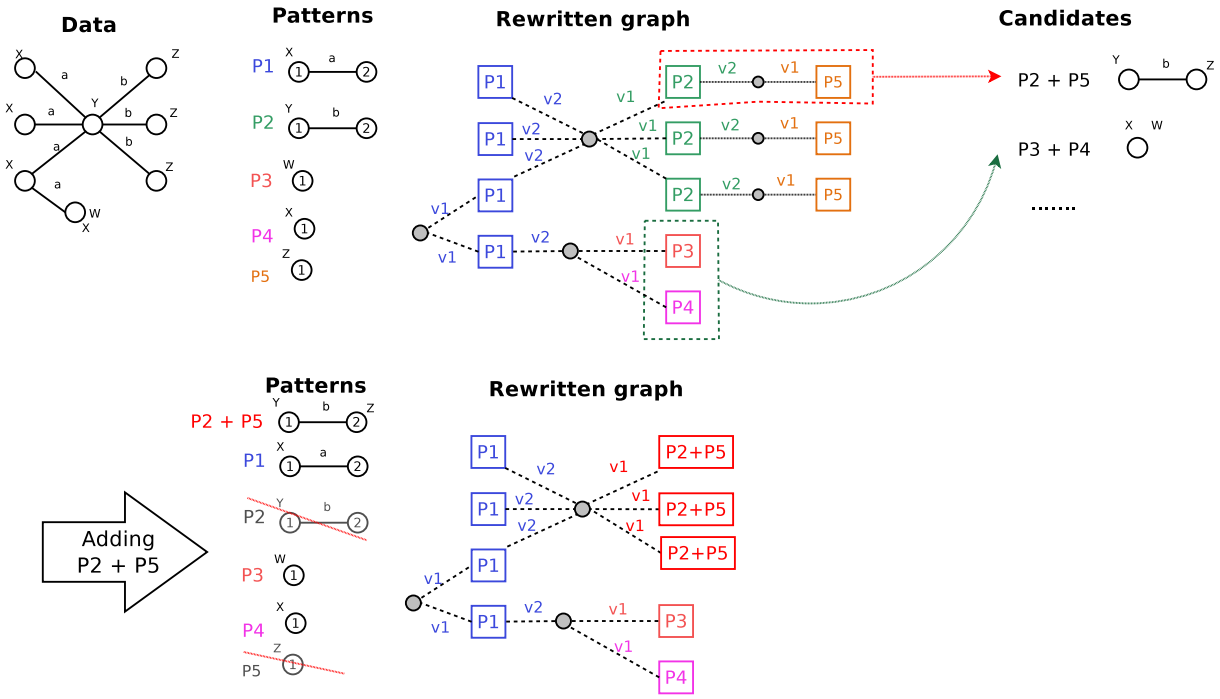


Figure 5.1 – How neighbouring pattern occurrences in the rewritten graph are used in GraphMDL+ to generate new patterns.

5.2 Merge Candidates

GraphMDL+ generates candidate patterns by merging patterns that have ports in common. Let P_A be a pattern which has an occurrence in the rewritten graph that maps one of the pattern vertices, v_A, to data vertex v_D (i.e. there is an edge marked v_A in the rewritten graph between one embedding vertex of P_A and the port vertex corresponding to v_D). Let P_B be a pattern which has an occurrence in the rewritten graph that maps one of the pattern vertices, v_B, to the same data vertex v_D. The intuition behind GraphMDL+ (see previous section) says that it may be interesting to merge P_A and P_B, merging their vertices v_A and v_B. The pattern resulting from this merge would allow to describe *with a single pattern* all those parts of the data where there is an occurrence of P_A and an occurrence of P_B mapping their respective vertices v_A and v_B on the same data vertex. For example, highlighted in red in Fig. 5.1 we can observe such a situation where $P_A = P_2$ and $P_B = P_5$, and where vertex v_2 of P_2 is connected to the same port vertex than vertex v_1 of P_5.

We call the structure describing the possible merge of two patterns and the resulting

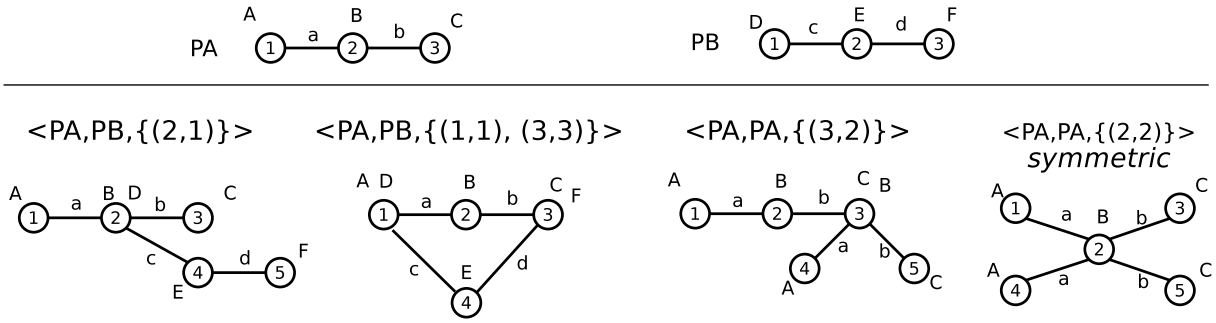


Figure 5.2 – Two patterns, P_A and P_B, and patterns obtained when they are merged following four different merge candidates.

new pattern a *merge candidate*, or *candidate* for short.

Definition 13 (Merge candidate) *A* merge candidate $C = < P_A, P_B, \Pi >$ *is composed of two patterns,* P_A *and* P_B*, and a non-empty set of* port tuples $\Pi = \{(\pi_{A,1}, \pi_{B,1}), (\pi_{A,2}, \pi_{B,2}),\ \ldots\}$ *that represents the vertices of each pattern that are to be merged together to form the new pattern.*

Note that P_A and P_B can be the same pattern. Indeed, there can be situations where two occurrences of the same pattern are neighbours and share a port, and merge candidates can describe those situations. There is a special case in which not only the two patterns of the candidate are the same, but the first occurrence can act as the second and conversely. In that case we say that the candidate is *symmetric*.

Definition 14 (Symmetric merge candidate) *A candidate* $< P_A, P_B, \{(\pi_{A,1}, \pi_{B,1}),\ \ldots\} >$ *is* symmetric *if and only if it is equal to* $< P_B, P_A, \{(\pi_{B,1}, \pi_{A,1}),\ \ldots\} >$.

Fig. 5.2 shows two patterns and the different patterns that are obtained when merging them following different merge candidates. The last two candidates in the image represent a merge between pattern P_A and itself: the first of the two is not symmetric since different ports are used in the first and second occurrence, but the last candidate is symmetric.

5.3 Candidate Ranking

Many different candidates can be generated from a rewritten graph: one for each different pair of neighbouring pattern occurrences. Thus, a choice has to be made about the

order in which those candidates are considered for inclusion in the code table. A naive approach would be to try them all separately and rank them based on how much the MDL description length improves (i.e. lowers) for the CT in which they are included versus the current CT, in which they are not. However such an approach would be computationally expensive because it implies recomputing the data encoded with the CT for each candidate, which is an expensive operation. In this section we present the heuristic that GraphMDL+ uses to rank candidates without needing to compute the exact description length gain that they give. This heuristic is evaluated experimentally in Section 5.6.2.

Our intuition is to select the candidate that is expected to be used the most when encoding the data. Remember that the more frequent a pattern is in the encoded data, the smaller its code is, and the more interesting it is from a MDL point of view. On top of that, the more a pattern is used, the higher the impact it has on the description length of the data encoded with the model (since many parts of the data are replaced by occurrences of the candidate). Therefore, by considering first the candidate with the most expected usage, we focus first on the candidate expected to give the highest swing in description length when added to the CT, since the cost of adding it to the model can be balanced by the increased gain in the description length of the encoded data. We call this measure the *usage estimate* of the candidate. We compute the usage estimate of a candidate $C =< P_A, P_B, \Pi >$ by counting the number of embeddings of P_A and P_B that appear next to each other in the rewritten graph as described by the candidate. We take as estimate the smaller of the two counts. For example, if there are 3 embeddings of each pattern, we can expect the pattern described by the candidate to be used at least 3 times as it covers all of those occurrences [1]. However, if there are only 2 embeddings of P_A versus 3 of P_B, it means that two embeddings of P_B share the same embedding of P_A, therefore we can expect the pattern described by the candidate to be used only two times.

There are some special cases in which the general formula for the usage estimate needs to be adapted. If one of the two patterns does not have edges (i.e. it only has one vertex), its number of embeddings is not a limiting factor to the number of embeddings of the candidate because GraphMDL+ (like GraphMDL) only forbids edge overlaps. In that case, the usage estimate of the candidate is the number of embeddings of the other

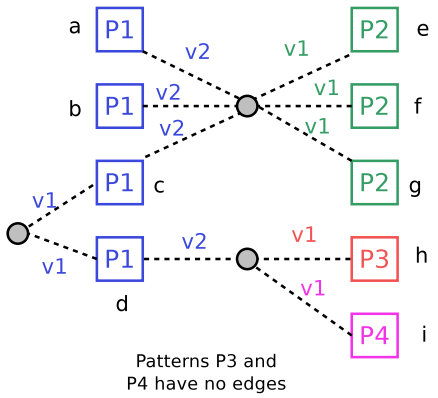
1. However the exact number of occurrences could be higher. The higher a pattern is in the code table, the more its embeddings have a chance of being used in the encoded data (See Section 4.3). Therefore, depending on where the new pattern is in the CT, it could potentially cover some occurrences of P_A and P_B that were not in the rewritten graph as they were already covered by a different CT pattern. In order to know the *exact* number of embeddings, the full rewritten graph would have to be recomputed.

pattern. If both patterns have no edges, the usage estimate of the candidate is the number of embeddings —that participate to the candidate— of any of the two (the number will be the same for both patterns). If a candidate is symmetric, it means that embeddings of P_A can count as embeddings of P_B and conversely. In that case, we aggregate both sets of embeddings and take as usage estimate half their number (rounded down). We take half the number of embeddings because candidates merge patterns two by two.
We summarise the computation of the usage estimate as follows:

1. If the candidate is symmetric, combine embeddings of the two patterns. Usage estimate is half the size of the resulting set.
2. Otherwise, if one of the patterns does not have edges, usage estimate is the number of embeddings of the other pattern.
3. Otherwise, usage estimate is the number of embeddings of the pattern that has less embeddings.

We break equalities between candidates having the same usage estimate by ordering them by descending number of *exclusive ports* and then description length of the structure of the new pattern. The number of *exclusive ports* for a candidate is the number of port vertices around which only the two patterns of the candidate are present. If the candidate is retained, those ports will disappear since they will be completely covered by the candidate. Since ports increase the description length of the rewritten graph we favor candidates that eliminate more of them. To further break equalities, we compute the description length of the structure of the pattern ($L(G^P)$) described by the candidate. We consider first the candidate whose pattern has the structure giving the smallest description length, because from an MDL point of view it is the "simplest".

Fig. 5.3 shows a rewritten graph and the candidates that can be generated from it, along with their usage estimate and number of exclusive ports. For example, the first line of the table shows the general case, in which pattern P_1 and pattern P_2 are connected through the vertex 2 of P_1 and the vertex 1 of P_2, yielding the merge candidate $< P_1, P_2, \{(v_2, v_1)\} >$. There are three embeddings of each pattern that can take part in this candidate (around the topmost port vertex). Therefore, the usage estimate of the candidate is 3. The candidate on the second line ($< P_1, P_1, \{(v_2, v_2)\} >$) is symmetric: it describes an embedding of P_1 connected to another embedding of P_1, through the vertex 2 of both. There are three embeddings of P_1 in the situation described by this candidate (left of topmost port vertex). Since the candidate is symmetric, each of them could be both an embedding of the first and the second pattern of the candidate. Therefore we



Candidate	Embeddings of P_1	Embeddings of P_2	Usage estimate	#excl. ports
$< P_1, P_2, \{(v_2, v_1)\} >$	$\{a, b, c\}$	$\{e, f, g\}$	3	0
$< P_1, P_1, \{(v_2, v_2)\} >$	$\{a, b, c\}$	$\{a, b, c\}$	$\lfloor 3/2 \rfloor = 1$	0
$< P_1, P_1, \{(v_1, v_1)\} >$	$\{c, d\}$	$\{c, d\}$	$\lfloor 2/2 \rfloor = 1$	1
$< P_1, P_3, \{(v_2, v_1)\} >$	$\{d\}$	$\{h\}$	1	0
$< P_1, P_4, \{(v_2, v_1)\} >$	$\{d\}$	$\{i\}$	1	0
$< P_2, P_2, \{(v_1, v_1)\} >$	$\{e, f, g\}$	$\{e, f, g\}$	$\lfloor 3/2 \rfloor = 1$	0
$< P_3, P_4, \{(v_1, v_1)\} >$	$\{h\}$	$\{i\}$	1	0

Figure 5.3 – A rewritten graph and the merge candidates that can be generated from it. The rewritten graph's embedding vertices have been numbered a to i to identify them in the table. The structures of patterns P_3 and P_4 contain no edges (thus they are one of the special cases of the usage estimate computation).

aggregate the embeddings and take half their number (rounded down) as usage estimate. The third line of the table shows candidate $< P_1, P_1, \{(v_1, v_1)\} >$, where two embeddings of P_1 are connected through their vertex 1: this situation can be seen around the leftmost port vertex. Around this port vertex there are only those two embeddings, therefore this port is an exclusive port, and we note that the candidate has one exclusive port. Finally, the fourth line ($< P_1, P_3, \{(v_2, v_1)\} >$) describes the last case of usage estimate: an embedding of P_1 connected to an embedding of P_3, which is a pattern that has no edges. In this case, only the number of embeddings of P_1 that participate to the candidate (i.e. embedding d) is considered for the usage estimate.

