Vouw: Geometric Pattern Mining using the MDL Principle

Micky Faas and Matthijs van Leeuwen

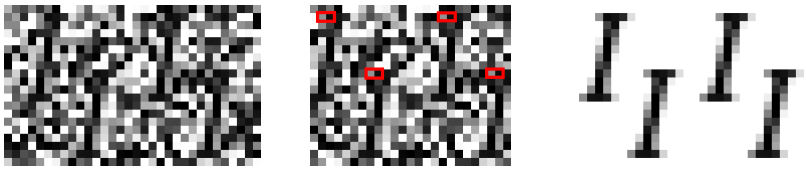
Leiden Institute for Advanced Computer Science, Leiden University

Abstract. We introduce geometric pattern mining, the problem of finding recurring local structure in discrete, geometric matrices. It differs from existing pattern mining problems by identifying complex spatial relations between elements, resulting in arbitrarily shaped patterns. After we formalise this new type of pattern mining, we propose an approach to selecting a set of patterns using the Minimum Description Length principle. We demonstrate the potential of our approach by introducing Vouw, a heuristic algorithm for mining exact geometric patterns. We show that Vouw delivers high-quality results with a synthetic benchmark.

1 Introduction

Frequent pattern mining [1] is the well-known subfield of data mining that aims to find and extract recurring substructures from data, as a form of knowledge discovery. The generic concept of pattern mining has been instantiated for many different types of patterns, e.g., for item sets (in Boolean transaction data), subgraphs (in graphs/networks), and episodes (in sequences). So far, however, little research has been done on pattern mining for raster-based data, i.e., geometric matrices in which the row and column orders are fixed. The exception is geometric tiling [3, 10], but that problem only considers tiles, i.e., rectangular-shaped patterns, in Boolean data.

In this paper we generalise this setting in two important ways. First, we consider geometric patterns *of any shape* that are geometrically connected, i.e., it must be possible to reach any element from any other element in a pattern by only traversing elements in that pattern. Second, we consider *discrete geometric data* with any number of possible values (which includes the Boolean case). We call the resulting problem *geometric pattern mining*.



(a) 32×24 'geometric matrix'. **(b)** Pair $(146, 11)$. **(c)** Pattern 'I' occurs four times.

Fig. 1: Geometric pattern mining example. Each element is in $[0, 255]$.

Figure 1 illustrates an example of geometric pattern mining. Figure 1a shows a 32×24 grayscale 'geometric matrix', with each element in $[0, 255]$, apparently filled with noise. If we take a closer look at all horizontal pairs of elements, however, we find that the pair $(146, 11)$ is, amongst others, more prevalent than expected from 'random noise' (Figure 1b). If we would continue to try all combinations of elements that 'stand out' from the background noise, we would eventually find four copies of the letter 'I' set in 16 point Garamond Italic (Figure 1c).

The 35 elements that make up a single 'I' in the example form what we call a *geometric pattern*. Since its four occurrences jointly cover a substantial part of the matrix, we could use this pattern to describe the matrix more succinctly than by 768 independent values. That is, we could describe it as the pattern 'I' at locations $(5, 4), (11, 11), (20, 3), (25, 10)$ plus 628 independent values, hereby separating structure from accidental (noise) data. Since the latter description is shorter, we have compressed the data. At the same time we have learned something about the data, namely that it contains four I's. This suggests that we can use compression as a criterion to find patterns that describe the data.

Approach and contributions. Our first contribution is that we introduce and formally define *geometric pattern mining*, i.e., the problem of finding recurring local structure in geometric, discrete matrices. Although we restrict the scope of this paper to two-dimensional data, the generic concept applies to higher dimensions. Potential applications include the analysis of satellite imagery, texture recognition, and (pattern-based) clustering.

We distinguish three types of geometric patterns: 1) *exact* patterns, which must appear exactly identical in the data to match; 2) *fault-tolerant* patterns, which may have noisy occurrences and are therefore better suited to noisy data; and 3) *transformation-equivalent* patterns, which are identical after some transformation (such as mirror, inverse, rotate, etc.). Each consecutive type makes the problem more expressive and hence more complex. In this initial paper we therefore restrict the scope to the first, exact type.

