GraphMDL+: Interleaving the Generation and MDL-based Selection of Graph Patterns

Francesco Bariatti
Univ Rennes, CNRS, IRISA
francesco.bariatti@irisa.fr

Peggy Cellier
Univ Rennes, INSA, CNRS, IRISA
peggy.cellier@irisa.fr

Sébastien Ferré
Univ Rennes, CNRS, IRISA
sebastien.ferre@irisa.fr

ABSTRACT

Graph pattern mining algorithms ease graph data analysis by extracting recurring structures. However, classic pattern mining approaches tend to extract too many patterns for human analysis. Recently, the GraphMDL algorithm has been proposed, which reduces the generated pattern set by using the *Minimum Description Length* (MDL) principle to select a small descriptive subset of patterns. The main drawback of this approach is that it needs to first generate all possible patterns and then sieve through their complete set. In this paper we propose GraphMDL+, an approach based on the same description length definitions as GraphMDL but which tightly interleaves pattern generation and pattern selection (instead of generating all frequent patterns beforehand), and outputs a descriptive set of patterns at any time. Experiments show that our approach takes less time to attain equivalent results to GraphMDL and can attain results that GraphMDL could not attain in feasible time. Our approach also allows for more freedom in the pattern and data shapes, since it is not tied to an external approach.

KEYWORDS

Pattern Mining, Graph Mining, Minimum Description Length

ACM Reference Format:
Francesco Bariatti, Peggy Cellier, and Sébastien Ferré. 2021. GraphMDL+: Interleaving the Generation and MDL-based Selection of Graph Patterns. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21), March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3412841.3441917

1 INTRODUCTION

Labeled graphs offer a powerful data representation in many fields. For example, in chemistry, molecules can be represented as graphs where vertices are atoms, and edges are bonds; in linguistics, sentences can be represented as graphs where vertices are word tokens, and edges are dependency relationships; in the semantic web, knowledge is represented as graphs where vertices are entities and edges are semantic relationships. Extracting knowledge from graph datasets is desirable but difficult due to the complexity of analyzing graphs, and to the large size of datasets (either large single graphs, or large collections of small graphs). To support knowledge extraction from graphs, there has been a number of proposals for *graph pattern mining*, i.e. for extracting frequent structures in graph datasets [10, 14].

A well-known issue of pattern mining is the large number of patterns that are extracted. Several kinds of approaches have been proposed to tackle this problem (e.g., constraint based approaches [16] or condensed representation [15]). Among them, the methods based on the Minimum Description Length (MDL) principle demonstrated that it is possible to drastically reduce the number of patterns by selecting a small set of descriptive patterns among all the generated patterns [4, 5, 8, 12, 13]. The MDL principle [6] comes from information theory, and states that the *model* that describes the data the best is the one that compresses the data the best, i.e. which yields the shortest description length. Few MDL-based approaches have been proposed for graphs. SUBDUE [3] iteratively compresses a graph by replacing each occurrence of a pattern by a single vertex, which entails a loss of information. VoG [9] summarizes graphs as a composition of predefined families of patterns (e.g., paths, stars) which restricts the type of patterns that can be extracted. Another limitation is that VoG works on unlabeled graphs only. GraphMDL [1] is a recent approach that works on labeled graphs, and leverages the MDL principle to select graph patterns. Contrary to SUBDUE, it ensures that there is no loss of information thanks to the introduction of the notion of *ports* associated to graph patterns. Ports represent how adjacent occurrences of patterns are connected. GraphMDL is a two-phase method. First, a large collection of patterns is generated by a graph miner like gSpan [14] or Gaston [10]. Second, it iterates through all generated patterns in order to select a small subset of descriptive patterns. This two-phase method has several drawbacks. The combinatorial number of patterns in graphs entails a high computational cost for both the generation phase and the selection phase. In addition, to mitigate the computational cost, during the generation phase a minimum support has sometimes to be set at a higher value than desired, so that possibly useful patterns are overlooked.

In this paper we propose the GraphMDL+ approach to tackle those limitations. In MDL-based pattern mining, GraphMDL+ is to GraphMDL on graph patterns as Slim [11] is to Krimp [13] on itemsets. In other words, GraphMDL+ uses the same description length definitions as GraphMDL but tightly interleaves pattern generation and pattern selection (instead of generating all frequent patterns beforehand). As a positive consequence, it becomes an anytime approach

SAC '21, March 22–26, 2021, Virtual Event, Republic of Korea

that can output a descriptive set of patterns whenever the user desires so. Another benefit is to free from the limits of graph miners about the nature of graphs (e.g., directed vs undirected) and patterns (e.g., exactly one label on each vertex), and hence recover patterns that were excluded. For the sake of tractability, we propose a heuristic to rank the candidate patterns at each generation step. In addition, we present and assess a refinement of GraphMDL+ that exploits *graph automorphisms*, i.e. symmetries that exist in graph patterns, in order to reduce description length further. We conducted experiments on four different datasets to validate our approach, in particular we show that it takes less time and can attain results that GraphMDL could not attain in reasonable time.