5.4 The GraphMDL+ Algorithm

Having defined how to generate candidate patterns and how to rank them, we now present the GraphMDL+ algorithm, shown in Alg. 2. First, the code table is initialised as a code table containing all the singleton patterns for the labels in the data (lines 1-2). Then, the algorithm starts iterating: during each iteration (lines 4-18), the current code table is improved by adding a single pattern generated from the previous CT. First, the data is encoded using the current CT, generating a rewritten graph (line 5). Then, merge candidates are generated from the rewritten graph (line 6) and are sorted (line 7) according to the ranking presented in Sec. 5.3 (usage estimate, exclusive ports, pattern structure description length). Then, the highest-ranking candidate is added to the code

Algorithm 2 The GraphMDL+ algorithm

Require: A data graph D

1: Compute initial singleton code table CT_0 from labels in D
2: $CT \leftarrow CT_0$
3: **return** GraphMDL+Iteration(D, CT)

4: **function** GraphMDL+Iteration(D, CT)
5: $G^R \leftarrow$ rewritten graph $D|CT$
6: $\mathcal{C} \leftarrow$ candidates generated from G^R ▷ See Sec. 5.2
7: $\mathcal{C}_{ranked} \leftarrow \mathcal{C}$ sorted by usage estimate ▷ See Sec. 5.3
8: **for all** $C \in \mathcal{C}_{ranked}$ **do**
9: $CT' \leftarrow CT \cup C$
10: **if** $L(CT', D) < L(CT, D)$ **then**
11: **if** pruning enabled **then**
12: $CT' \leftarrow$ Pruning(CT, CT')
13: **end if**
14: **return** GraphMDL+Iteration(D, CT')
15: **end if**
16: **end for**
17: **return** CT
18: **end function**

table (line 9). If this new CT improves the description length w.r.t. the previous one (line 10), the new code table becomes the CT for a new iteration (line 14). Otherwise, another candidate is tested. When all candidates from the current rewritten graph have been tested without any of them yielding a better description length, the program returns the current code table as the best CT found (line 17). Note that GraphMDL+ also implements GraphMDL's optional post-acceptance pruning, where after a new code table is found that improves description length, patterns whose usage has lowered can be removed if their removal helps the description length.

An interesting aspect of the GraphMDL+ algorithm is that it can be interrupted whenever the user desires so, since it can just return the code table found at the previous iteration. This makes GraphMDL+ an *anytime* algorithm, and effectively a *parameter-less* one as well, since the only input it needs is the data graph.

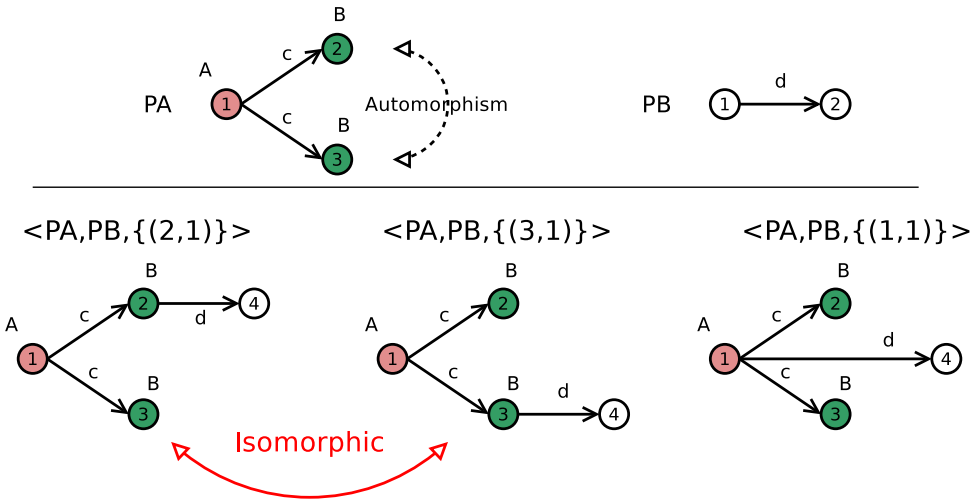


Figure 5.4 – Three merge candidates generated from two patterns. The first two candidates describe isomorphic patterns because of the automorphism of pattern P_A.

5.5 Isomorphisms in the Context of GraphMDL+

We mentioned in Section 2.1.2 that isomorphisms and automorphisms are important concepts when dealing with graphs. The GraphMDL+ merge candidate generation is a situation where those concepts play a part. Fig. 5.4 shows two patterns, and three different merge candidates concerning those patterns. The three merge candidates are different, in that their patterns and port tuples differ. However, when looking at the patterns that are generated by each candidate, we observe that the first two candidates generate isomorphic patterns, i.e. they generate the same graph. This is because pattern P_A has an automorphism that swaps its vertices 2 and 3, so that under that automorphism the port tuple of the first candidate becomes the one of the second and conversely. If GraphMDL+ did not understand this concept, it would see the two candidates as being different and treat them separately. This would have several drawbacks. First, the usage estimate of the candidate (as presented in Section 5.3) would be incorrect, since the occurrences of the first candidate would be considered disjoint from the occurrences of the second, whereas they actually represent occurrences of the same larger pattern. This would have the consequence of devalorising the candidate, as the computed usage estimate would be lower. Second, this multiplies the number of candidates. Imagine that the first candidate did not improve the code table: if GraphMDL+ treated them as different, it would then test the second. However, if the pattern generated from the first candidate did not improve the CT, the pattern generated by the second would naturally not improve it either. By

recognizing that they are the same candidates, they are grouped together and tested only once.

The way that GraphMDL+ deals with isomorphisms in merge candidates is the following. Each time a pattern is generated, its automorphisms are computed (we use the bliss algorithm [JK07] in our implementation). Then, when a merge candidate is generated, we loop through the automorphisms of the two component patterns, applying them to the port tuples of the candidate, checking if that transforms the candidate into a candidate that we have already seen. In that case, we correctly consider the new candidate as an additional occurrence of the one that we have already seen, instead of a separate candidate. In our experiments, we found that this additional isomorphism check between candidates had a negligible impact on the overall runtime of the approach.

5.6 Experiments

In this section we first present a comparison between GraphMDL+ and GraphMDL. Then, we evaluate the candidate ranking presented in Section 5.3. We then observe how the description length of GraphMDL+'s code tables evolves during the search, and how the user could take advantage of that. Then, we discuss the advantages given by having control over the pattern generation and its integration in the MDL selection process. Finally, we evaluate the impact that handling automorphisms has on the results of GraphMDL+.

We conducted the experiments on the same datasets used in Sec. 4.4 for the GraphMDL evaluation. We generated candidates for GraphMDL using the same gSpan implementation as in Sec. 4.4. We implemented a prototype of GraphMDL+ in Java 1.8 based on GraphMDL's code. It is available as a git repository [2].

5.6.1 Comparison with GraphMDL.

In order to compare the performances of GraphMDL and GraphMDL+ we ran experiments, allowing each experiment a 4 hours maximum runtime. For GraphMDL, we chose a gSpan support as low as possible that would yield a set of candidate patterns which could be treated withing the given time limit (not including gSpan runtime).

Controlling GraphMDL runtime. The first observation that we made while running

2. `https://gitlab.inria.fr/fbariatt/graphmdl/`

Table 5.1 – Comparison of GraphMDL and GraphMDL+. The time required for generating candidates with gSpan is not included in the runtime of GraphMDL, since it is negligible.

	GraphMDL best $L\%$			GraphMDL+ for $L\% \leq L_1$		GraphMDL+ best $L\%$		
Dataset	$L_1 = best\ L\%$	time	$\|CT\|$	time	$\|CT\|$	best $L\%$	time	$\|CT\|$
AIDS-CA	19.2%	1h52m	148	36s	110	12.0%	1h17m	305
AIDS-CM	20.7%	2h50m	212	1m22s	170	14.7%	4h00m	644
Mutag	15.4%	2h38m	29	3s	33	10.7%	1m28s	48
PTC-FM	22.9%	1h42m	77	12s	77	22.0%	1m08s	87
PTC-FR	23.4%	28m	70	8s	77	22.6%	1m23s	85
PTC-MM	23.7%	2h09m	75	4s	76	22.1%	1m16s	84
PTC-MR	23.1%	19m	78	7s	84	21.4%	1m01s	87
UD-PUD-En	26.8%	1h42m	647	2m59s	459	25.3%	2h32m	801

these experiments is that it is difficult to control the runtime of GraphMDL: a small change in gSpan support can drastically change the number of candidates generated and therefore the GraphMDL runtime. It was difficult to tune the support parameter so as to generate the best set of candidates which could fit in the experiment's time limit. On the opposite, since GraphMDL+ is designed to be an anytime algorithm, it can be easily stopped at a chosen time limit.

GraphMDL+ versus GraphMDL. We present the results of the experiments in Table 5.1. For each dataset there are three sections in the table. The first section presents the results obtained by GraphMDL on the dataset: the compression ratio $L\% = \frac{L(CT,D)}{L(CT_0,D)}$ given by the best CT it finds ; the runtime of the approach ; and the number of patterns in that CT. The second section of the table presents the time needed by GraphMDL+ to find a code table yielding a $L\%$ at least as good as the one found by GraphMDL, and the number of patterns in that code table. The third and last section presents the best results obtained by GraphMDL+ within the time limit of 4 hours: the compression ratio given by the best CT it finds ; the runtime of the experiment ; and the number of patterns in the CT. Note that for most experiments GraphMDL+ finished before the 4 hours time limit was attained. This is because it reached a rewritten graph where no candidate improved the description length (see Section 5.4).

We observe that GraphMDL+ manages on all datasets to find a code table with a compression ratio at least as good as GraphMDL and a similar amount of patterns. It does so in significantly less time. For instance, on the UD-PUD-En dataset, GraphMDL+ only takes 3 minutes to reach the same description length than GraphMDL, which takes 1h41m. We attribute this to the fact that the latter processes all candidate patterns in

Table 5.2 – Classification accuracies for GraphMDL and GraphMDL+

	AIDS-CA/CI	Mutag	PTC-FM	PTC-FR	PTC-MM	PTC-MR	UD-PUD-En/De/Fr/It
GraphMDL	71.61 ± 0.96	80.79 ± 1.51	59.02 ± 1.96	62.70 ± 1.86	62.08 ± 1.98	57.38 ± 1.68	69.45 ± 0.27
GraphMDL+	79.70 ± 1.16	80.69 ± 2.14	57.18 ± 2.42	60.43 ± 1.90	58.67 ± 2.13	55.61 ± 2.29	77.83 ± 0.39

its input set of patterns, but most of those patterns will not be useful for decreasing the description length, since they are generated by an external approach with no knowledge of the MDL principle. On the other hand, GraphMDL+ generates less candidates of a higher quality for its MDL approach, hence its increased speed for attaining an equivalent description length. A consequence of this speed increase is that GraphMDL+ can find code tables that yield a description length even lower than the best ones found by GraphMDL. For instance, for the Mutag dataset, the best code table computed by GraphMDL yields a compression ratio of 15.42% whereas the best one computed by GraphMDL+ gives 10.73%. We think that given enough time and candidate patterns, GraphMDL would eventually find those code tables, but in such a long time that it would impair the practicality of the approach. We observe that the number of patterns in the best GraphMDL+ code table is higher than in the best GraphMDL one. It remains however significantly smaller than the number of patterns generated by gSpan on the same datasets.

In short GraphMDL+ is faster than GraphMDL, which allows it to find better code tables in less time than GraphMDL. It is therefore a clear improvement from the previous algorithm. We attribute this improvement to the interleaving of the generation and selection of patterns, which allow the generation step to generate less patterns of higher quality for the selection step.

Assessing patterns through classification. In Sec. 4.4 we used a classification task to evaluate GraphMDL. We performed a similar experiment with GraphMDL+, in order to compare them. Table 5.2 presents the accuracy obtained over 10 runs of 10-fold classification for both approaches. The results of this experiment are not very significative, as we observe that the two approaches are on par with each-other, each one being sometimes better and sometimes worse. We think that this is because both approaches follow the same objective function (the MDL description length) when finding patterns. Even if GraphMDL+ extracts better patterns by this MDL metric, this does not necessarily correlate with a better performance in the classification task. In particular, both approaches may have the same inconvenient that they extract descriptive patterns more

than discriminative patterns, which could be better for classification.

In conclusion, GraphMDL+ is a net improvement over GraphMDL, capable of extracting better patterns (by their metrics) and in less time. However it is not such an improvement in the ability to classify graphs, where a different approach may be more suitable. Since it is an improvement from GraphMDL on all other metrics, there are no drawbacks on using it over its predecessor.

5.6.2 Candidate Ranking Evaluation

In Sec. 5.3 we presented the ranking that GraphMDL+ uses to choose in which order to test the candidates generated on a rewritten graph. In order to evaluate this choice of ranking, we ran three versions of GraphMDL+ which used different candidate ranking functions. The "test all candidates" version computes for each candidate the *exact* description length obtained when it is added to the CT. It does so by simply trying all candidates one by one and computing for each the MDL description length for a code table with that candidate added. Note that this computation is costly in terms of time. The "random ranking" version randomly sorts the candidates. The "GraphMDL+ ranking" version uses the ranking that we presented in Sec. 5.3. All versions use the same algorithm (presented in Sec. 5.4, without pruning enabled) except for the candidate ranking. Meaning that after the candidates are ranked, they proceed through the ranked list and select the first candidate that improves the description length. This experiment does not question this greedy approach, but evaluates the heuristic that guides it.

Fig. 5.5 presents the results of this experiment on the AIDS-CA dataset. Results on the other datasets follow the same behaviour. The figure shows the evolution of the description length of the CT as a function of time (left) or GraphMDL+ iterations (right). Each point corresponds to a single code table that improved the description length w.r.t. the previous one. Note that the "test all candidates" experiments was stopped after 18 hours, while the others terminated before that.