As many geometric patterns can be found in a typical matrix, it is crucial to find a compact set of patterns that together describes the structure in the data well. We regard this as a model selection problem, where a model is defined by a set of patterns. Following our observation above, that geometric patterns can be used to compress the data, our second contribution is the formalisation of the model selection problem by using the *Minimum Description Length (MDL) principle* [7, 4]. Central to MDL is the notion that 'learning' can be thought of as 'finding regularity' and that regularity itself is a property of data that is exploited by *compressing* said data. This matches very well with the goals of pattern mining, as a result of which the MDL principle has proven very successful for MDL-based pattern mining [11, 6].

Finally, our third contribution is Vouw, a heuristic algorithm for MDL-based geometric pattern mining that (1) finds compact yet descriptive sets of patterns, (2) requires no parameters, and (3) is tolerant to noise in the data (but not in the occurrences of the patterns). We empirically evaluate Vouw on synthetic data and demonstrate that it is able to accurately recover planted patterns.

2 Related Work

As the first pattern mining approach using the MDL principle, Krimp [11] was one of the main sources of inspiration for this paper. Many papers on pattern-based modelling using MDL have appeared since, both improving search, e.g., Slim [9], and extensions to other problems, e.g., Classy [6] for rule-based classification.

The problem closest to ours is probably that of geometric tiling, as introduced by Gionis et al. [3] and later also combined with the MDL principle by Tatti and Vreeken [10]. Geometric tiling, however, is limited to Boolean data and rectangularly shaped patterns (tiles); we strongly relax both these limitations (but as of yet do not support patterns based on densities or noisy occurrences).

Campana et al. [2] also use matrix-like input data (textures) and develops a compression-based similarity measure. Their method, however, cannot be used for *explanatory* data analysis as it relies on a generic image compression algorithm that is essentially a black box.

3 Geometric Pattern Mining using MDL

We define geometric pattern mining on bounded, discrete and two-dimensional raster-based data. We represent this data as an $M \times N$ matrix A whose rows and columns are finite and in a fixed ordering (i.e. reordering rows and columns semantically alters the matrix). For elements $a_{i,j}$, where row i is on $[0; N)$ and column j is on $[0; M)$, holds that $a_{i,j} \in S$, the finite set of symbols over A.

According to MDL, the shortest (optimal) description of A reveals all structure of A in the most succinct way possible and we approximate this optimal description through the compression of the original data. This optimal description A' is only optimal if we can unambiguously reconstruct A from it and nothing more — the compression is both minimal and lossless. Intuitively compression means defining A using as few building blocks as possible. We illustrate this in Figure 2. Given the matrix A we decompose it in patterns, denoted X and Y. The set of all these patterns is the **model** of a A, denoted H_A. In order to reconstruct A from this model, we alse need a mapping from the H_A back to A. This mapping represents in what MDL calls the **the data given the model** H_A. In this context we think of it as 'instructions' of how to reconstruct A. Let us call the set of all instructions required to rebuild A from H_A the **instantiation** of H_A. It is denoted by I_A in the example. The result is a notation that allows us to express matrix A as if decomposed into sets of local and global spatial information, which we will now describe in more detail.

$$A = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & 1 & 1 \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ 1 & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & 1 \\ 1 & 1 & 1 & 1 & \cdot & \cdot \end{bmatrix},\ I = \begin{bmatrix} X & \cdot & \cdot & \cdot & Y & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ X & \cdot & \cdot & \cdot & X & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ Y & \cdot & Y & \cdot & \cdot & \cdot \end{bmatrix},\ H = \{X = \begin{bmatrix} 1 & \cdot \\ \cdot & 1 \end{bmatrix}, Y = \begin{bmatrix} 1 & 1 \end{bmatrix}\}$$

Fig. 2: Example decomposition of A into instantiation I and patterns X, Y

3.1 Patterns and Instances

▷ *We define a* ***pattern*** *as a $M_X \times N_X$ submatrix X of the original matrix A. Elements of this submatrix may be $\cdot$, the empty element, which gives us the ability to cut-out any irregular-shaped part of A. We additionally require the elements of X to be adjacent (horizontal, vertical or diagonal) to at least one non-empty element and that no rows and columns are empty.*

From this definition, the dimensions $M_X \times N_X$ give the smallest rectangle around X (the *bounding box)*. We also define the cardinality $|X|$ of X as the number of non-empty elements. We call a pattern X with $|X| = 1$ a **singleton pattern**, i.e. a pattern containing exactly one element of A.