2 PRELIMINARIES

In this section we first recall the MDL principle (Section 2.1). We then give graph definitions that are useful for this paper (Section 2.2). Finally, we present the concepts from GraphMDL [1] that are used in the following (Section 2.3).

2.1 The MDL Principle

The *Minimum Description Length* (MDL) principle [6] is a technique from information theory that allows to select the model, from a family of models, that best describes some data. In practice, MDL states that the model M that best describes some data D 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 $L(D|M)$ the description length of the data encoded with the model. Since description lengths are often expressed in bits of information, MDL gives a numerical measure of the quality of a model versus another (w.r.t. given data). The MDL principle has been successfully employed to select descriptive sets of patterns from transactional databases [13], sequence databases [5, 12], relational databases [8], geometric data [4], and graphs [1].

2.2 Graphs and Embeddings

In this part we recall graph theory definitions for labeled graphs, and for the embeddings of a labeled graph into another.

Definition 2.1. A *labeled graph* $G = (V, E, l_V, l_E)$ over two label sets $\mathcal{L}_V$ and $\mathcal{L}_E$ is a data structure composed of a set of *vertices* V, a set of *edges* $E \subseteq V \times V$, and two *labeling functions* $l_V \in V \to 2^{\mathcal{L}_V}$ and $l_E \in E \to \mathcal{L}_E$ that associate a set of labels to vertices, and one label to edges. G is said *undirected* if E is symmetric, and *simple* if E is irreflexive.

This definition of a graph can define both directed and undirected graphs. Note that this graph definition allows for vertices to have several labels, as well as no labels at all (our approach takes advantage of this, see Section 4). Figure 1 shows a graph with 8 vertices with labels from $\{W, X, Y, Z\}$ and 7 undirected edges with labels from $\{a, b\}$ connecting the vertices.

Definition 2.2. Let G^P and G^D be graphs. An *embedding* (or *occurrence*) of G^P (the pattern) in G^D (the data) is an injective function $\varepsilon \in V^P \to V^D$ such that: (1) $l^P_V(v) \subseteq l^D_V(\varepsilon(v))$ for all $v \in V^P$; (2) $(\varepsilon(u), \varepsilon(v)) \in E^D$ for all $(u, v) \in E^P$; and (3) $l^P_E(e) = l^D_E(\varepsilon(e))$ for all $e \in E^P$.

Fig. 2 shows the three embeddings (ε_1 in green, ε_2 in red, and ε_3 in blue) of a simple pattern (with 2 vertices and one edge) in the graph of Figure 1.

2.3 GraphMDL

GraphMDL takes two inputs: the graph dataset and a set of candidate patterns, and then outputs a subset of the candidate pattern set which yields the smallest possible description length. GraphMDL+ borrows five notions from GraphMDL: graph model, rewritten graph, ports, code table, and description length definitions. We recall those five notions in this section.

Graph model. The model, in the MDL sense, used to compress graph datasets in GraphMDL is a list of graph patterns, selected from the candidate patterns received as input. The simplest patterns handled by GraphMDL are called *singletons*. A *vertex singleton pattern* has one vertex with exactly one label, an *edge singleton pattern* has two vertices without label, connected by an edge (with a label). Fig. 3 shows an example of each type of singleton pattern. Singleton patterns can be seen as "building blocks" for other patterns: e.g. the pattern of Fig. 2 is the union of singleton vertex pattern Z and singleton edge pattern b.

Rewritten graph and ports. The intuition behind GraphMDL is that a data graph can be encoded and compressed as a composition of pattern embeddings, where patterns are taken from the graph model. The way the graph model is used to compress the data graph is made explicit by a structure called the *rewritten graph*. We now walk through the creation of such a structure, which is shown in Fig. 4. Fig. 4*a* represents the graph model as a list of patterns. Those patterns are considered top to bottom[1]: for each one, its embeddings in the data (in this case the graph of Fig. 1) are computed. Not all embeddings are retained: for an embedding to be retained its edges must not overlap with edges of previously-chosen embeddings (be they of the same pattern or a different one). Fig. 4*b* shows the retained embeddings. Each of those embeddings generates what is called an *embedding vertex* in the rewritten graph. Each embedding vertex indicates that in the data there is an occurrence of the pattern corresponding to its label. Note that the embeddings overlap on some vertices: these are what GraphMDL calls *port vertices*. Port vertices are represented in the rewritten graph as well. Lastly, an embedding vertex is connected to a port vertex in the rewritten graph if that specific embedding uses that specific port: the label of the edge connecting them indicates which of the pattern's ports is mapped to that vertex. Fig. 4*c* shows the rewritten graph which results from the embeddings of Fig. 4*b*. GraphMDL associates a description length to a

[1] The order of patterns is determined by a heuristic, as detailed in [1]

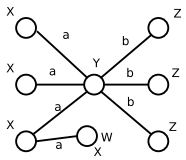


Figure 1: Graph used as example throughout this paper.

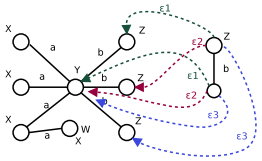


Figure 2: Embeddings of a pattern in the graph of Fig. 1.