Evolution over time. We observe that testing all candidates takes a significant amount of time and that a heuristic is needed if we want the approach to run at a reasonable speed. The ranking that we propose is the fastest approach of the three, faster even than the random ranking. We attribute this to the fact that the first candidates in the random ranking may not actually reduce the description length, so that more of them have to be tested before one is found yielding a better description length. Also, the time cost

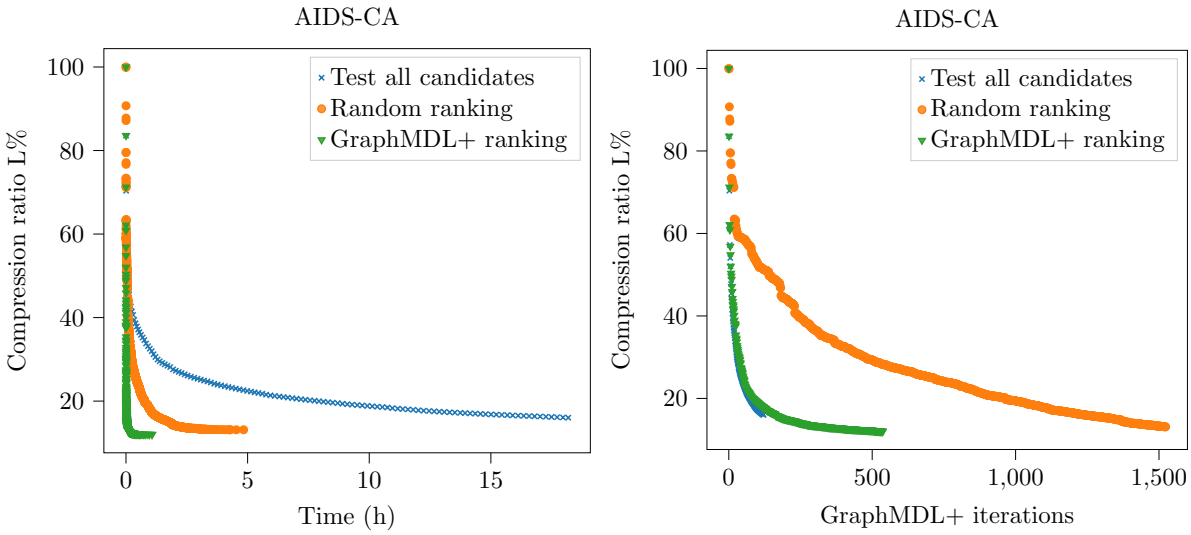
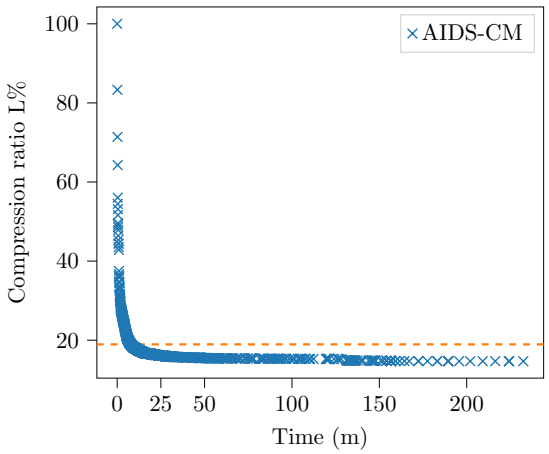


Figure 5.5 – Evolution of the description length for different candidate rankings on AIDS-CA. Left: as a function of time. Right: as a function of GraphMDL+ iterations.

of computing our candidate ranking is negligible, since it can be computed during the candidate generation phase: since all pattern embeddings in the rewritten graph have to be processed to generate the candidates, they can also be stored at the same time in order to compute the usage estimate.

Evolution over iterations. The "test all candidates" version is the one that improves the description length the most each iteration. This was expected, since it tests all candidates and selects the one that improves the description length the most, by definition. The ranking that we propose is close to it, meaning that it is a good approximation of the exact gain given by each candidate. The random ranking is the worst from this point of view as expected: while the selected candidate must improve the description length, there is no guarantee that it is the one that improves it the most. Thus, the description length decreases the least each iteration w.r.t. the other two versions. As a consequence, the random ranking needs to perform more iterations of the GraphMDL+ algorithm in order to find code tables that approach the ones found by the other two versions.

In conclusion this experiment shows that a heuristic is needed in order for GraphMDL+ to complete in a reasonable time, and that the one that we propose in Sec. 5.3 is both fast and efficient in finding good candidates.



Dataset	Time to reach 95% of L% gain
AIDS-CA	1m38s
AIDS-CM	7m54s
Mutag	0m08s
PTC-FM	0m6s
PTC-FR	0m7s
PTC-MM	0m8s
PTC-MR	0m42s
UD-PUD-En	3m47s

Figure 5.6 – Left: evolution of the description length given by the CT on the AIDS-CM dataset as a function of time. Right: time to reach a CT giving a description length that is 95% of the way w.r.t. the best CT found at the end of the 4h experiment.

5.6.3 Description Length Evolution over Time

Even though we ran our experiments with up to 4 hours of runtime, we expect that a user would probably not want to wait so long in order to get a result from their experiments. Since GraphMDL+ is an anytime algorithm, it can be stopped at any point during the search, yielding the best result found so far. It is interesting to analyze how the description length of the best CT found by GraphMDL+ evolves over time, in order to know after how much time it could be stopped while yielding a good enough result. On the left of Fig. 5.6 we plotted the evolution of the description length of the code table found by GraphMDL+ on the AIDS-CM dataset as a function of time. Each point of the plot corresponds to one iteration of GraphMDL+, where a new candidate is inserted in the code table and the MDL description length lowers. We observe that most of the gain is obtained in the first few minutes, going from the compression ratio of 100% of the starting model CT_0, to around 19% (corresponding to the orange dashed line). After that, the decrease is much slower, going from that 19% to around 15% in about 4 hours. The same behaviour is obtained on the other datasets. In the table on the right of Fig. 5.6 we report for each dataset the time needed to attain a compression ratio that is 95% of the way w.r.t. the best compression ratio found at the end of the experiment, in the same way that a compression ratio of 19% is 95% of the way between 100% and 15%. We observe that on all datasets GraphMDL+ obtains most of its results quickly, which means that the user could stop the experiment quite soon and still obtain most

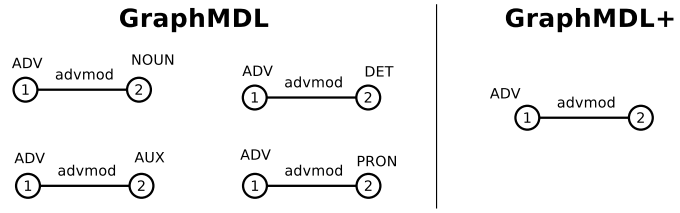


Figure 5.7 – Some patterns extracted by GraphMDL and GraphMDL+ on the UD-PUD-En dataset.

of the work that GraphMDL+ can provide. The user could also monitor the evolution of the compression ratio, drawing a plot similar to the one that we presented, and stop GraphMDL+ when the evolution flattens in the same way that we observe on our plot.

5.6.4 Advantages of Controlling the Pattern Generation

Since GraphMDL+ directly controls the generation of patterns, it has more freedom than GraphMDL over the shape of patterns, which can only use the patterns generated by the external pattern generation approach.

Vertices without labels. The graph definitions used by GraphMDL and GraphMDL+ (and in this thesis, see Sec. 2.1) allow for vertices to have any number of labels, including none. However, gSpan only treats graphs whose vertices have exactly one label each (which is a common assumption used by graph mining approaches), which means that the patterns that it generates also have exactly one label per vertex. Therefore, when GraphMDL is used in conjunction with gSpan it can only ever consider pattern that have that characteristic, since patterns are chosen exclusively from the given set of candidates. On the contrary, GraphMDL+ does not have this limitation because it generates the candidate patterns itself. Fig. 5.7 show some patterns extracted by the two approaches on the UD-PUD-En dataset. Those patterns can be interpreted as "an adverb that is the adverbial modifier of *something*". GraphMDL+ specifies this notion with a single pattern, in which the "something" does not have a label, meaning that it can be matched to any type of vertex. GraphMDL, on the other hand, can only use the patterns generated by gSpan, in which the second vertex is forced to have a label. Therefore, it generates multiple patterns: "an adverb that is the adverbial modifier of a noun", "... an auxiliary", "... a pronoun", etc. Thus, GraphMDL patterns are more specific, and lack the generalisation power of the single pattern generated by GraphMDL+. Having control over the pattern generation step gives GraphMDL+ a greater generalisation power, allowing it

Table 5.3 – Description length and runtime for the best CT found by GraphMDL+ with and without isomorphism checks on the generated candidates. Experiments limited to a runtime of 4h.

	Description length			Runtime	
Dataset	Without isomorphisms	With isomorphisms	Ratio without / standard	Without isomorphisms	With isomorphisms
AIDS-CA	111191 bits	107207 bits	1.037	3h11m	1h17m
AIDS-CM	287482 bits	287101 bits	1.001	4h00m	4h00m
Mutag	12756 bits	12388 bits	1.030	1m	1m
PTC-FM	17905 bits	17900 bits	1.000	1m	1m
PTC-FR	16988 bits	16704 bits	1.017	1m	1m
PTC-MM	16191 bits	16215 bits	0.999	1m	1m
PTC-MR	19582 bits	19509 bits	1.004	1m	1m
UD-PUD-En	298812 bits	299073 bits	0.999	4h00m	2h32m

to fully use the pattern shapes that it supports.

Directed graphs. While GraphMDL and GraphMDL+ can handle both directed and undirected simple graphs, most graph pattern mining approaches found in the literature are developed for undirected graphs only. This means that it is harder to run GraphMDL on directed data, because it depends on an external pattern generation. GraphMDL+ does not have this problem. This also means that extending this approach to other types of data will be simpler, since extending GraphMDL implies to both extend the approach and find an external graph generation approach that supports the new type of data.

5.6.5 Impact of Handling Isomorphisms

In Section 5.5 we presented how the concept of isomorphism intervenes when generating merge candidates in GraphMDL+. In order to evaluate the impact that handling isomorphisms has on the resulting code tables, we ran some experiments on a modified version of GraphMDL+ that does not handle them. Table 5.3 shows for each dataset the description length found by GraphMDL+ for this modified version and the standard version, and the ratio between the two for a maximum runtime of 4h. We observe that handling isomorphisms has a small but positive impact on the resulting description length. Indeed, the resulting description length when not handling them is increased by 4% and 3% in AIDS-CA and Mutag respectively, and not significantly changed for the other datasets. Handling isomorphisms also has a positive impact on the runtime of the approach on the AIDS-CA and UD-PUD-En datasets (the runtime on the other datasets being too small to notice an impact or reaching the 4h limit). We attribute this to the

fact that recognizing that candidates that appear different are in reality the same (via isomorphism) allows to avoid performing the same computations multiple times.

Take-Home Message

In this chapter we presented GraphMDL+, an MDL-based graph pattern mining approach that *generates and selects* a descriptive set of graph patterns from a labeled graph. GraphMDL+ tackles drawbacks of the GraphMDL approach presented in Chapter 4. Since it generates its own patterns, it does not depend on an external graph pattern mining approach to generate candidate patterns, which allows it to generate patterns that are more suited for its MDL measure. As a consequence, GraphMDL+ is an *anytime* approach which is faster than GraphMDL and can attain better results in less time. GraphMDL+ generates patterns iteratively by merging patterns of the previous iteration that appear next to each other in the data. In order to do so, we defined the *merge candidate* structure, which allows to represent the merge of two patterns. We also defined a heuristic used to rank those candidates in order to direct the search algorithm towards favorable candidates as quickly as possible. We evaluated our proposals experimentally, showing that: the resulting approach is more efficient than GraphMDL; that the proposed candidate ranking heuristic is effective at directing the search towards patterns that are interesting for the MDL criterion; that controlling the pattern generation gives more expressive power to the approach; and that due to its anytime nature, GraphMDL+ can be stopped whenever the user decides so, allowing more control about the runtime of the approach.

Chapter 6

To Knowledge Graphs and Beyond: KG-MDL

In Chapters 4 and 5, we presented two approaches, GraphMDL and GraphMDL+, that extract a small and descriptive set of graph patterns from graph data. In this chapter, we focus on performing the same task on knowledge graphs (presented in Section 2.1.3). While this may seem a trivial adaptation that only requires the aforementioned approaches to accept that multiple edges may be present between two vertices —as KGs are multi-graphs but they only treat simple graphs— we show that this is not the case. First, some of the MDL description length formulas used in GraphMDL and GraphMDL+ depend on the hypothesis that the graph is a simple graph. Thus, a new MDL description length measure needs to be defined. A second challenge comes from the graph representation of KGs. The RDF data model, which defines knowledge graphs, also defines a "graph" representation for them. However, this representation presents some peculiarities that hinder its usage in a pattern mining context. For instance, all lists are inter-connected, as they all have by definition `rdf:nil` as their last element. If this representation is treated naively, it leads to spurious patterns conveying no knowledge to the user, containing entities that are unrelated semantically, but are in the same pattern because they are adjacent in the representation of the KG. Finally, real-life knowledge graphs have vertices that often follow a power-law [GGD04], meaning that most vertices have few connections, but few "hub" vertices have a large number of connections. This proves to be a challenge in practice, as those hub vertices can yield patterns with exponential amounts of occurrences.

In this chapter we present KG-MDL, an extension of GraphMDL+ that generates and selects small and descriptive sets of graph patterns on knowledge graphs. For that, we propose a new MDL description length measure that does not rely on the data and the patterns being simple graphs. Additionally, without loss of functionality, we construct this new measure such that it can be applied to hypergraphs as well, in order to make the approach more general and applicable to even more data. In this chapter we also discuss

```
@prefix rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix xsd:    <http://www.w3.org/2001/XMLSchema#>.

<Book1>         rdf:type        <Book> ;
                <author>        <Auth1> .
<Auth1>         rdf:type        <Person> ;
                <name>          "Alice"^^xsd:string ;
                <born_in>       <City1> ;
                <died_in>       <City1> .
<City1>         rdf:type        <City> .
```

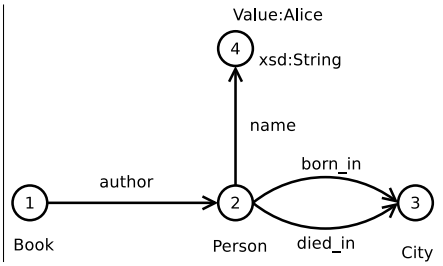


Figure 6.1 – How a KG (in Turtle) is transformed into a labeled graph.

the differences between knowledge graphs and the graphs usually used in pattern mining. We present a mapping between the RDF representation of a knowledge graph and the labeled graphs presented in Chapter 2, which is well suited for a pattern mining task. Finally, we evaluate KG-MDL on real-life medium-sized knowledge graphs, and show that the extracted patterns convey knowledge to the user about both the schema of the KG and the concrete usage of the schema in the data.

The plan of this chapter is the following. In Section 6.1 we discuss the differences between KGs and the graphs usually used in pattern mining, and describe a mapping between the usual representation of KGs and the graph definitions given in Chapter 2. Section 6.2 presents the MDL description length that KG-MDL uses to evaluate pattern sets, and its differences from the one used by GraphMDL and GraphMDL+. In Section 6.3 we present some hurdles that appeared when trying to apply KG-MDL to real-life data, and how we tackle them. Finally, Section 6.4 presents the experiments that we conducted in order to evaluate KG-MDL.