Each pattern contains a special **pivot** element: $pivot(X)$ is the first non-empty element of X. A pivot can be thought of as a fixed point in X which we can use to position its elements in relation to A. This translation, or **offset**, is a tuple $q = (i, j)$ that is on the same domain as an index in A. We realize this translation by placing all elements of X on an empty $M \times X$ size matrix in such that the pivot element is at (i, j). We formalize this in the **instantiation operator** $\otimes$:

▷ *We define the* ***instance*** $X \otimes (i, j)$ *as the $M \times N$ matrix containing all elements of X such that* $\text{pivot}(X)$ *is at index (i, j) and the distances between all elements are preserved. The resulting matrix contains no additional non-empty elements.*

Obviously this does not yield a valid result for an arbitrary offset (i, j). We want to limit ourselves to the space of pattern instances that are actually valid in relation to matrix A. Therefore two simple constraints are needed: (1) an instance must be **well-defined**: *placing* $\text{pivot}(X)$ *is at index (i, j) results in an instance that contains all elements of X, and (2) elements of instances cannot overlap, meaning each element of A should be described at most once.* This allows for a description that is both unambiguous and minimal.

▷ *Two pattern instances $X \otimes q$ and $Y \otimes r$, with $q \neq r$ are* ***non-overlapping*** *if $|(X \otimes q) + (Y \otimes r)| = |X| + |Y|$.*

From here on we will use the same letter in lower case to denote an arbitrary instance of a pattern, e.g. $x = X \otimes q$ when the exact value of q is unimportant. Since instances are simply patterns projected on an $M \times N$ matrix, we can reverse $\otimes$ by removing all completely empty rows and columns:

▷ *Let $X \otimes q$ be an instance of X, then by definition we say that $\oslash(X \otimes q) = X$.*

We briefly introduced the instantiation I as a set of 'instructions' of where instances of each pattern should be positioned in order to obtain A. As Figure 2 suggests, this mapping has the shape of an $M \times N$ matrix.

▷ *Given the set of patterns H, the* ***instantiation (matrix)*** *I is an incomplete $M \times N$ matrix such that $I_{i,j} \in H \cup \{\cdot\}$ for all (i, j). For all non-empty elements $I_{i,j}$ it holds that $I_{i,j} \otimes (i, j)$ is a non-overlapping instance of $I_{i,j}$ in A.*

3.2 The Problem and its Solution Space

Patterns can be constructed by joining smaller patterns in a bottom-up fashion. We can define the exact way two patterns should be joined by enumerating the

distance of their respective pivots. To limit the possibilities to patterns relevant to A, instances can be used as an intermediate step. As Figure 3 demonstrates, we can use a simple element-wise matrix addition to sum two instances and use $\oslash$ to obtain a joined pattern. Here we start by instantiating X and Y with offsets $(1,0)$ and $(1,1)$ respectively. We add the resulting x and y to obtain $\oslash z$, the union of X and Y with relative offset $(1,1)-(1,0)=(0,1)$.

The Sets $\mathcal{H}_A$ and $\mathcal{I}_A$ We define the **model class** $\mathcal{H}$, the set of all possible models for all possible inputs. Without any prior knowledge, this is the search space of our algorithm. We will first look at a more bounded subset $\mathcal{H}_A$ of all possible models for A, and $\mathcal{I}_A$, the set of all possible instantiations to these models. We will also take H_A^0 to be the model with only singleton patterns. As singletons are just individual elements of A, we can simply say that $H_A^0 = S$. The instantiation matrix corresponding to H_A^0 is denoted I_A^0. Given that each element of this matrix must correspond to exactly one element of A in H_A^0, we see that each $I_{i,j} = a_{i,j}$ and so I_A^0 is equal to A.