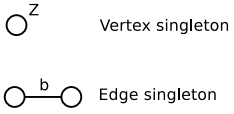


Figure 3: Two singleton patterns.

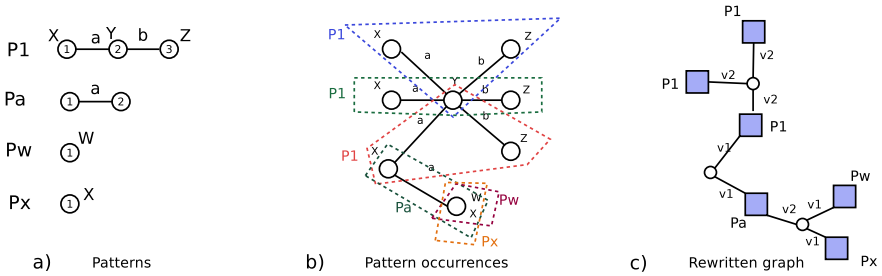


Figure 4: How a set of patterns is used to encode the graph of Fig.1. *a*) **Some patterns.** *b*) **Retained occurrences of the patterns in the data.** *c*) **Resulting rewritten graph: blue squares are pattern embeddings, white circles are ports.**

rewritten graph. This numerical value corresponds to the $L(D|M)$ term of the MDL formula and is used to evaluate the "complexity" of the rewritten graph.

Code table. The *code table* is a representation of the graph model that not only contains the representation of the list of graph patterns but also various coding information about those patterns [13]. For each pattern, the code table stores its graph structure, a code based on its *usage*, i.e. the number of times the pattern appears in the rewritten graph, and information about the pattern's ports. This information is needed to compute the description length $L(D|M)$ of the rewritten graph. GraphMDL also computes the description length of the code table itself: it is the $L(M)$ term of the MDL formula. Including this term in the formula ensures that overly-complicated code tables (e.g. many patterns with many ports) will be penalized.

Description length definitions. The definition of $L(M, D)$ is decomposed into the definitions of $L(M)$, the description length of a code table, and of $L(D|M)$, the description length of a rewritten graph. Each defintion boils down to a sum of code lengths. Those code lengths are determined according to different distributions depending on the symbols to encode. Vertex and edge labels are encoded according to their distribution in the data graph, patterns and ports are encoded according to their usage in the rewritten graph, and standard encodings are used for various integers (e.g., number of vertices). The detailed definitions of those description lengths are available in [1].

3 GRAPHMDL+: INTERLEAVING GENERATION AND SELECTION

In this section we present our approach, GraphMDL+. The key contribution of GraphMDL+ is to define a way to interleave pattern generation and pattern selection. Instead of using an external independent pattern generation algorithm, patterns are generated during the selection process. The idea is to generate *candidate patterns* that may be a good addition to the code table, and yield a better description length.

In this section, first we present the approach informally (Section 3.1). We then introduce the definition of candidates (Section 3.2) and how their compression capability is estimated in order to rank them (Section 3.3). Finally, the complete algorithm of GraphMDL+ is given (Section 3.4) with its extension that handles *graph automorphisms* (Section 3.5).

3.1 A Sketch of GraphMDL+

We base the candidate generation around GraphMDL's notion of rewritten graph: given a code table, the rewritten graph represents how the algorithm uses the patterns in the code table to encode the data. Our intuition is to take the rewritten graph and observe which patterns appear frequently together: two patterns appear together if some of their embeddings share at least one port vertex of the rewritten graph. If two patterns appear frequently together, we merge them to create a larger pattern covering their common occurrences. The shared ports may then become internal vertices of the larger pattern, and hence not ports anymore.

Every time 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 description length of the new

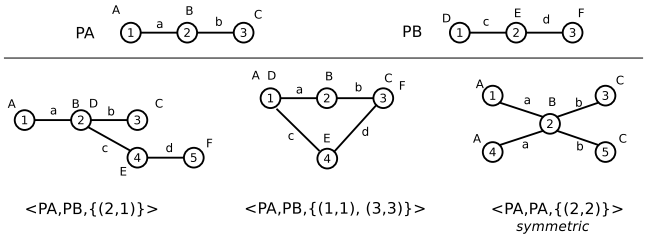


Figure 5: Two patterns, P_A and P_B, and the merged patterns obtained following three different candidates.

encoding is better, the pattern is kept. Thanks to this approach, we can start with small (very general) patterns in the code table, and make our way up to generate larger (i.e. more specific) patterns. Larger patterns are interesting because they allow to describe more of the data in a single occurrence. At the same time, they induce a higher description length of the code table. The MDL principle acts as a gatekeeper, guaranteeing that among all generated patterns, only the ones that are interesting enough will make their way into the code table.