6.1 Representing KGs as Labeled Graphs

In this section we describe how knowledge graphs can be mapped from their usual RDF representation to the graph definition presented in Chapter 2. Fig. 6.1 illustrates this mapping by showing on the left an example knowledge graph in Turtle notation, and on the right its modelling as a labeled graph. The usual RDF representation of KGs already defines a "graph" representation (see Section 2.1.3), that we generally follow. However, our representation differs from it in certain cases, that we detail and justify. In the general case, each individual RDF entity is represented by an individual graph vertex, e.g. in Fig. 6.1 node `<Book1>` becomes vertex 1 of the graph. In the same vein, most RDF triples are turned into a directed edge that connects the vertex corresponding to the

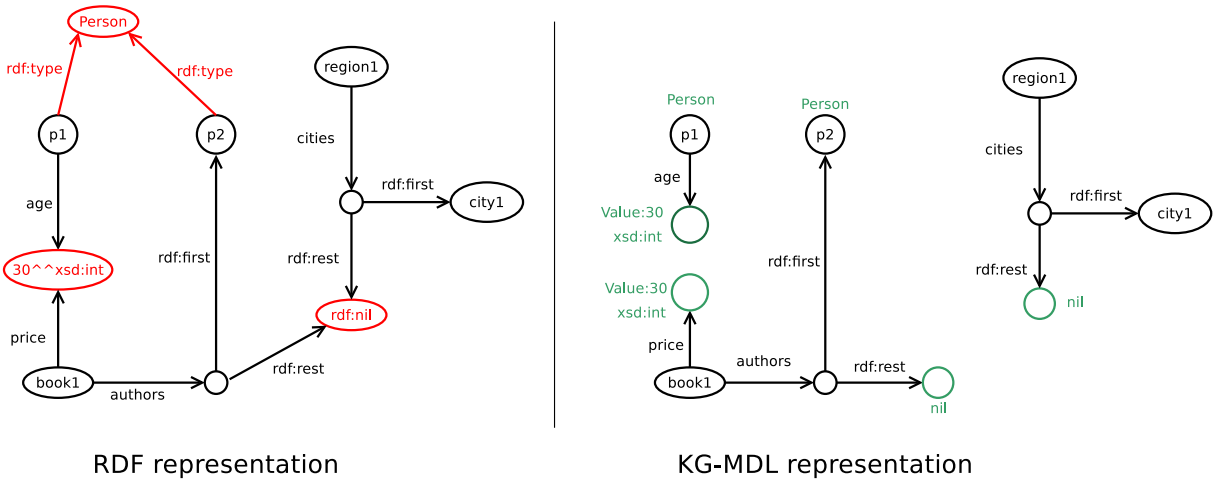


Figure 6.2 – Left: a KG represented using the usual RDF graphical representation, mapping each RDF entity to a vertex and each predicate to an edge. Right: the same KG represented following KG-MDL's mapping. Elements represented in colour differ in the two representations.

subject to the vertex corresponding to the object and that is labeled with the predicate. For instance, that is the case in Fig. 6.1 for the triple `<Book1> <author> <Auth1>`, which becomes a directed edge between vertices 1 and 2 of the graph (corresponding respectively to `<Book1>` and `<Auth1>`). However, there are three special cases in which the KG-MDL mapping differs from the RDF representation:

- Triples with predicate `rdf:type` are not transformed into edges, but instead the object (i.e. the type of the entity) is added as a vertex label to the vertex corresponding to the subject. This is the case in Fig. 6.1, for the triple `<Book1> rdf:type <Book>` that is represented by a label "Book" on vertex 1 of the graph.
- For each triple having a literal as object, a new *distinct* vertex in the graph is created for the literal, labeled with its datatype and value. For instance, in Fig. 6.1, the literal `"Alice"^^xsd:string` of triple `<Auth1> <name> "Alice"^^xsd:string` in the KG is represented by vertex 4 in the graph with two labels "Value:Alice" and "xsd:string".
- The `rdf:nil` node receives a similar treatment to literals. Each time it appears as the object of a triple, a new vertex with label "nil" is created in the graph.

The reason for these special cases is to avoid creating spurious connections between vertices that are not related semantically. Fig. 6.2 shows two representations of a KG.

In the first one the special cases that we presented are not followed, and the graph is represented as is usual in RDF: each entity is mapped to a vertex and each triple is mapped to an edge between the vertices corresponding to its subject and object. However, there are some problems with this representation. First, the vertices corresponding to `book1` and `p1` are connected to the same literal representing value 30 (by their predicates `price` and `age` respectively), even though the two measured quantities —currency and time— are not related at all. This is not a problem when the KG is used to answer queries (the context in which this representation is usually used), but it is a problem if it is used to generate patterns: we would see patterns such as "a book that costs as many dollars as a person is old in years", which relate entities just because they happen to have the same numerical values for unrelated quantities. A similar problem appears with `rdf:nil`. In RDF, all lists end with the `rdf:nil` node, which marks their end. However, when this representation is used in pattern mining, it leads to patterns where totally unrelated lists are connected just because they terminate with `rdf:nil`, i.e. in other words because they are lists. Finally, the reasoning behind the special case that we use for the `rdf:type` predicate is similar to the previous ones: we avoid connecting many vertices together just because they have the same type. But also, it stems from the fact that in graph mining vertex types are expressed as vertex labels.

In conclusion, even though knowledge graphs are "graphs", their usual representation requires some adaptations —that we presented in this section— for a pattern mining task, otherwise irrelevant patterns can be generated, which do not make sense semantically and clutter the search space.

6.2 MDL Description Lengths

KG-MDL uses the same MDL model and data encoding as GraphMDL and GraphMDL+ (see Chap. 4). The model is a *code table*, whose rows represent patterns and their associated information: their structure, a prefix code based on their usage in the encoded data, and the description of their ports (their number and identities, and a prefix code based on their usage w.r.t. other ports of the pattern). The data encoding is called the *rewritten graph*, and it shows how the data is represented as a composition of CT pattern occurrences connected by ports. Fig. 6.3, 6.4 and 6.5 represent respectively an example knowledge graphs, a KG-MDL code table, and the corresponding rewritten graph. For example we can see that the first pattern of the code table, P_1, represents the structure

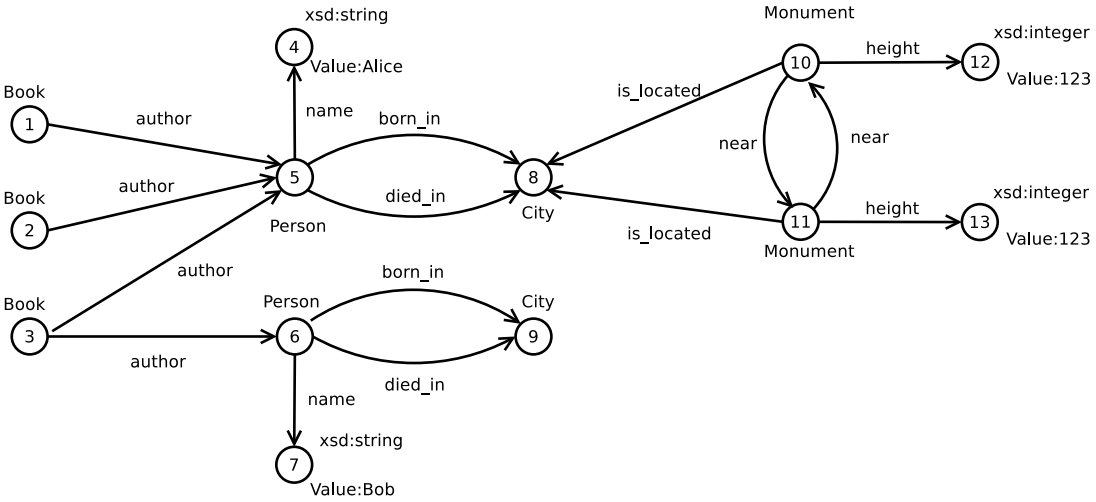


Figure 6.3 – Graph used as an example of data graph in this section.

"two monuments of height 123 that are near each other and located in the same place". This pattern is used once in the rewritten graph, in order to cover data vertices 8, 10, 11, 12 and 13. It has a single port, which is the vertex corresponding to "...the same place", which appears in the rewritten graph alongside an occurrence of pattern P_2. In total, there are 9 pattern occurrences that appear in the rewritten graph, which gives P_1 (because of its one occurrence) a prefix code of length $-\log(\frac{1}{9}) \approx 3.17$ *bits*. Since the pattern has a single port (used once), it has an associated code of length $-\log(\frac{1}{1}) = 0$ *bits* (i.e. no information is needed in order to distinguish between pattern ports as there is only one).

Code table description length. KG-MDL uses the same description length formulas for the code table as GraphMDL (see Chap. 4), excepted for the description length of the pattern's structure $L(G^P)$. This is because GraphMDL assumes that the patterns are *simple graphs*, i.e. that they have a maximum of one edge between any pair of vertices. Therefore, the number of edges around a given vertex is encoded in GraphMDL with $\log(|V^P|)$ *bits* (where V^P are the pattern's vertices), because in a simple graph there can be between 0 edges and one edge to every other vertex of the pattern (i.e. $|V^P| - 1$). This is obviously not the case in knowledge graphs, as it can be seen in Fig. 6.3, where vertex 5 has both a `born_in` and a `died_in` edge towards vertex 8.

The KG-MDL encoding of a pattern's structure is shown in Fig. 6.6 and presented hereafter. First, we encode the number of vertices $|V^P|$ of the pattern. Since a pattern

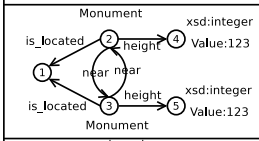
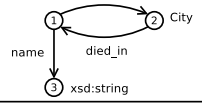
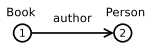
	G^P		C_P		$\lvert\pi_P\rvert$	π		C_π	
	Pattern structure	*Pattern usage*	Pattern code	*Pattern code length (bits)*	Port count	Port ID	*Port usage*	Port code	*Port code length (bits)*
P1		*1*	P1	*3.17*	1	1	*1*		*0*
P2		*2*	P2	*2.17*	3	1 2 3	*2* *1* *2*	v1 v2 v3	*1.32* *2.32* *1.32*
P3		*4*	P3	*1.17*	2	1 2	*2* *4*	v1 v2	*1.59* *0.59*
P4	Value:Alice	*1*	P4	*3.17*	1	1	*1*		*0*
P5	Value:Bob	*1*	P5	*3.17*	1	1	*1*		*0*

Figure 6.4 – The best code table found by KG-MDL on the graph of Fig. 6.3. Values in light blue italic are for illustration only and are not part of the model.

can have any number of vertices, we use a universal integer encoding $L_{\mathbb{N}}$ to encode this information. Then, we describe all of the label symbols that appear in the pattern. First we indicate how many of them appear in the pattern: since a pattern can contain between one[1] and all of the label symbols that are in the data (i.e. $\mathcal{L}$), this information can be encoded with $\log(|\mathcal{L}|)$ *bits*. Then, for each label symbol we encode: its identity, how many occurrences it has, and which ones they are between all the possible ones it can have. In order to indicate which label symbol $l \in \mathcal{L}$ we are describing, we need a code to differentiate them. We use for this a prefix code $L_{\mathcal{L}}(l)$ based on the number of occurrences that each label symbol has in the data graph. Fig. 6.7 presents the codes obtained for the graph of Fig. 6.3. Note that —differently from GraphMDL— we compute this code by mixing all label symbols, independently of their arity (vertex labels have an arity of 1, edge labels an arity of 2). This makes the encoding more general, since labels with an arity greater than 2 can be easily introduced if the approach is to be expanded to hypergraphs. In terms of description length, this is equivalent to having a first prefix code that indicates the arity of the symbol (based on the number of labels of each arity in the data) then having a second prefix code that indicates which symbol it is (based on its occurrences in the data w.r.t. other symbols of same arity)[2]. Now that we encoded which label symbol

1. A pattern with zero labels would not make sense, as it would correspond to a structure with no edges and no vertex labels.

2. This is one useful property of description lengths based on their interpretation as probability dis-

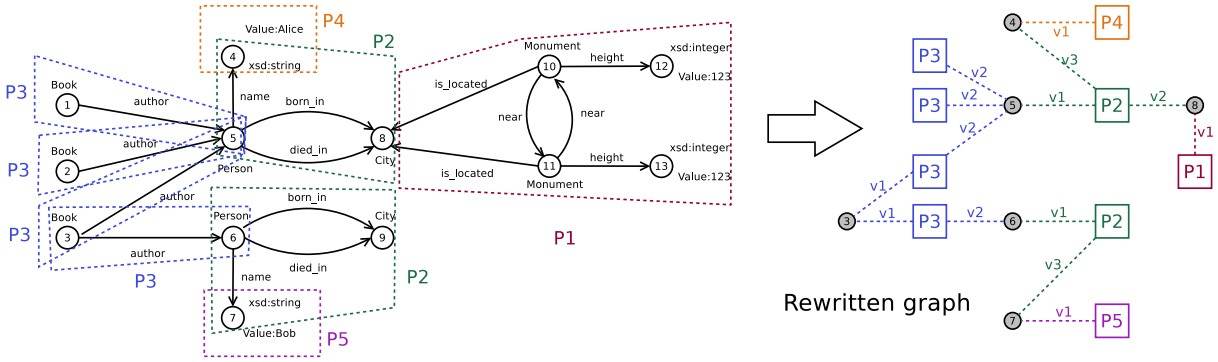
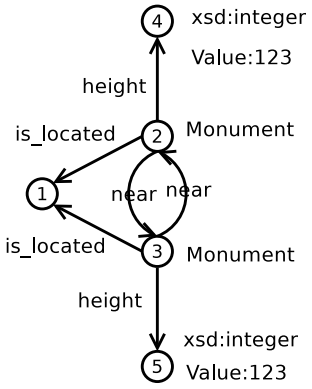


Figure 6.5 – Rewritten graph for the code table of Fig. 6.4 on the graph of Fig. 6.3.



Value to encode		Description length $L(G^P)$	
formula	concrete	formula	concrete
$\lvert V^P\rvert$	5	$L_{\mathbb{N}}(\lvert V^P\rvert)$	$L_{\mathbb{N}}(5)$
$\lvert\{l \in \mathcal{L} \mid l \text{ is in } P\}\rvert$	6	$\log(\lvert\mathcal{L}\rvert)$	$\log(16)$
l	Monument	$L_{\mathcal{L}}(l)$	4.04
#occ	2	$L_{\mathbb{N}}(\#occ)$	$L_{\mathbb{N}}(2)$
occ	$\{v_2, v_3\}$	$\log(\binom{\lvert V^P\rvert^n}{\#occ})$	$\log(\binom{5}{2})$
...+2 rows for vertex label symbols “xsd:integer” and “Value:123”			
l	near	$L_{\mathcal{L}}(l)$	4.04
#occ	2	$L_{\mathbb{N}}(\#occ)$	$L_{\mathbb{N}}(2)$
occ	$\{(v_2, v_3), (v_3, v_2)\}$	$\log(\binom{\lvert V^P\rvert^n}{\#occ})$	$\log(\binom{5^2}{2})$
...+2 rows for edge label symbols “is_located” and “height”			

Figure 6.6 – How the structure of the pattern on the left is encoded in KG-MDL. For each label symbol $l \in \mathcal{L}$, *occ* represents its occurrences (vertices or edges) in the pattern.