Using H_A^0 and I_A^0 as base cases we can now inductively define the set $\mathcal{I}_A$:

Base case $I_A^0 \in \mathcal{I}_A$

By induction If I is in $\mathcal{I}_A$ then take any pair $I_{i,j}, I_{k,l} \in I$ such that $(i,j) \leq (k,l)$ in lexicographical order. Then the set I' is also in $\mathcal{I}_A$, providing I' equals I except:

$$I'_{i,j} := \oslash\big(I_{i,j} \otimes (i,j) + I_{k,l} \otimes (k,l)\big)$$
$$I'_{k,l} := \cdot$$

This shows we can add any two instances together, which are by definition always non-overlapping and thus valid in A, in any order and obtain an element of $\mathcal{I}_A$. Eventually this results in just one big instance that is equal to A. Note that when we take two elements $I_{i,j}, I_{k,l} \in I$ we force $(i,j) \leq (k,l)$, not only to eliminate different routes to the same instance matrix, but also such that the pivot of the new pattern coincides with $I_{i,j}$. We can then leave $I_{k,l}$ empty.

The construction of $\mathcal{I}_A$ also implicitly defines $\mathcal{H}_A$. While this may seem odd — defining models for instantiations instead of the other way around — note that there is no unambiguous way to find one instantiation for a given model. Instead we find the following definition by applying the inductive construction:

$$\mathcal{H}_A = \big\{\{\oslash(I) \mid I \in I\} \;\big|\; I \in \mathcal{I}_A\big\}. \tag{1}$$

So for any instantiation $I \in \mathcal{I}_A$ there is a corresponding set in $\mathcal{H}_A$ of all patterns that occur in I. This results in an interesting symbiosis between model and

$$x = X \otimes (1,0) = \begin{bmatrix} \cdot & \cdot \\ 1 & \cdot \\ \cdot & 1 \end{bmatrix},\ y = Y \otimes (1,1) = \begin{bmatrix} \cdot & \cdot \\ \cdot & 1 \\ \cdot & \cdot \end{bmatrix}, x + y = \begin{bmatrix} \cdot & \cdot \\ 1 & 1 \\ \cdot & 1 \end{bmatrix},\ Z = \oslash(x+y) = \begin{bmatrix} 1 & 1 \\ \cdot & 1 \end{bmatrix}$$

Fig. 3: Example of joining patterns X and Y to construct Z.

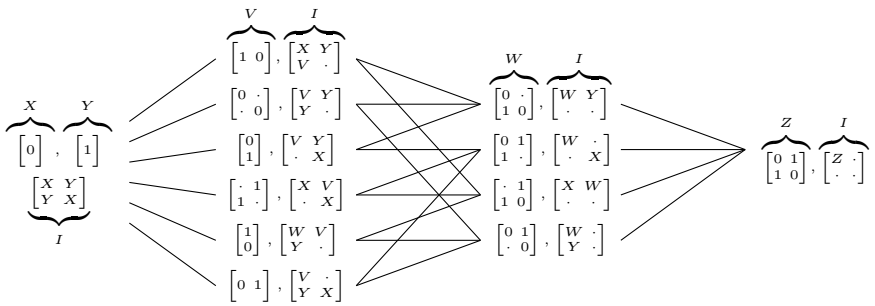


Fig. 4: Model space lattice for a 2×2 Boolean matrix. The Left hand column shows which pattern is added in each step, I is the current instantiation.

instantiation: increasing the complexity of one decreases that of the other. This construction gives a tightly connected lattice as shown in Figure 4.

3.3 Encoding Models and Instances

From all parametrized models in $\mathcal{H}_A$ we want to select (approximate) the model that describes A best. We use two-part MDL to quantify how well a given model and instantiation matrix fit A. Two-part MDL tells us to minimize the sum of $L_1(H_A) + L_2(A|H_A)$, two functions that give the length of the model and the length of 'the data given the model', respectively. In this context, the model is the set of patterns H_A and the data given the model is I_A, the accidental information needed to reconstruct the data from H_A.

In order to compute their lengths, we need to decide on a way to encode H and I first. This encoding is of great influence on the length functions and therefore on the ability to accurately quantify the fit of a given H and I to A. Although the decision for this encoding is obviously prone to bias, practice shows that good results can be had once certain conditions are met: (1) all data is encoded (lossless) and (2) the encoding is as concise as possible (nothing but the data is encoded). Based on these conditions we give length functions for pattern sets and instantiations, but we do not actually need to encode them.