3.2 Candidate Generation

In order to generate new patterns, GraphMDL+ analyzes the current rewritten graph. Let P_A be a pattern which has an embedding in the rewritten graph that maps one of its vertices, v_A, to a certain data vertex v_D. Let P_B be another pattern which has an embedding in the rewritten graph that maps one of its vertices, v_B, to the same data vertex v_D. The intuition behind GraphMDL+ is to say that it may be interesting to merge P_A and P_B, merging their vertices v_A and v_B. This would allow to describe *with a single pattern* all those parts of the data where there is an embedding of P_A and an embedding of P_B mapping their respective vertices v_A and v_B on the same data vertex. The structure describing this possible merge of two patterns is called a *candidate*:

Definition 3.1. A *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.

Definition 3.2. 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 shows what it means to merge two patterns P_A and P_B following a candidate, by showing the resulting merged pattern for three different candidates. The last one is symmetric.

Note that any pattern can be decomposed into singleton patterns (describing a single edge or a single vertex label) and therefore can be generated by successive merges through candidates.

3.3 Candidate Ranking

On a given rewritten graph, many candidates can be generated. A choice has to be made about which one to consider first for insertion into the list of selected patterns (a.k.a. the code table). A naive approach may be to try them all separately and rank them based on how much the description length lowers when they are inserted in the code table, in order to choose the one that improves the description length the most. However such an approach would be computationally expensive because it implies recomputing the MDL encoding of the data 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 4.2.

Our intuition is to select the candidate that is expected to be used the most in the MDL encoding of the data. Remember that the more frequently something appears in the data, the more interesting it tends to be according to MDL. We estimate the usage of a candidate $C =< P_A, P_B, \Pi >$ by looking at the number of embeddings of P_A and P_B that participate to the candidate in the rewritten graph. For example, if there are 3 embeddings of each pattern, we can reasonably expect the candidate to be used at least 3 times[2]. 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. In that case, we take the smaller of the two counts, because embeddings are not allowed to overlap. We call this measure the *usage estimate* of the candidate. It is similar to the *minimum image based support* proposed in [2]. To break equalities, we compute for each candidate its number of *exclusive ports*: the rewritten graph port vertices around which only the two patterns of the candidate are present. Those ports will disappear since they are completely covered by the candidate, and 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 pattern resulting from the candidate: we consider first the candidate with the smallest description length, because from an MDL point of view it is the "simplest".

Special cases of usage estimation. There are some special cases in which the general formula for estimating the usage of a candidate 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. This is because GraphMDL+ only forbids edge overlaps, and does not forbid multiple patterns to describe the same vertex labels. In that case, the usage estimate of the candidate is the number of embeddings of the other pattern. If both patterns do not have edges, the

[2]The exact number depends on the position that the generated pattern will have in the code table. The higher the pattern is in the code table, the more its embeddings have a chance of being *retained* (See Section 2.3). In order to know the exact number of retained embeddings, the full rewritten graph would have to be recomputed.

usage estimate of the candidate is the number of embeddings —that participate to the candidate— of any of the two (both patterns will have the same number).

If a candidate is symmetric (see Def. 3.2), it means that embeddings of P_A can count as embeddings of P_B and conversely. In that case, we aggregate the embeddings of both patterns and take as usage estimate half their number (rounded down). We need to take half the number of embeddings, because the candidate merges the patterns two by two.

We summarise the computation of the usage estimate as follows:

- if the candidate is symmetric, combine embeddings of the two patterns. Usage estimate is half the size of the resulting set;
- otherwise, if one of the patterns does not have edges, usage estimate is the number of embeddings of the other pattern;
- otherwise, usage estimate is the number of embedding of the pattern that has less embeddings.

Fig. 6 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. There are three embeddings of each pattern that take part in this candidate (they are around the topmost port vertex). Therefore, the usage estimate of the candidate is 3. The candidate on the second line is symmetric: it describes an embedding of P_1 connected to another embedding of P_1, through the vertex 1 of both. Two embeddings of P_1 take part in this candidate (embeddings c and d). Each of them could be both an embedding of the first pattern of the candidate and the second. Therefore we aggregate the two embeddings and take half their number as usage estimate. Also, these embeddings are the only embeddings around the leftmost port vertex, so this candidate has an exclusive port. Finally, the fourth line describes the last case: 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.

3.4 The GraphMDL+ Algorithm

The GraphMDL+ algorithm is presented in Alg. 1. The code table is initialised with the code table containing all singleton patterns (line 1). At each iteration, from the current code table, GraphMDL+ generates the associated rewritten graph (line 5). Then (lines 6-7), the rewritten graph is parsed, and for each pair of embeddings that share at least a port, a candidate is generated. The most interesting candidate w.r.t. usage estimate (see Section 3.3) is added to the code table (line 9). If this new code table lowers the MDL description length, a new iteration is done with the new code table (line 11). Otherwise, the candidate is removed from the code table and the next 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 exits returning the best code table found (line 14).