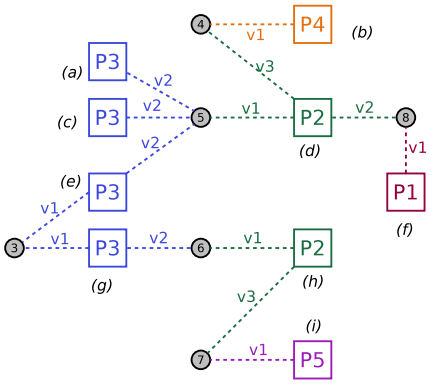
we are describing using this code $L_{\mathcal{L}}$, we encode how many occurrences ($\#occ$) it has in the pattern: we use a universal integer encoding for that. Finally, we must indicate what those occurrences are. A label symbol of arity n has a maximum $|V^P|^n$ occurrences: a vertex label symbol can appear on each vertex for a maximum of $|V^P|^1$ occurrences, an edge label symbol can appear between each pair of vertices, in each direction, and on each vertex as a self-loop, for a maximum of $|V^P|^2$ occurrences, and so on. Since we already encoded in the previous step the actual number of occurrences, we can encode their identity using $\log(\binom{|V^P|^n}{\#occ})$ *bits*.

tributions [Grü07].

Label symbol $l \in \mathcal{L}$	Arity	Occurrences in the data graph	Label code $L_{\mathcal{L}}(l)$
Book	1	3	3.46 bits
Person	1	2	4.04 bits
City	1	2	4.04 bits
Monument	1	2	4.04 bits
xsd:string	1	2	4.04 bits
xsd:integer	1	2	4.04 bits
Value:Alice	1	1	4.04 bits
Value:Bob	1	1	5.04 bits

Label value $l \in \mathcal{L}$	Arity	Occurrences in the data graph	Label code $L_{\mathcal{L}}(l)$
Value:123	1	2	4.04 bits
author	2	4	3.04 bits
name	2	2	4.04 bits
born_in	2	2	4.04 bits
died_in	2	2	4.04 bits
is_located	2	2	4.04 bits
near	2	2	4.04 bits
height	2	2	4.04 bits

Figure 6.7 – Codes for the label symbols $l \in \mathcal{L}$ for the graph of Fig. 6.3 (the table has been split in two for presentation purposes only). The length $L_{\mathcal{L}}$ of a code corresponds to a prefix code based on the number of occurrences of each label symbol.



Value to encode		Description length $L(D\|M) = L(G^R)$	
formula	concrete	formula	concrete
P	P_3	$L(C_P)$	$L(C_{P_3}) \approx 1.17$
$\#ports$	1	$\log(\|\Pi_P\| + 1)$	$\log(3)$
$ports$	$\{v_2\}$	$\sum_{\pi \in ports} L(C_\pi)$	$L(C_{v_2}) \approx 0.59$
$data\ vertices$	$\{5\}$	$\log(\binom{\|V^D\|}{\#ports})$	$\log(\binom{13}{1})$
P	P_4	$L(C_P)$	$L(C_{P_4}) \approx 3.17$
$\#ports$	1	$\log(\|\Pi_P\| + 1)$	$\log(2)$
$ports$	$\{v_1\}$	$\sum_{\pi \in ports} L(C_\pi)$	$L(C_{v_1}) = 0$
$data\ vertices$	$\{4\}$	$\log(\binom{\|V^D\|}{\#ports})$	$\log(\binom{13}{1})$
...+7 rows for embedding vertices (c) to (i)			

Figure 6.8 – How the rewritten graph of Fig. 6.5 is encoded in KG-MDL. Left: rewritten graph, where embedding vertices have been marked (a) to (i). Right: description length for embedding vertices (a) and (b).

Rewritten graph description length. The KG-MDL encoding of a rewritten graph (the "data encoded with the model" part of the MDL description length) is shown in Fig. 6.8 and presented hereafter. In order to encode a rewritten graph, we list all of its embedding vertices (each one representing a pattern occurrence) and how each of them connects to port vertices (i.e. how the pattern occurrence connects to the rest of the data). In order to do so, for each embedding vertex in the rewritten graph we first indicate to which pattern P it corresponds, using the pattern code C_P specified in the code table. Then we must encode how many ports ($\#ports$) this specific embedding has, i.e. how many of the pattern's vertices overlap with other pattern occurrences. Since the number of ports ranges from 0 to the maximum number of ports for the pattern $|\Pi_P|$, we can encode this value using $\log(|\Pi_P|+1)$ *bits*. Then, we encode the identities of those pattern vertices that are ports by listing their port codes C_π that can be found in the CT. Finally,

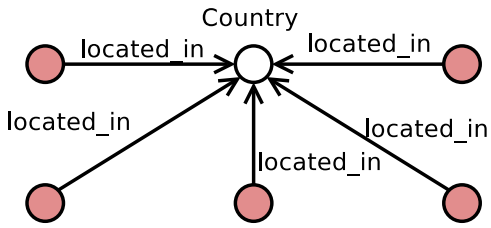


Figure 6.9 – The pattern *"N elements located in the same country"* for $N = 5$. Coloured vertices are equivalent and as such they increase the number of possible occurrences of the pattern in a combinatorial manner.

we indicate to which vertices of the data graph the ports correspond (so that it is possible to know when different patterns use the same data vertex as port). Since we already encoded in the previous step the actual number of ports $\#ports$ of this occurrence, we can simply encode this information using $\log(\binom{|V^D|}{\#ports})$ *bits* (where V^D are the data graph vertices).

Note that the encoding of this last element (i.e. the identities of the port vertices) differs from the encoding used in GraphMDL. This is because in GraphMDL, the identity of port vertices is encoded with a code of $\log(|V^R_{port}|)$ bits, with $|V^R_{port}|$ being the total number of port vertices in the rewritten graph. Therefore, the length of that encoding depends on the number of port vertices in the rewritten graph. As a consequence, when a pattern makes the number $|V^R_{port}|$ of port vertices in the rewritten graph grow —by using a port vertex that no other pattern uses— the encoding of the identity of *all* port vertices of *all patterns* grows in size. Therefore, the GraphMDL encoding favours patterns that use as ports the same vertices as other patterns. In KG-MDL, we changed the description length of this element in order to remove this potential source of bias. Our new encoding does not depend on the total number of port vertices in the rewritten graph, but on the number of vertices in the data graph, which is constant for all patterns. In conclusion, with this new encoding, patterns that use more ports are still discouraged —voluntarily as we want patterns with few connection points— but only because of their *number* of ports, and not the *identity* of those ports.

6.3 When Theory Meets Practice

In the previous sections we defined how KG-MDL measures the MDL description length of a set of patterns and its associated rewritten graph. The search algorithm used

by KG-MDL to generate and select patterns is the same as GraphMDL+ (see Section 5.4). Therefore, we could expect the implementation of this new approach to be just a matter of modifying the algorithm of Section 5.4 with the new description length definitions. However, that is not the case. Applying KG-MDL on real-life KGs triggers computational difficulties that are not observed when applying GraphMDL+ to the datasets of Chapters 4 and 5. The algorithm has a high memory usage and a long runtime. Based on our analysis, those difficulties arise because the approach generates some patterns that have a huge number of embeddings, and it tries to compute them all in order to compute the rewritten graph. This is because when creating the rewritten graph GraphMDL+ considers all CT rows one by one, and for each it runs through *all of its occurrences* in the data, in order to see if some parts of the data can be described with an occurrence of the pattern. An example of problematic pattern that appeared on knowledge graphs is the pattern *"N elements located in the same country"*, represented in Fig. 6.9. Because the data contains hundreds of countries with hundreds of elements each, the number of possible embeddings of this pattern is exponentially high. Thus the approach requires a long time and a lot of memory only for that pattern. This type of structure seems typical of KGs, which often have predicates used to "list" elements, and where the number of edges around vertices often follows a power law [GGD04].

Our first solution is to avoid computing all embeddings for each pattern and instead only compute the ones strictly needed for the rewritten graph. For each pattern in a code table, KG-MDL goes through the data graph, looking for parts of the data that are *not yet encoded by any pattern* and that are possible occurrences for the pattern. When one is found, KG-MDL marks that part as being encoded with the pattern. Since KG-MDL does not allow pattern occurrences to overlap, this strategy allows to greatly reduce the amount of embeddings to compute and store, since most of them are not needed.

This first solution manages to solve the memory usage problem, but does not completely solve the runtime one. For some complex patterns, the embedding search takes a long time to end. Most of the embeddings are found very quickly, but the remaining ones take a long time. Thus, our second solution is to add the possibility to specify a parameter that we call the *cover timeout*, which corresponds to a time limit for the embedding search on each CT pattern. This allows to enforce that a fairer share of time is allocated to patterns, avoiding that some complex patterns slow the whole computation. This is important in the context of an anytime approach such as KG-MDL, since time is an important resource and as such it is crucial to avoid spending too much time on a small

Table 6.1 – Characteristics of the datasets used in the experiments.

Dataset	#triples	KG-MDL graph modeling $\lvert V\rvert$	$\lvert V_{\mathcal{L}}\rvert$	$\lvert E_{\mathcal{L}}\rvert$	$\lvert\mathcal{L}\rvert$
SemanticBible-NTNames	4k	2k	4k	3k	1k
Lemon-dbpedia-en	30k	14k	17k	17k	2k
Mondial-europe	50k	22k	28k	34k	3k
Taaable	138k	82k	140k	115k	17k
Mondial	168k	74k	95k	114k	8k

part of the search space. We evaluate this solution experimentally in Section 6.4.1.

6.4 Experiments

In order to evaluate KG-MDL we developed a prototype based on the GraphMDL+ one. It has been developed in Java 1.8 and is available as a git repository [3]. Experiments were run on a single core of an Intel Xeon E5-2630 v3 CPU (our implementation is not multi-threaded), and had a RAM limit of 30GB (excepted for the Taaable dataset which had a limit of 40GB). Most of them actually required less than 15GB.

We conducted the experiments on five datasets. The **Lemon-dbpedia-en** dataset [4] is a lexicalisation of the most common DBpedia classes and properties using the "Lemon" ontology. It contains information such as *"The term* island *is a common noun whose sense is the* `dbo:Island` *class"*. In this dataset —because of the way it is used— we considered the `partOfSpeech` predicate equivalent to the `rdf:type` predicate. The **Mondial** dataset [5] contains geographical information, such as *"Mt Everest is a mountain in the Himalayas"*. The **Mondial-europe** dataset is the subset covering Europe only. Since these two datasets contain a large number of numerical values, we removed the concrete values and encoded them with a vertex only labeled by the type of the value (e.g. `xsd:Integer`). The **SemanticBible-NTNames** dataset [6] is a semantic web representation of named entities in the Bible with associated information, such as *"Adam is a man, spouse of Eve, parent of Abel, Cain and Seth"*. The **Taaable** dataset [7] corresponds to an RDF representation of the Wikitaaable semantic wiki [Cor+14], containing recipe and food information

3. `https://gitlab.inria.fr/fbariatt/graphmdl`
4. `https://lemon-model.net/lexica/dbpedia_en/`
5. `https://www.dbis.informatik.uni-goettingen.de/Mondial/`
6. `http://www.semanticbible.com/ntn/ntn-overview.html`
7. We obtained the data from the authors of [Cor+14].

Table 6.2 – Results of running KG-MDL on the different datasets for 8h with a cover timeout of 200ms, 500ms and 1s. Only the best timeout is reported for each dataset.

Dataset	Cover timeout	$\|CT_{kgmdl}\|$	%labels described by patterns	Compression ratio $L\% = \frac{L(CT_{kgmdl},D)}{L(CT_0,D)}$	Time to reach 95% of L% gain
NTNames	200ms	394	87.3%	57.6%	2h36m
Lemon	500ms	165	94.7%	25.4%	0h06m
Mondial-europe	200ms	100	89.3%	39.7%	2h03m
Taaable	200ms	67	82.7%	50.2%	2h59m
Mondial	500ms	59	85.8%	45.3%	4h25m

such as *"Bread is a baked good made using 2 teaspoons of salt"*.

The characteristics of the datasets are presented in Table 6.1. For each dataset we report the number of triples in its original RDF representation and the number of vertices, vertex labels, edges, and the total number of label symbols in their KG-MDL representation (presented in Section 6.1). For example the Lemon dataset is composed of 30 thousand triples, which in KG-MDL correspond to 14 thousand vertices having a total of 17 thousand vertex labels, connected by 17 thousand edges; in total there are 2 thousand unique label symbols in the dataset.

6.4.1 Quantitative Evaluation

Table 6.2 presents the results obtained when running KG-MDL on the different datasets. For each dataset three experiments were run with a cover timeout of 200ms, 500ms, and 1s respectively. Each experiment was given 8 hours. We only report in the table the experiment that gave the best (i.e. the lowest) description length at the end of the 8 hours (we evaluate the impact of the cover timeout further along this section). The first column of the table reports the dataset, and the second one reports the timeout for that experiment.

Number of generated patterns. The third column of Table 6.2 ($|CT_{kgmdl}|$) reports the number of patterns in the best code table found by KG-MDL. We observe that on all datasets, even on the biggest ones, the number of generated patterns is in the lower hundreds. Thus, we can conclude that KG-MDL is effective at its goal of generating a *small and human-sized* set of patterns.

Labels described by patterns. The fourth column of Table 6.2 reports how many of each dataset's labels are described by instances of the KG-MDL patterns (i.e. not by singletons). We observe that on all datasets at least 80% of the labels are described by

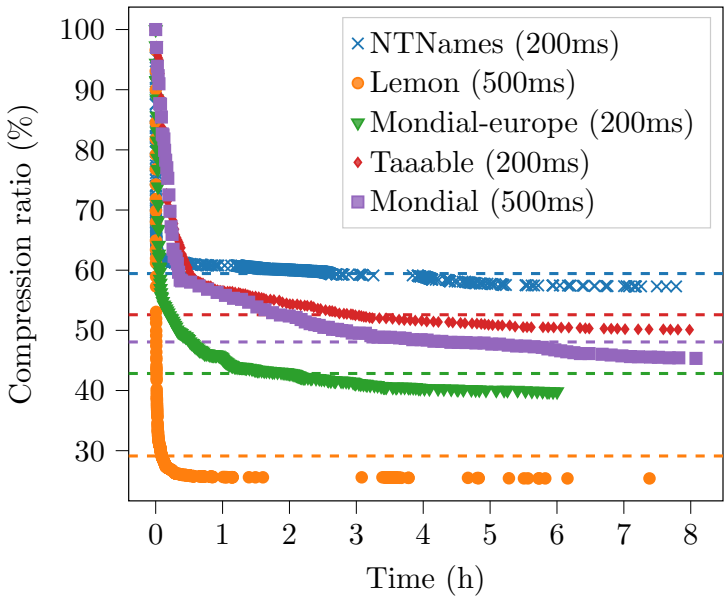


Figure 6.10 – Evolution of the compression ratio given by the CT found by KG-MDL as a function of time. Dotted lines represent for each dataset the compression ratio that is 95% of the way w.r.t. the full 8 hours experiment.

a pattern, which shows that KG-MDL patterns are descriptive enough to cover most of the data. When looking at the results in detail, we notice that most of the non-covered labels correspond to concrete literal values (e.g. “123” or “Alice”). This is consistent with the idea of finding patterns to describe the *structure* of the data, without caring about the concrete values of the different properties.

Description length. The fifth column of Table 6.2 reports the compression ratio $L\% = \frac{L(CT_{kgmdl},D)}{L(CT_0,D)}$, corresponding to the ratio between the description length given by the code table found at the end of the experiment and the one given by the initial singleton-only code table CT_0. This is the MDL measure that KG-MDL tries to minimize. We observe that the approach is efficient in doing so, obtaining a compression ratio that ranges from 25% to 58% depending on the dataset. From an information theory point of view, that means that KG-MDL manages to find a set of patterns that makes it possible to describe the data with a quarter to half as much information than without those patterns.

Description length evolution over time. We expect that a user would probably not want to wait for 8 hours in order to get a result from their experiments. Since KG-MDL is an anytime algorithm, it can be stopped at any point during the search, yielding the best result found so far. In order to evaluate the impact of stopping the algorithm sooner,

Table 6.3 – Compression ratio on the Mondial dataset for different cover timeouts (8h experiments).

Timeout per CT row during search	Compression ratio
50ms	52.8%
100ms	49.3%
200ms	49.8%
500ms	**45.3%**
1s	47.0%
5s	48.6%
10s	49.3%

we show in Fig. 6.10 the evolution of description length over time. Each point of the plot corresponds to one iteration of the search algorithm, i.e. to a code table that is found which is better than the previous one. We observe that most of the gain is obtained in the first few hours. For instance, on the Mondial-europe dataset, the compression ratio goes from 100% (the starting model) to around 45% in 2 hours, and the last 6 hours are spent going from 45% to 40%. We report in the last column of Table 6.2 the time needed in each experiment to attain a compression ratio that is 95% of the way w.r.t. the full 8 hours experiment. We observe that for all datasets, KG-MDL obtains good results fairly quickly, which means that the user can usually stop the algorithm quite early and still obtain good results.

Impact of the cover timeout. In order to evaluate the impact of the cover timeout parameter described in Section 6.3, we ran some additional experiments on the Mondial dataset, whose results are shown in Table 6.3. We observe that a timeout too low or too high can negatively impact the final description length. We believe that too low a timeout on some patterns can stop the embedding computation before some important occurrences are found, which makes the algorithm under-estimate useful patterns. On the other hand, too high a timeout entails the risk of spending so much time on the embedding computation of a few complex patterns that it will limit the amount of different patterns that can be processed in a given time. In conclusion, the cover timeout is a useful parameter, which makes a difference in the performances of the algorithm.

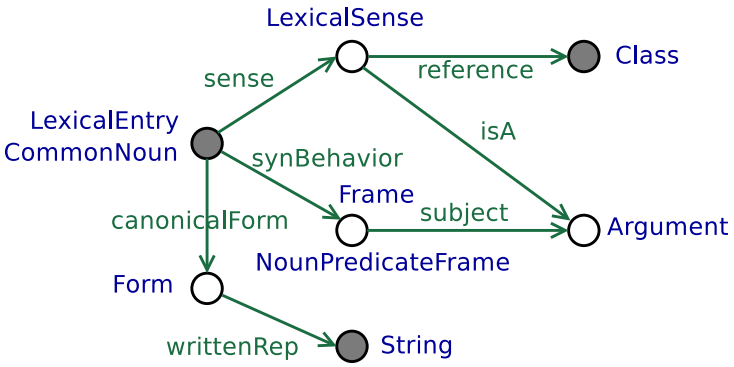


Figure 6.11 – The pattern that appears the most in the KG-MDL rewritten graph on the Lemon dataset. Vertices filled in grey are the ports of the pattern.

6.4.2 Qualitative Evaluation

In this section we present some of the patterns extracted by KG-MDL. Our goal is to present evidence that KG-MDL patterns are relevant to their KG. Meaning that they highlight the *schema* used for creating the data, but also show statistical regularities in the data that go beyond schema patterns and instead stem from the *usage* of the properties and classes defined in the schema.

Lemon patterns. By looking at the github page of the Lemon dataset [8], we observe that the dataset is created using some predefined structures called *lemon design patterns*. KG-MDL does not have any knowledge of those design patterns, but all of them (around ten different types of design patterns are used in the Lemon-dbpedia-en dataset) are found in the set of extracted patterns, demonstrating the ability of our approach to identify them as important structures in the data. Fig. 6.11 shows the pattern that appears the most in the KG-MDL rewritten graph on the lemon dataset: it correspond to the lemon design pattern "ClassNoun". This structure describes a lexical entry that is a common noun (e.g. *"lake"*), and whose meaning is a class (e.g. `dbo:Lake`). The structure used in Lemon to encode this entry is not trivial: the pattern contains 7 vertices, 9 vertex labels, and 7 edge labels. The pattern combines three linguistic aspects: lexical (the written representation of the term), syntactic (a noun predicate frame with one argument, the subject), and semantic (the sense is given as a reference to an RDFS class). Syntax and semantics are aligned through the Argument vertex, which links the subject of the frame and the

8. `https://github.com/ag-sc/lemon.dbpedia`

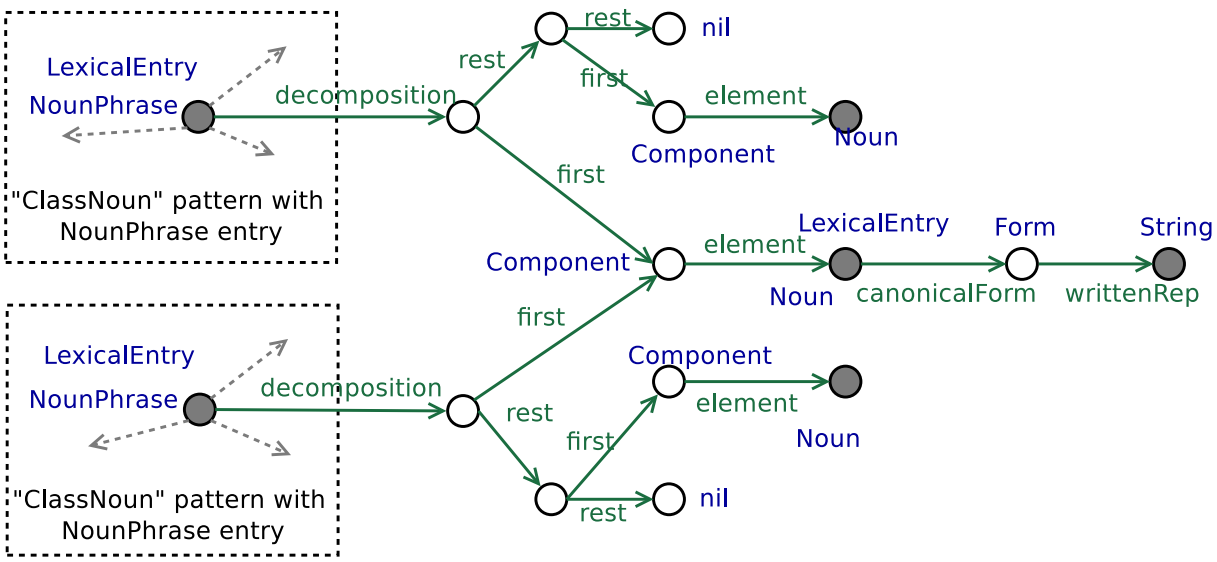


Figure 6.12 – A pattern extracted by KG-MDL on the Lemon dataset. The squares on the left correspond to the structure of Fig. 6.11 (not shown in order to simplify the image) except for label "CommonNoun" being replaced by label "NounPhrase".

instance of the class. Only three out of the seven pattern vertices —filled in grey in the figure— appear as ports in the rewritten graph: the lexical entry, the class, and the written representation. They correspond to the vertices that differ from one pattern occurrence to another, i.e. they have different additional labels, either vertex labels or edges connecting them to other pattern occurrences. Hence, each pattern occurrence is fully determined by the choice of three data vertices. This is close to the lemon design pattern where only two pieces of information need to be provided per instance: the canonical form and the RDFS class. However some lexical entries can be part of more complex lexical entries (e.g. noun phrases), hence why that vertex is also a port.

Fig. 6.12 shows another pattern extracted from the Lemon dataset. Contrary to the previous one, this pattern does not correspond directly to a lemon design pattern (what we could call the "schema" of the graph). It represents a pair of lexical entries of type "NounPhrase" (i.e. terms composed by multiple elements) that *share the same first element*, but whose meaning are distinct classes, e.g. `dbo:BloodVessel` and `dbo:BloodType`, `dbo:RadioProgram` and `dbo:RadioStation`, or `dbo:MountainRange` and `dbo:MountainPass`. This patterns tells us that it is common to have terms with different meanings but whose lexical form starts with the same word. Indeed the pattern has 34 occurrences in the rewritten graph (i.e. it describes 68 lexical entries). By finding this

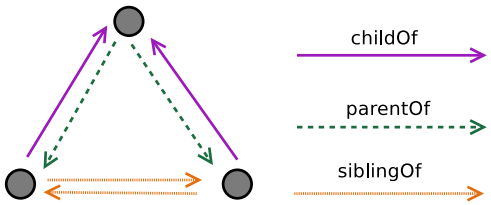


Figure 6.13 – A pattern extracted by KG-MDL on the NTNames dataset. Vertices filled in grey are the ports of the pattern.

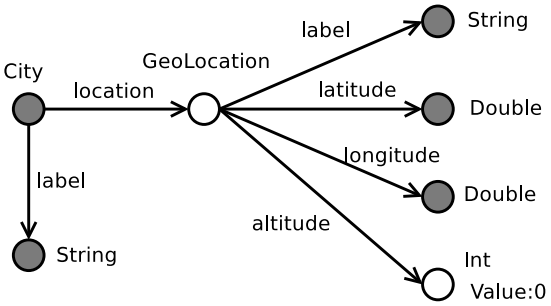


Figure 6.14 – A pattern extracted by KG-MDL on the NTNames dataset. Vertices filled in grey are the ports of the pattern.

pattern, KG-MDL gives insights about the combined *usage* of design patterns in the data.

NTNames patterns. Fig. 6.13 shows one of the patterns extracted by KG-MDL on the NTNames dataset. In the schema of the dataset `childOf` and `parentOf` are marked as inverse properties, and `siblingOf` is marked as symmetric. KG-MDL generates a pattern that strongly suggests those axioms without having access to the schema itself, just by observing the data. The algorithm also suggest an axiom that is not present in the schema of the dataset: when two people have the same parent, they are siblings one of the other. The user could use this pattern in order to validate the data, or by inferring missing edges using this new knowledge.

Fig. 6.14 shows another pattern extracted from the NTNames dataset. By looking at this pattern we can understand how the geographical information of cities is represented in the data, i.e. learn something about the schema of the KG. There is an interesting detail in this pattern: the altitude vertex is not a port and has a value of 0. This may seem strange, but when looking at the cities in the dataset it is actually the case that they all have altitude 0. This is probably due to a lack of information, which however would not have been visible by looking at the schema of the KG alone. The patterns extracted by KG-MDL are extracted from the actual data, so they can highlight this type of information related to the actual *usage* of properties and classes.

Take-Home Message

In this chapter we presented KG-MDL, an adaptation of GraphMDL+ to handle knowledge graphs. The approach can also be easily extended to hypergraphs, since its MDL description length definitions (see Section 6.2) are not limited to graph labels with arity ≤ 2 (i.e. vertices and edges). The adaptation from labeled graphs to knowledge graphs is not trivial, mainly because —even if they are often represented as "graphs"— KGs have some peculiar characteristics that distinguish them from the graphs usually found in the graph mining domain (see Sections 6.1 and 6.3). We evaluated our approach experimentally on medium-sized KGs (see Section 6.4), showing that it is efficient in finding sets of patterns that are good according to its MDL evaluation criterion, and that it does so in reasonably short time. We showed that the extracted patterns cover most of the data, and that they give insights about both the *schema* of the KG, and the *concrete usage* of the schema that is done in the data.

Chapter 7

Conclusion and Perspectives

In this thesis we addressed a well-known problem in pattern mining: the *pattern explosion problem*. That is, classic pattern mining approaches usually extract sets of patterns that are too large for human analysis (pattern sets with millions of patterns are not unusual). We addressed this problem in the context of *graph pattern mining*, where both the patterns and the data are labeled graphs. In order to do so, we proposed three graph pattern mining approaches based on the Minimum Description Length principle [Grü07; Ris78], that are capable to extract a *small and human-sized* set of *descriptive* patterns. For that, we instantiated the Minimum Description Length principle in a graph pattern mining context, designing *description length measures* that are used as a criterion to evaluate candidate sets of patterns. Designing such measures requires to: define a family of *models* that are capable to *encode the data*; define a way for a specific model to encode the data; and define a way to measure the amount of information needed by such a model and its data encoding. The model that we use is inspired from the *code table* structure used by the Krimp algorithm [VLS11], a MDL-based itemset mining approach working on transactional data. The proposed model corresponds to a set of candidate patterns and information associated to each pattern, so that the patterns can be used to encode the data. In order to correctly and losslessly encode the data using this model, we introduced the notions of *ports* and *rewritten graph*, which allow to exactly describe the data as a composition of pattern occurrences overlapping on vertices. They also allow to identify which vertices are at the interface of a pattern or internal to it, thus helping their interpretation. We proposed heuristic algorithms that exploit those definitions in order to efficiently extract small sets of patterns from the data. We showed through experiments that the proposed approaches allow for efficient selection of patterns which are descriptive of the data, and that they convey useful information to the user.

The advantages of our proposals are the following. First, they evaluate sets of patterns *as a whole* instead of evaluating each pattern individually. As such they take into account interactions between patterns. This is a usual advantage of MDL-based approaches over

more classic pattern mining approaches [Gal20]. Second, they do not impose any constraint on the structure of the extracted patterns. This is an advantage over other MDL-based graph mining approaches [Kou+15; Goe+16], that can only extract patterns with predefined shapes. Finally, they do not require any additional parameters except the data from which to extract patterns (with the exception of one of them —GraphMDL— which also requires a set of candidate patterns). As such, they can be directly applied to the data without the need for parameter tuning.