Algorithm 1 The GraphMDL+ algorithm

```
Require: A data graph D
1:  CT ← singleton-only code table
2:  return GraphMDL+Iteration(D, CT)
3:
4:  function GraphMDL+Iteration(D, CT)
5:      G^r ← rewritten graph from D and CT
6:      C ← candidates generated from G^r        ▷ See Sec. 3.2
7:      Sort C by usage estimate                 ▷ See Sec. 3.3
8:      for all C ∈ C do
9:          CT' ← CT ∪ C
10:         if L(CT', D) < L(CT, D) then
11:             return GraphMDL+Iteration(D, CT')
12:         end if
13:     end for
14:     return CT
15: end function
```

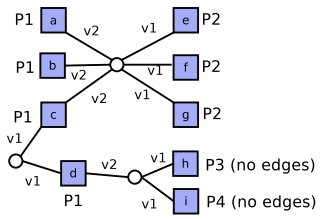
An interesting aspect of the GraphMDL+ algorithm is that it can be interrupted whenever the user desires so, since it can just return the best code table found so far. This makes GraphMDL+ an *anytime* algorithm, and effectively a *parameter-less* one as well, since the only input it needs is the data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

3.5 Handling Automorphisms

A peculiar characteristic of graphs w.r.t. other types of data (e.g. itemsets, sequences) is that different graphs can be related by *isomorphisms* and *automorphisms* [7].

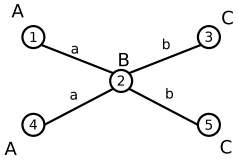
Definition 3.3. Let $G = (V, E, l_V, l_E)$ and $G' = (V', E', l'_V, l'_E)$ be two graphs. G and G' are said to be *isomorphic*, if there exists a bijection $\gamma \in V \mapsto V'$, called an *isomorphism*, such that: (1) $(u, v) \in E \iff (\gamma(u), \gamma(v)) \in E'$; (2) $l_V(v) = l'_V(\gamma(v))$ for all $v \in V$; and (3) $l_E(e) = l'_E(\gamma(e))$ for all $e \in E$. An isomorphism between a graph and itself is called an *automorphism*.

An automorphism can be seen as an alternative numbering of the vertices of a graph, and exhibits symmetries in the graph structure. For example, in the graph of Fig. 7 the vertices 1 and 4 are completely equivalent: exchanging them does not change the structure of the graph. The same happens for vertices 3 and 5. All the possible configurations issued by exchanging those pairs of vertices (listed in the table) are called the *automorphisms* of the graph. We integrate in GraphMDL+ the *bliss* algorithm [7] for the computation of all automorphisms of a labeled graph. It is important for a graph mining algorithm to handle automorphisms as otherwise it may consider several graphs or vertices as different whereas they are —in fact— equivalent. For example, let P_A be a pattern that has two vertices, v_1 and v_2, that are equivalent under automorphism. If GraphMDL+ did not detect this automorphism, it would consider



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
$< P_1, P_1, \{(v_1, v_1)\} >$	$\{c, d\}$	$\{c, d\}$	$\lfloor 2/2 \rfloor = 1$	1
$< P_1, P_1, \{(v_2, v_2)\} >$	$\{a, b, c\}$	$\{a, b, c\}$	$\lfloor 3/2 \rfloor = 1$	0
$< P_1, P_3, \{(v_2, v_1)\} >$	$\{d\}$	$\{h\}$	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 6: A rewritten graph and the candidates that can be generated from it. Embedding vertices (blue squares) have been assigned letters for illustration purposes.



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

Figure 7: A simple graph (left) and its automorphisms, represented as alternative numberings of the vertices (right).

Table 1: Characteristics of the datasets in experiments.

Dataset	Graph count	Vertices	Edges	Vertex labels	Edge labels
AIDS-CA	423	17k	18k	21	3
AIDS-CM	1082	34k	37k	26	3
Mutag	125	2k	3k	7	4
PTC-FM	143	2k	2k	18	4
PTC-FR	121	2k	2k	19	4
PTC-MM	129	2k	2k	20	4
PTC-MR	152	2k	2k	18	4
UD-PUD-En	1000	21k	20k	17	46

that candidate $< P_A, P_B, \{(v_1, v_b)\} >$ is different from candidate $< P_A, P_B, \{(v_2, v_b)\} >$, whereas they are —in fact— the same. This would lead to underestimating the usage of that candidate.

Another instance where handling automorphisms is important is in the description length computation. A pattern with many ports has a higher description length than with few ports. If two vertices are equivalent under automorphism, GraphMDL+ always uses the same vertex as port, concentrating the usage on a single port, thus reducing the description length.

4 EXPERIMENTAL EVALUATION

In this section we present a quantitative evaluation of GraphMDL+ and discuss the advantages that controlling the pattern generations gives us w.r.t. GraphMDL. In this paper we focus on the comparison between GraphMDL and GraphMDL+, in order to show the impact that the interleaving of the candidate generation and the selection has on the runtime, description length, and selected patterns. A comparison of GraphMDL w.r.t. other graph mining approaches has already been done in [1].

Settings. We have developed a prototype of GraphMDL+ in Java 1.8, available as a git repository[3]. We conducted the experiments on four different datasets (some of them having several sub-datasets). The AIDS[4] (AIDS-CA and AIDS-CM), Mutag[5], and PTC[6] (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 labels and many cycles. The UD-PUD-En dataset is from the Universal Dependencies project[7]. Each graph of the dataset represents a sentence: vertices are words, vertex labels are POS tags, and edge labels are dependency relationships. This dataset has many labels and no cycles. For datasets that have both a positive and a negative class, only the positive class was used for the evaluation. The characteristics of the used datasets are presented in Table 1.