The first approach that we proposed, called GraphMDL (see Chapter 4), selects a descriptive subset of patterns from a set of patterns given as input. As such, it depends on an external approach to first generate those patterns. The second proposed approach, called GraphMDL+ (see Chapter 5), interleaves a pattern generation process with the pattern selection. Therefore, it does not need an external approach to generate patterns, giving it more freedom about the patterns it generates. As a consequence, it is significantly faster than GraphMDL, and is a *parameterless* and *anytime* algorithm, as the only input it needs is the data from which the patterns should be extracted. The third approach that we proposed, called KG-MDL (see Chapter 6), is an extension of GraphMDL+ targeted specifically at knowledge graphs. Because of the differences between KGs and classic labeled graphs used in pattern mining (which are discussed in Chapter 6), the MDL measure and algorithm that we proposed for GraphMDL and GraphMDL+ must be revised. Notably because KGs are multigraphs (i.e. there can be multiple edges between a pair of vertices), and because they do not have the same distribution of edges around vertices (i.e. many vertices have few edges and some have a large amount). We defined this new MDL measure in such a way that it can also easily be applied to hypergraphs, a generalization of the concept of graphs with more-than-binary edges. Finally, we also proposed GraphMDL Visualizer (see Section 4.5), a tool to visualize graph patterns that can be directly applied to the results of our approaches. It helps the user better understand the data via an interactive visualization of the patterns and the way that they can be used to encode the data.

7.1 Perspectives

In this section we give three perspectives, representing possible future research directions in the domain of MDL-based graph pattern mining, using the approaches presented in this thesis as a starting point.

7.1.1 Hierarchical Patterns

An interesting extension of our work would be to allow patterns to be defined as containing other patterns, e.g. by saying "pattern P_1 is `A-B-C`, pattern P_2 is pattern P_1 plus `D`, i.e. `A-B-C-D`". Smaller patterns could be viewed as building blocks for larger patterns, or conversely larger pattern could be seen as specialisations of smaller patterns. This would allow a hierarchical view of the patterns. By choosing a level in this hierarchy of patterns, the user would be able to choose a level of abstraction to represent the data. For example we could imagine on a graph representation of a text corpus, to have patterns that represents groups of words, that are then used in patterns that represent sentence structure, that are then used in patterns that represent paragraph structure, and so on.

While the approaches that we proposed in this thesis do allow a pattern P_2 to have the same structure as a pattern P_1 plus something, this is not taken into account in the encoding. Indeed, in the code table patterns P_1 and P_2 would both list all of their vertices and edges. There is currently no way for P_2 to express in its structure "here include all the structure of P_1". Thus, recognizing that P_2 is a specialisation of P_1 is a task that is done as post-processing by the user. In particular the user must remember that all occurrences of P_2 should also be interpreted as occurrences of P_1 plus a specialisation.

Since our approaches not only output the selected patterns but also a representation of the data as a composition of pattern occurrences (which have a graph structure as well), a way of achieving hierarchical patterns could be by applying the approach iteratively on its previous results. Previous pattern occurrences could be simply treated as new hyper-edges covering parts of the data.

7.1.2 Extensions to the Pattern Language

Another extension of the work presented in this thesis could be to extend the expressiveness of pattern structures, in order to encode some concepts in a way that is closer to human understanding. We present here two possible extensions:

- Allow to express the concept of "*several* neighbours with structure X". In our approaches, patterns must express structures precisely, as they are matched exactly to the data. This can lead to patterns that do not feel natural to a human user. For instance, on a family tree, our approaches could potentially both extract the pattern "a person with two children", "a person with three children" and "a person with four children". If a human was tasked to describe the data instead, they would

probably express the more general idea "a person with several children". Therefore, a possible extension of our approaches could be to allow patterns to not only express *exact counts* of vertices in their structures, but also express this concept of *several vertices*. The exact number of vertices could then be encoded when the pattern is used to encode the data. Note however that it would be important to still allow patterns to express exact counts: for example if in the family tree most people had exactly two children, an exact pattern would convey more information by saying that having exactly two children is common enough to be granted its own pattern. Therefore, choosing between the more precise "exact count" pattern and the less precise "several" pattern could be included as a task to be done using the MDL principle.

- Allow patterns to express in their structure "regular expressions" that indicate chains of edges. For example we could imagine a pattern describing the concept of *ancestor* in a family tree as $v_1 \xrightarrow{parent+} v_2$, where there could be any number of intermediate vertices between v_1 and v_2, all connected into a chain by edges labeled *parent*. In the Semantic Web, there exists a similar concept called *property paths*, which can be used in the SPARQL language to query knowledge graphs.

7.1.3 Quantitative Evaluation of MDL Approaches

When working with MDL-based pattern mining approaches, it is often difficult to provide convincing and objective quantitative evaluations [Gal20]. While qualitative evaluations of the extracted patterns are valuable to show their usefulness, they often need an expertise on the domain of the data to truly convince. Quantitative evaluations, on the contrary, can be a powerful tool to show the efficiency of the approach in a way that can be accepted objectively. However, performing such evaluations on MDL-based approaches is often difficult. Since different approaches define different description length measures, comparing their raw description lengths is not meaningful. Since compression ratios depend on both the MDL description length definitions and the baseline used, comparing their raw values across different approaches is not meaningful either. On top of that, most approaches use search algorithms that are greedy and guided by heuristics, so there is often no gold standard about the patterns that should be extracted by such an approach on a specific dataset.

The most common way to provide objective quantitative evaluations for MDL-based

pattern mining approaches, is to use the extracted patterns to perform some kind of task that can be evaluated objectively. For example, in Chapter 4 we evaluated our approach by performing a classification task. This same evaluation was used in [VLS11]. Other possible tasks can be, for example, using synthetic datasets with known ground-truth patterns (used in [FL20]) or domain-specific tasks such as link prediction and error detection in knowledge graphs (used in [Bel+20]). The difficulty in performing such tasks is that the adaptation of the results of the approach to the task is not always straightforward. For example, when performing a classification task in Chapter 4, we ran into the problem that the patterns selected by the approach were not specifically selected with a classification goal in mind. The consequence was that while the patterns where *descriptive* of their respective classes, they were not *discriminative* enough to differentiate between classes with performances competitive with state-of-the-art classifiers. Thus, possible extensions of our approaches would be to add directly into the mining process or the MDL selection some specializations targeting tasks that can be quantitatively evaluated. This could allow to select patterns that are better suited for the evaluation that is performed, and as such that can be compared with state-of-the-art approaches performing the same tasks using different techniques.

While performing objectively convincing quantitative evaluations is important in order to evaluate an approach, one should not forget the initial goal, which is to extract descriptive patterns. A danger with focusing on a specific evaluation task is to create an approach that is only capable of such a task and nothing else. If the goal of the approach is to be a *general* pattern mining approach, allowing a human to explore data, care should be taken not to jeopardize this generality in favour of a specific task. Thus, a correct balance between specialisation and generality must be found when designing such extensions.

BIBLIOGRAPHY

[Agr+96] Rakesh Agrawal et al., « Fast discovery of association rules. », *in*: *Advances in knowledge discovery and data mining* 12.*1* (1996), pp. 307–328 (cit. on pp. 6, 14, 36, 38, 39).

[AIS93] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami, « Mining association rules between sets of items in large databases », *in*: *ACM SIGMOD Record* 22.*2* (June 1993), pp. 207–216, DOI: 10.1145/170036.170072 (cit. on pp. 5, 13, 36).

[AVH04] Grigoris Antoniou and Frank Van Harmelen, *A semantic web primer*, MIT press, 2004 (cit. on p. 26).

[Bay98] Roberto J. Bayardo, « Efficiently mining long patterns from databases », *in*: *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, SIGMOD '98, Association for Computing Machinery, June 1998, pp. 85–93, DOI: 10.1145/276304.276313 (cit. on pp. 6, 14, 45).

[BB02] C. Borgelt and M. R. Berthold, « Mining molecular fragments: finding relevant substructures of molecules », *in*: *2002 IEEE International Conference on Data Mining, ICDM 2002. Proceedings.* Dec. 2002, pp. 51–58, DOI: 10.1109/ICDM.2002.1183885 (cit. on pp. 5, 13, 40, 42).

[BBL16] Molood Barati, Quan Bai, and Qing Liu, « SWARM: An Approach for Mining Semantic Association Rules from Semantic Web Data », *in*: *PRICAI 2016: Trends in Artificial Intelligence*, Lecture Notes in Computer Science, Springer International Publishing, 2016, pp. 30–43, DOI: 10.1007/978-3-319-42911-3_3 (cit. on p. 43).

[BCF20a] Francesco Bariatti, Peggy Cellier, and Sébastien Ferré, « GraphMDL : sélection de motifs de graphes avec le principe MDL », *in*: *Revue des Nouvelles Technologies de l'Information* Extraction et Gestion des Connaissances , RNTI-E-36 (2020), pp. 37–48 (cit. on p. 54).

[BCF20b] Francesco Bariatti, Peggy Cellier, and Sébastien Ferré, « GraphMDL: Graph Pattern Selection Based on Minimum Description Length », *in*: *Advances in Intelligent Data Analysis XVIII*, ed. by Michael R. Berthold, Ad Feelders, and Georg Krempl, Lecture Notes in Computer Science, Springer International Publishing, 2020, pp. 54–66, DOI: 10.1007/978-3-030-44584-3_5 (cit. on p. 54).

[BCF20c] Francesco Bariatti, Peggy Cellier, and Sébastien Ferré, « GraphMDL Visualizer: Interactive Visualization of Graph Patterns », *in*: *GEM: Graph Embedding and Mining Workshop @ ECML-PKDD 2020*, Sept. 2020, URL: https://hal.inria.fr/hal-03142207 (cit. on p. 54).

[BCF21] Francesco Bariatti, Peggy Cellier, and Sébastien Ferré, « GraphMDL+: interleaving the generation and MDL-based selection of graph patterns », *in*: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, SAC '21, Association for Computing Machinery, Mar. 2021, pp. 355–363, DOI: 10.1145/3412841.3441917 (cit. on p. 80).

[Bel+20] Caleb Belth et al., « What is Normal, What is Strange, and What is Missing in a Knowledge Graph: Unified Characterization via Inductive Summarization », *in*: *Proceedings of The Web Conference 2020*, WWW '20, Association for Computing Machinery, Apr. 2020, pp. 1115–1126, DOI: 10.1145/3366423.3380189 (cit. on pp. 7, 15, 43, 50, 51, 121).

[BN08] Björn Bringmann and Siegfried Nijssen, « What Is Frequent in a Single Graph? », *in*: *Advances in Knowledge Discovery and Data Mining*, vol. 5012, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2008, pp. 858–863, DOI: https://doi.org/10.1007/978-3-540-68125-0_84 (cit. on pp. 40, 41, 63).

[Bob+20] Carlos Bobed et al., « Data-driven assessment of structural evolution of RDF graphs », *in*: *Semantic Web* 11.*5* (Jan. 2020), pp. 831–853, DOI: 10.3233/SW-200368 (cit. on p. 43).

[BR20] Peter Bloem and Steven de Rooij, « Large-scale network motif analysis using compression », *in*: *Data Mining and Knowledge Discovery* 34.*5* (Sept. 2020), pp. 1421–1453, DOI: 10.1007/s10618-020-00691-y (cit. on pp. 7, 15, 50, 51).

[CH93] D. J. Cook and L. B. Holder, « Substructure Discovery Using Minimum Description Length and Background Knowledge », *in*: *Journal of Artificial Intelligence Research* 1 (1993), pp. 231–255, DOI: 10.1613/jair.43 (cit. on pp. 7, 15, 40, 50, 53).

[Cor+14] Amélie Cordier et al., « Taaable: A Case-Based System for Personalized Cooking », *in*: *Successful Case-based Reasoning Applications-2*, Studies in Computational Intelligence, 2014, pp. 121–162 (cit. on p. 109).

[DL17] Vladimir Dzyuba and Matthijs van Leeuwen, « Learning what matters–sampling interesting patterns », *in*: *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, Springer, 2017, pp. 534–546 (cit. on pp. 6, 14, 44).

[Eli75] P. Elias, « Universal codeword sets and representations of the integers », *in*: *IEEE Transactions on Information Theory* 21.*2* (Mar. 1975), pp. 194–203, DOI: 10.1109/TIT.1975.1055349 (cit. on p. 33).

[FL20] Micky Faas and Matthijs van Leeuwen, « Vouw: Geometric Pattern Mining Using the MDL Principle », *in*: *Advances in Intelligent Data Analysis XVIII*, vol. 12080, Springer International Publishing, Apr. 2020, pp. 158–170, DOI: 10.1007/978-3-030-44584-3_13 (cit. on pp. 7, 15, 49, 121).

[For96] Scott Fortin, « The Graph Isomorphism Problem », *in*: (July 1996), p. 25 (cit. on pp. 7, 15, 23).

[FPSS96] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth, « From Data Mining to Knowledge Discovery in Databases », *in*: *AI Magazine* 17.*3* (Mar. 1996), pp. 37–37, DOI: 10.1609/aimag.v17i3.1230 (cit. on p. 35).

[Gal+15] Luis Galárraga et al., « Fast rule mining in ontological knowledge bases with AMIE+ », *in*: *The VLDB Journal* 24.*6* (Dec. 2015), pp. 707–730, DOI: 10.1007/s00778-015-0394-1 (cit. on p. 43).

[Gal+19] Esther Galbrun et al., « Mining Periodic Patterns with a MDL Criterion », *in*: *Machine Learning and Knowledge Discovery in Databases*, Lecture Notes in Computer Science, Springer International Publishing, 2019, pp. 535–551, DOI: 10.1007/978-3-030-10928-8_32 (cit. on pp. 7, 15, 49).

[Gal20] Esther Galbrun, « The Minimum Description Length Principle for Pattern Mining: A Survey », *in*: *arXiv:2007.14009 [cs, math]* (July 2020), URL: http://arxiv.org/abs/2007.14009 (cit. on pp. 7, 15, 47, 49, 50, 55, 118, 120).

[GGD04] Rosa Gil, Roberto García, and Jaime Delgado, « Measuring the semantic web », *in*: *AIS SIGSEMIS Bulletin* 1.*2* (2004), pp. 69–72 (cit. on pp. 43, 99, 108).

[GH06] Liqiang Geng and Howard J. Hamilton, « Interestingness measures for data mining: A survey », *in*: *ACM Computing Surveys* 38.*3* (Sept. 2006), 9–es, DOI: 10.1145/1132960.1132963 (cit. on pp. 6, 14, 44).

[Goe+16] S. Goebl et al., « MeGS: Partitioning Meaningful Subgraph Structures Using Minimum Description Length », *in*: *2016 IEEE 16th International Conference on Data Mining (ICDM)*, Dec. 2016, pp. 889–894, DOI: 10.1109/ICDM.2016.0108 (cit. on pp. 50, 54, 118).

[Grü07] Peter D Grünwald, *The minimum description length principle*, MIT press, 2007 (cit. on pp. 6, 14, 32, 105, 117).

[GSV06] E. Gudes, S. E. Shimony, and N. Vanetik, « Discovering Frequent Graph Patterns Using Disjoint Paths », *in*: *IEEE Transactions on Knowledge and Data Engineering* 18.*11* (Nov. 2006), pp. 1441–1456, DOI: 10.1109/TKDE.2006.173 (cit. on p. 41).

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin, « Mining Frequent Patterns Without Candidate Generation », *in*: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, ACM, 2000, pp. 1–12, DOI: 10.1145/342009.335372 (cit. on pp. 36, 39, 42).

[Hua+04] Jun Huan et al., « SPIN: Mining Maximal Frequent Subgraphs from Graph Databases », *in*: *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, ACM, 2004, pp. 581–586, DOI: 10.1145/1014052.1014123 (cit. on pp. 40, 45).

[HWP03] J. Huan, W. Wang, and J. Prins, « Efficient mining of frequent subgraphs in the presence of isomorphism », *in*: *Third IEEE International Conference on Data Mining*, Nov. 2003, pp. 549–552, DOI: 10.1109/ICDM.2003.1250974 (cit. on pp. 40, 43).

[IWM00] Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda, « An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data », *in*: *Principles of Data Mining and Knowledge Discovery*, vol. 1910, Lecture Notes

in Computer Science, Springer Berlin Heidelberg, 2000, pp. 13–23 (cit. on pp. 40, 41).

[JCZ13] Chuntao Jiang, Frans Coenen, and Michele Zito, « A survey of frequent subgraph mining algorithms », *in*: *The Knowledge Engineering Review* 28.*1* (Mar. 2013), pp. 75–105, DOI: 10.1017/S0269888912000331 (cit. on pp. 6, 14, 21, 24, 41, 45).

[JK07] T. Junttila and P. Kaski, « Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs », *in*: *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, Society for Industrial and Applied Mathematics, Jan. 2007, pp. 135–149, DOI: 10.1137/1.9781611972870.13 (cit. on p. 89).

[JK19] Wolfgang Jentner and Daniel A Keim, « Visualization and visual analytic techniques for patterns », *in*: *High-Utility Pattern Mining*, Springer, 2019, pp. 303–337 (cit. on p. 70).

[KK01] M. Kuramochi and G. Karypis, « Frequent subgraph discovery », *in*: *Proceedings 2001 IEEE International Conference on Data Mining*, Nov. 2001, pp. 313–320, DOI: 10.1109/ICDM.2001.989534 (cit. on pp. 40, 41).

[KK05] Michihiro Kuramochi and George Karypis, « Finding Frequent Patterns in a Large Sparse Graph* », *in*: *Data Mining and Knowledge Discovery* 11.*3* (Nov. 2005), pp. 243–271, DOI: 10.1007/s10618-005-0003-9 (cit. on pp. 40–42).

[Kou+15] Danai Koutra et al., « Summarizing and understanding large graphs », *in*: *Statistical Analysis and Data Mining: The ASA Data Science Journal* 8.*3* (June 2015), pp. 183–202, DOI: 10.1002/sam.11267 (cit. on pp. 7, 15, 40, 50, 54, 118).

[KS09] Arne Koopman and Arno Siebes, « Characteristic Relational Patterns », *in*: *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, ACM, 2009, pp. 437–446, DOI: 10.1145/1557019.1557071 (cit. on pp. 7, 15, 49).

[KSM12] Hadi Katebi, Karem A. Sakallah, and Igor L. Markov, « Graph Symmetry Detection and Canonical Labeling: Differences and Synergies », *in*: *arXiv:1208.6271 [cs, math]* (Aug. 2012), URL: http://arxiv.org/abs/1208.6271 (cit. on p. 26).

[Lee01] Thomas C. M. Lee, « An Introduction to Coding Theory and the Two-Part Minimum Description Length Principle », *in*: *International Statistical Review* 69.*2* (2001), pp. 169–183, DOI: 10.1111/j.1751-5823.2001.tb00455.x (cit. on p. 33).

[Leh+15] Jens Lehmann et al., « Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia », *in*: *Semantic web* 6.*2* (2015), pp. 167–195 (cit. on p. 26).

[May99] Wolfgang May, *Information Extraction and Integration with Florid: The Mondial Case Study*, tech. rep. 131, Available from http://dbis.informatik.uni-goettingen.de/Mondial, Universität Freiburg, Institut für Informatik, 1999 (cit. on p. 26).

[Ng+98] Raymond T. Ng et al., « Exploratory mining and pruning optimizations of constrained associations rules », *in*: *ACM SIGMOD Record* 27.*2* (June 1998), pp. 13–24, DOI: 10.1145/276305.276307 (cit. on pp. 6, 14, 44).

[NK04] Siegfried Nijssen and Joost N. Kok, « A Quickstart in Frequent Structure Mining Can Make a Difference », *in*: *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, ACM, 2004, pp. 647–652, DOI: 10.1145/1014052.1014134 (cit. on pp. 40, 42).

[Pas+99] Nicolas Pasquier et al., « Discovering Frequent Closed Itemsets for Association Rules », *in*: *Database Theory — ICDT'99*, Lecture Notes in Computer Science, Springer, 1999, pp. 398–416, DOI: 10.1007/3-540-49257-7_25 (cit. on pp. 6, 14, 45).

[Pau17] Heiko Paulheim, « Knowledge graph refinement: A survey of approaches and evaluation methods », *in*: *Semantic Web* 8.*3* (Jan. 2017), pp. 489–508, DOI: 10.3233/SW-160218 (cit. on p. 43).

[Ram+14] R Ramezani et al., « SWApriori: a new approach to mining Association Rules from Semantic Web Data », *in*: *Journal of Computing and Security* (2014) (cit. on p. 43).

[RBB19] Bastian Rieck, Christian Bock, and Karsten Borgwardt, « A Persistent Weisfeiler-Lehman Procedure for Graph Classification », *in*: *Proceedings of the 36th International Conference on Machine Learning*, vol. 97, Proceedings of Ma-

chine Learning Research, Springer Berlin Heidelberg, June 2019, pp. 5448–5458 (cit. on pp. 68, 69).

[Ris78] J. Rissanen, « Modeling by shortest data description », *in*: *Automatica* 14.*5* (Sept. 1978), pp. 465–471, DOI: 10.1016/0005-1098(78)90005-5 (cit. on pp. 6, 14, 32, 117).

[RP15] T. Ramraj and R. Prabhakar, « Frequent Subgraph Mining Algorithms – A Survey », *in*: *Procedia Computer Science*, Graph Algorithms, High Performance Implementations and Its Applications (ICGHIA 2014) 47 (Jan. 2015), pp. 197–204, DOI: 10.1016/j.procs.2015.03.198 (cit. on pp. 6, 14, 41).

[RZ07] Luc De Raedt and Albrecht Zimmermann, « Constraint-Based Pattern Set Mining », *in*: *Proceedings of the 2007 SIAM International Conference on Data Mining (SDM)*, Proceedings, Society for Industrial and Applied Mathematics, Apr. 2007, pp. 237–248, DOI: 10.1137/1.9781611972771.22 (cit. on pp. 6, 14, 44).

[SV12] K. Smets and J. Vreeken, « Slim: Directly Mining Descriptive Patterns », *in*: *Proceedings of the 2012 SIAM International Conference on Data Mining*, Society for Industrial and Applied Mathematics, Apr. 2012, pp. 236–247, DOI: 10.1137/1.9781611972825.21 (cit. on pp. 7, 15, 47, 80).

[To96] Hannu Toivonen and others, « Sampling large databases for association rules », *in*: *Vldb*, vol. 96, 1996, pp. 134–145 (cit. on pp. 6, 14, 44).

[TV12] Nikolaj Tatti and Jilles Vreeken, « The Long and the Short of It: Summarising Event Sequences with Serial Episodes », *in*: *KDD'12*, ACM, 2012, pp. 462–470, DOI: 10.1145/2339530.2339606 (cit. on pp. 7, 15, 49).

[TVK10] Lini T. Thomas, Satyanarayana R. Valluri, and Kamalakar Karlapalem, « MARGIN: Maximal Frequent Subgraph Mining », *in*: *ACM Trans. Knowl. Discov. Data* 4.*3* (Oct. 2010), 10:1–10:42, DOI: 10.1145/1839490.1839491 (cit. on pp. 40, 45).

[VLS11] Jilles Vreeken, Matthijs van Leeuwen, and Arno Siebes, « Krimp: mining itemsets that compress », *in*: *Data Mining and Knowledge Discovery* 23.*1* (July 2011), pp. 169–214, DOI: 10.1007/s10618-010-0202-x (cit. on pp. 7, 15, 47–49, 57, 62, 68, 80, 117, 121).

[Wö+05] Marc Wörlein et al., « A Quantitative Comparison of the Subgraph Miners MoFa, gSpan, FFSM, and Gaston », *in*: *Knowledge Discovery in Databases: PKDD 2005*, vol. 3721, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2005, pp. 392–403, DOI: https://doi.org/10.1007/11564126_39 (cit. on pp. 42, 43).

[YH02] Xifeng Yan and Jiawei Han, « gSpan: Graph-based substructure pattern mining », *in*: *2002 IEEE International Conference on Data Mining. Proceedings*, Springer Berlin Heidelberg, 2002, pp. 721–724, DOI: https://doi.org/10.1109/ICDM.2002.1184038 (cit. on pp. 40, 42, 65, 79).

[YH03] Xifeng Yan and Jiawei Han, « CloseGraph: Mining Closed Frequent Graph Patterns », *in*: *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, ACM, 2003, pp. 286–295, DOI: 10.1145/956750.956784 (cit. on pp. 40, 44, 45, 47).

[YZH05] Xifeng Yan, X. Jasmine Zhou, and Jiawei Han, « Mining closed relational graphs with connectivity constraints », *in*: *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, KDD '05, Association for Computing Machinery, Aug. 2005, pp. 324–333, DOI: 10.1145/1081870.1081908 (cit. on pp. 40, 45).

[Zhu+07] Feida Zhu et al., « gPrune: A Constraint Pushing Framework for Graph Pattern Mining », *in*: *Advances in Knowledge Discovery and Data Mining*, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 388–400 (cit. on pp. 40, 45, 53).

Appendix A

Publications

The following publications have been produced during the course of this PhD:

[BCF20a] Francesco Bariatti, Peggy Cellier, and Sébastien Ferré, « GraphMDL : sélection de motifs de graphes avec le principe MDL », *in*: *Revue des Nouvelles Technologies de l'Information* Extraction et Gestion des Connaissances , RNTI-E-36 (2020), pp. 37–48 (cit. on p. 54).

[BCF20b] Francesco Bariatti, Peggy Cellier, and Sébastien Ferré, « GraphMDL: Graph Pattern Selection Based on Minimum Description Length », *in*: *Advances in Intelligent Data Analysis XVIII*, ed. by Michael R. Berthold, Ad Feelders, and Georg Krempl, Lecture Notes in Computer Science, Springer International Publishing, 2020, pp. 54–66, DOI: 10.1007/978-3-030-44584-3_5 (cit. on p. 54).

[BCF20c] Francesco Bariatti, Peggy Cellier, and Sébastien Ferré, « GraphMDL Visualizer: Interactive Visualization of Graph Patterns », *in*: *GEM: Graph Embedding and Mining Workshop @ ECML-PKDD 2020*, Sept. 2020, URL: https://hal.inria.fr/hal-03142207 (cit. on p. 54).

[BCF21] Francesco Bariatti, Peggy Cellier, and Sébastien Ferré, « GraphMDL+: interleaving the generation and MDL-based selection of graph patterns », *in*: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, SAC '21, Association for Computing Machinery, Mar. 2021, pp. 355–363, DOI: 10.1145/3412841.3441917 (cit. on p. 80).

DOCTORAT BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Génération et sélection d'ensembles de motifs de graphes avec le principe MDL

Mot clés : Fouille de motifs • Fouille de graphes • Principe MDL

Résumé : De nos jours, dans de nombreux domaines, de grandes quantités de données sont disponibles sous la forme de graphes. En les analysant, un utilisateur peut en extraire de la connaissance utile. Cependant, la taille et la complexité des données rendent leur exploitation complexe pour un humain. Afin de faciliter l'analyse de ces données, des approches de *fouille de motifs* ont été développées. Elles permettent d'extraire des structures locales fréquentes, appelées *motifs*, desquels l'utilisateur peut déduire de la connaissance, au lieu d'analyser l'intégralité des données. Un problème courant en fouille de motifs est *l'explosion du nombre de motifs* extraits. Même sur de petits jeux de données, les ensembles de motifs extraits par les approches classiques sont de très grande taille et contiennent de nombreuses redondances. Dans cette thèse, nous proposons trois approches qui utilisent le *principe Minimum Description Length* (MDL) afin de générer et de sélectionner *des petits ensembles* de motifs *descriptifs* de type graphe à partir de données de type graphe. Pour cela, nous instancions le principe MDL dans un contexte de fouille de motifs de graphe et nous proposons des mesures MDL pour évaluer des ensembles de motifs. Nous introduisons également la notion de *ports*, permettant de décrire les données comme une composition d'occurrences de motifs sans perte d'information. Nous évaluons toutes nos contributions sur des jeux de données de graphes provenant de différents domaines, y compris du web sémantique.

Title: Mining Tractable Sets of Graph Patterns with the Minimum Description Length Principle

Keywords: Pattern Mining • Graph Mining • Minimum Description Length Principle

Abstract: Nowadays, large quantities of graph data can be found in many fields, encoding information about their respective domains. Such data can reveal useful knowledge to the user that analyzes it. However, the size and complexity of real-life datasets hinders their usage by human analysts. To help the users, *pattern mining* approaches extract frequent local structures, called *patterns*, from the data, so that they can focus on inferring knowledge from them, instead of analyzing the whole data at once. A well-known problem in pattern mining is the so-called problem of *pattern explosion*. Even on small datasets, the set of patterns that are extracted by classic pattern mining approaches can be very large in size, and contain many redundancies. In this thesis we propose three approaches that use the *Minimum Description Length principle* in order to generate and select *small, human-sized* sets of *descriptive* graph patterns from graph data. For that, we instantiate the MDL principle in a graph pattern mining context and we propose MDL measures to evaluate sets of graph patterns. We also introduce the notion of *ports*, allowing to describe the data as a composition of pattern occurrences with no loss of information. We evaluate all our contributions on real-life graph datasets from different domains, including the semantic web.