GraphMDL needs an external approach to generate the candidate patterns that it filters. We used the algorithm gSpan[8] for that task. 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 used the lowest value possible which allowed to terminate the experiments in a reasonable amount of time.

All experiments were conducted with a RAM limit of 10Gb on a laptop with a Intel Core i7-7600U@2.80Ghz cpu.

4.1 Comparison with GraphMDL

In order to compare the performances of GraphMDL and GraphMDL+ we ran some 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. The gSpan runtime is not included in the GraphMDL runtime. One first observation that we made while running 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. On the

[3] https://gitlab.inria.fr/fbariatt/graphmdl/
[4] https://wiki.nci.nih.gov/display/NCIDTPdata/AIDS+Antiviral+Screen+Data
[5] http://networkrepository.com/Mutag.php
[6] http://networkrepository.com/PTC-FM.php (also replace FM with FR, MM or MR)
[7] https://universaldependencies.org/
[8] https://sites.cs.ucsb.edu/~xyan/software/gSpan.htm

Table 2: Comparison of GraphMDL and GraphMDL+ results. $L\%$ is the ratio between the description length of the found code table and the singleton-only code table. $|CT|$ is the number of patterns in the code table found by the algorithm.

	GraphMDL for best $L\%$			GraphMDL+ for $L\% \leq L_1$		GraphMDL+ for best $L\%$		
Dataset	$L_1 = best\ L\%$	time	$\|CT\|$	time	$\|CT\|$	best $L\%$	time	$\|CT\|$
AIDS-CA	19.15%	1h51m	148	36s	110	12.02%	1h17m	305
AIDS-CM	20.71%	2h50m	212	1m22s	170	14.68%	4h00m	644
Mutag	15.42%	2h38m	29	3s	33	10.73%	1m28s	48
PTC-FM	22.87%	1h42m	77	12s	77	22.01%	1m08s	87
PTC-FR	23.35%	27m54s	70	8s	77	22.62%	1m23s	85
PTC-MM	23.65%	2h09m	75	4s	76	22.12%	1m16s	84
PTC-MR	23.09%	18m43s	78	7s	84	21.43%	1m01s	87
UD-PUD-En (undir.)	26.84%	1h41m	647	2m59s	459	25.29%	2h32m	801

opposite, since GraphMDL+ is an anytime algorithm, it can be easily stopped at a chosen time limit. Table 2 presents the results of these experiments. For each dataset we present the description length of the best code table found by GraphMDL (as a percentage of the length of a singleton-only code table), the time it took to find it, and the number of patterns it contains. We then present the time it takes GraphMDL+ to find a code table with an equivalent description length and the number of patterns in that code table. Finally we present the description length of the best code table that can be found by GraphMDL+ within the given time limit, the time it took to find it, and the number of patterns it contains.

We observe that GraphMDL+ manages in all cases to find a code table resulting in the same description length and a similar amount of patterns (between -29% and +14%) than GraphMDL. It does so in significantly less time. For instance, for 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 needs to process all candidate patterns, and most of them will not be useful for decreasing the description length (since they are generated by an external approach with no knowledge of the MDL principle), while the former generates less candidates of higher quality for an MDL approach. A consequence of this speed increase is that GraphMDL+ can find code tables that have a description length even lower than the best ones found by GraphMDL. For instance, for the Mutag dataset, the best description length computed by GraphMDL is 15.42% whereas the best description length computed by GraphMDL+ is 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. Note that for most experiments the GraphMDL+ algorithm was not actually stopped at the 4 hours time limit, but completed before. This is because it reached a rewritten graph where no candidate improved the description length (see Section 3.4). The number of patterns selected for the best GraphMDL+ code table is higher, but remains significantly smaller than gSpan's (at least 10k on all datasets).

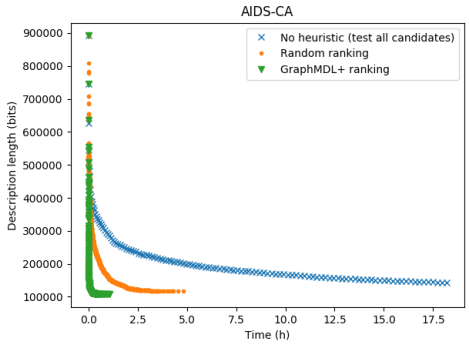


Figure 8: Evolution of the description length over time for different candidate rankings on AIDS-CA.

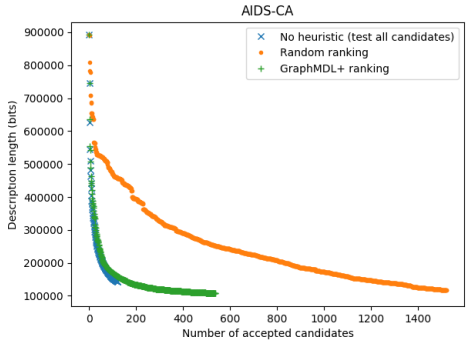


Figure 9: Evolution of the description length for each addition to the code table for different rankings on AIDS-CA.

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.

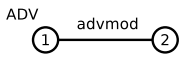


Figure 10: A GraphMDL+ pattern on UD-PUD-En.

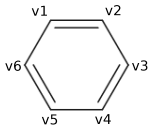


Figure 11: A GraphMDL+ pattern on AIDS-CA.

4.2 Candidate Ranking Heuristic Evaluation

In Section 3.3 we give a heuristic (the *usage estimate*) that GraphMDL+ uses to choose which candidate to test first for inclusion in the code table (line 7 in Alg. 1). In order to evaluate this heuristic, we ran three versions of GraphMDL+ which used different candidate ranking functions. The "test all candidates" version sorts all possible candidates by the actual description length gain, which is costly to compute. The "random ranking" version randomly sorts the candidates. The "GraphMDL+ ranking" version sorts the candidates using the usage estimate heuristic. All versions after ranking the candidates then proceed through the list and select the first candidate that when added to the code table produces a description length which is better than the current best one (See Section 3.4). This experiment does not question the greedy approach of GraphMDL+, but evaluates the heuristic that guides this approach.

Fig. 8 and Fig. 9 show the results of this experiment on the AIDS-CA dataset. Results on the other datasets follow the same behaviour. We stopped the "test all" version after 18 hours, the others terminated. We observe in Fig. 8 that testing all candidates takes a significant amount of time and that a heuristic is needed in order for the approach to complete in a reasonable amount of time. The ranking that we propose is the fastest approach of the three, faster even than the random ranking. We assume that the cost of computing the usage estimate is negligible (it can be computed during the candidate generation phase), and the proposed candidates are efficient in reducing the description length. We observe in Fig. 9 that the "test all" version always selects the candidate that improves the description length the most, as expected. 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, which has the consequence of needing to test more candidates in order to converge to similar description lengths as our ranking, since candidates improve less.

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 Section 3.3 is both fast and efficient in finding good candidates.

Table 3: Port usage of the pattern of Fig. 11 with automorphisms detection, and theoretical usage without.

	With automorphisms		Without automorphisms	
Vertex	Usage	Code length	Usage	Code length
v_1	33	0.63	8	2.67
v_2	0	0	8	2.67
v_3	2	4.67	8	2.67
v_4	13	1.97	9	2.50
v_5	1	5.67	9	2.50
v_6	2	4.67	9	2.50
Total DL	70.75		161.58	

4.3 Advantages of Controlling Generation

Since GraphMDL+ directly controls the generation of patterns, it has more freedom than GraphMDL over the shape of patterns.

Patterns without labels. GraphMDL allows for vertices to have any number of labels, including none, however gSpan requires each vertex to have exactly one label, which means that when GraphMDL is used to select between patterns generated by gSpan, it can only select patterns which have a label on each vertex. GraphMDL+ does not have this limitation. Fig. 10 shows one of the patterns extracted by GraphMDL+ on the UD-PUD-En dataset: this pattern can be interpreted as "an adverb that is the adverbial modifier of *something*". Since its second vertex does not have a label, it can be anything. And in fact, this pattern is used by GraphMDL+ for describing adverbial modifiers of many different entities: nouns, auxiliaries, determiners, pronouns, etc. GraphMDL (coupled with gSpan) could not have such a pattern and would instead have more specific patterns such as "an adverb that is the adverbial modifier of a noun", "... of an auxiliary", "... of a pronoun", etc. Having the control over the pattern generation step gives to GraphMDL+ a greater generalisation power.

Directed graphs. GraphMDL has the potential to handle directed graph patterns. However graph pattern mining algorithms that can handle directed graphs are rare in the literature, which makes it difficult to apply GraphMDL on directed sources of data. While it is possible to interpret a directed graph as undirected, doing so removes part of the information from the generated patterns and can therefore hinder the interpretation for the users of GraphMDL. GraphMDL+ does not have this limitation, since it directly controls the pattern generation.

Handling automorphisms. Fig. 11 shows one of the patterns extracted by GraphMDL+ on the AIDS-CA dataset. This pattern is a benzene cycle, and has many automorphisms that reflect the rotation and mirror symmetries: every vertex is equivalent to every other. Table 3 shows what is the usage of each port and its associated code length when automorphisms are handled —as in GraphMDL+— (on the left) and what those port usages and code lengths would be if the approach did not handle automorphisms (on the right). Every time

Table 4: Difference in description length given by the best code table found by GraphMDL+ with and without automorphisms (4 hours maximum runtime).

Dataset	Without automorphisms	With automorphisms	Ratio without / standard
AIDS-CA	111191 bits	107207 bits	1.037
AIDS-CM	287482 bits	287101 bits	1.001
Mutag	12756 bits	12388 bits	1.030
PTC-FM	17905 bits	17900 bits	1.000
PTC-FR	16988 bits	16704 bits	1.017
PTC-MM	16191 bits	16215 bits	0.999
PTC-MR	19582 bits	19509 bits	1.004
UD-PUD-En	298812 bits	299073 bits	0.999

this pattern has an embedding which requires a port, GraphMDL+ identifies that all the vertices are equivalent, and therefore always chooses the same vertex (or vertices when the pattern has multiple ports). This leads to the usage being concentrated on the same ports, making the code associated to these ports shorter, which leads to a smaller total description length (sum of code lengths of each port times their usages). On the contrary, an algorithm that does not detect automorphisms would choose a random vertex every time, since every vertex is as likely to be chosen. This leads to a uniform spread of the usage among all vertices, which leads to them having longer codes, which leads to a higher description length. In turn, this penalizes the pattern by overestimating the description length of its ports.

In order to analyze the impact of handling automorphisms on the performances of GraphMDL+, we ran some experiments with and without it. Table 4 presents the description length attained by the best code table found by the algorithm on each dataset for a maximum runtime of 4h. We observe that the impact of handling automorphisms is quantitatively small but positive. Indeed, the resulting description length without it is increased by 4% and 3% in AIDS-CA and Mutag respectively, and not significantly changed for the other datasets. From a qualitative point of view, handling automorphisms gives a guarantee that GraphMDL+ will not consider as different patterns or vertices that are actually equal.

5 CONCLUSION

In this paper we propose GraphMDL+, an MDL-based graph pattern mining approach that generates and selects a descriptive set of graph patterns from labeled graphs. GraphMDL+ tightly interleaves pattern generation and selection to quickly generate descriptive patterns in an anytime manner. Our experiments show that GraphMDL+ is faster than its main competitor, GraphMDL, and can extract patterns that attain better description lengths. Additionally, the handling of automorphisms allows to precisely detect whether two patterns or vertices are equivalent. Thanks to the control that GraphMDL+ gives to the pattern generation, we plan to extend this approach to other kinds of graph datasets, e.g. knowledge graphs, which are directed multi-graphs.

REFERENCES

[1] F. Bariatti, P. Cellier, and S. Ferré. 2020. GraphMDL: Graph Pattern Selection Based on Minimum Description Length. In *Advances in Intelligent Data Analysis XVIII (LNCS)*. Springer International Publishing, 54–66.

[2] Björn Bringmann and Siegfried Nijssen. 2008. What Is Frequent in a Single Graph?. In *Pacific-Asia Conf. Knowledge Discovery and Data Mining (PAKDD)*, Vol. LNCS 5012. 858–863.

[3] D. J. Cook and L. B. Holder. 1993. Substructure Discovery Using Minimum Description Length and Background Knowledge. *Journal of Artificial Intelligence Research* 1 (1993), 231–255.

[4] Micky Faas and Matthijs van Leeuwen. 2020. Vouw: Geometric Pattern Mining Using the MDL Principle. In *Advances in Intelligent Data Analysis XVIII (LNCS)*. Springer International Publishing, 158–170.

[5] E. Galbrun, P. Cellier, N. Tatti, A. Termier, and B. Crémilleux. 2018. Mining Periodic Patterns with a MDL Criterion. In *Eu. Conf. Machine Learning and Principles and Practices of Knowledge Discovery in Databases (ECML-PKDD)*.

[6] P Grünwald. 2000. Model Selection Based on Minimum Description Length. *Journal of Mathematical Psychology* 44, 1 (2000), 133–152.

[7] T. Junttila and P. Kaski. 2007. Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs. In *Work. Algorithm Engineering and Experiments (ALENEX)*. 135–149.

[8] A. Koopman and A. Siebes. 2009. Characteristic Relational Patterns. In *ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining*. 437–446.

[9] D. Koutra, U. Kang, J. Vreeken, and C. Faloutsos. 2015. Summarizing and understanding large graphs. *Statistical Analysis and Data Mining: The ASA Data Science Journal* 8, 3 (2015), 183–202.

[10] S. Nijssen and J. N. Kok. 2005. The Gaston Tool for Frequent Subgraph Mining. *Electron. Notes Theor. Comput. Sci.* 127, 1 (2005), 77–87.

[11] K. Smets and J. Vreeken. 2012. Slim: Directly Mining Descriptive Patterns. In *SIAM Int. Conf. Data Mining*. 236–247.

[12] N. Tatti and J. Vreeken. 2012. The Long and the Short of It: Summarising Event Sequences with Serial Episodes. In *Int. Conf. Knowledge Discovery and Data Mining (KDD)*. ACM, 462–470.

[13] J. Vreeken, M. van Leeuwen, and A. Siebes. 2011. Krimp: mining itemsets that compress. *Data Min. Knowl. Discov.* 23, 1 (2011), 169–214.

[14] Xifeng Yan and Jiawei Han. 2002. gSpan: Graph-Based Substructure Pattern Mining. In *IEEE Int. Conf. on Data Mining (ICDM)*. 721–724.

[15] X. Yan and J. Han. 2003. CloseGraph: Mining Closed Frequent Graph Patterns. In *ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining*. ACM, 286–295.

[16] F. Zhu, X. Yan, J. Han, and P. S. Yu. 2007. gPrune: A Constraint Pushing Framework for Graph Pattern Mining. In *PAKDD*. 388–400.