The fictional encoder sequentially sends each symbol in the datastream to the 'decoder' using a code word. Information theory tells us that we optimal length of a code word is given by $-\log(p)$[1], where p is the exact probability that the code word occurs in the output. We therefore need not compute the actual code words, just their probabilities. For this to work, both the encoder and hypothetical decoder must know either the exact probability distribution or agree upon an approximation beforehand. Such approximation is often called a **prior** and is used to fix 'prior knowledge' that does not have to be encoded explicitly.

[1] We calculate lengths in bits and therefore all logarithms in this paper have base 2.

Table 1: Length computation for the different classes of matrices. The total length is the sum of the listed terms.

Matrix	Bounds	# Elements	Positions	Symbols
$L_p(X)$Pattern	$\log(MN)$	$L_{\mathbb{N}}\binom{M_X N_X}{\lvert X\rvert}$		$\log(\lvert S\rvert)$
$L_1(H)$Model	N/A	$L_N(\lvert H\rvert)$	N/A	$L_p(X \in H)$
$L_2(I)$ Instantiation	*constant*	$\log(MN)$	*implicit*	$L_{pp}(I)$

The length function for incomplete matrices. To losslessly encode A' we have to encode both H and I individually. As both instances and patterns are both matrices it is tempting to utilize the same length function for both. Empirical evidence has shown that this is not a good idea though. For example, we do not consider certain values such as the size of the instance matrix because it is constant, however, the size of each individual pattern is not. We therefore have to construct a different length function for each type of matrix. These are listed in Table 1 When encoding I, we observe that it contains each pattern $X \in H$ multiple times, given by the **usage** of X. Using **prequential plug-in code** [4] to encode I enables us to omit encoding these usages separately, which would create unwanted bias. Prequential plug-in code gives us the following length function for I. We elaborate on the derivation of this equation in Appendix A.

$$L_{pp}(I \mid P_{plugin}) = -\sum_{X_i \in h}^{|H|} \left[\log \frac{\Gamma(\text{usage}(X_i) + \epsilon)}{\Gamma(\epsilon)}\right] + \log \frac{\Gamma(|I| + \epsilon|H|)}{\Gamma(\epsilon|H|)} \tag{2}$$

Each length function has four terms. First we encode the total size of the matrix. Since we assume MN to be known/constant, we can use this constant to define the uniform distribution $\frac{1}{MN}$, such that $\log MN$ encodes an arbitrary index of A. Next we encode the number of elements that are non-empty. For patterns this value is encoded together with the third term, namely the positions of the non-empty elements. We use the previously encoded $M_X N_X$ in the binominal function to enumerate the ways we can place the $(|X|)$ elements onto a grid of $M_X N_X$. This gives us both *how many* non-empties there are as well as *where* they are. Finally the fourth term is the length of the actual symbols that encode the elements of matrix. In case we encode single elements of A, we simply assume that each unique value in A has an equal possibility of occurring. For the instance matrix, which encodes symbols to patterns, the prequential code is used as demonstrated before. Notice that L_N is the universal prior for integers[8] that can be used to encode integers on an arbitrary range.

4 The Vouw Algorithm

Pattern mining often yields vast search spaces and geometric pattern mining is no exception. Since by definition two-part MDL is already an approximation of Kolmogorov complexity, it does not make sense to try to compute an exact

solution and we therefore use the greedy heuristic widely used in MDL-based approaches [11, 9, 6]. We devise an inductive algorithm similar to the lattice in Figure 4: we start with a completely underfit model (left of the lattice), where there is one instance for each matrix element. On each iteration we combine two patterns, resulting in one or more pairs of instances to be merged (one step right in the lattice). We pick the pair of patterns that improve the compression ratio the most and we repeat these steps until no improvements are possible.

4.1 Finding candidates

The first step of the algorithm is to find the 'best' **candidate** pair of patterns to merge (Algorithm 1). Candidates are denoted as a tuple (X, Y, δ), where X and Y are patterns and δ the relative offset of X and Y as they occur in the data. All possibilities form an enormous vector space. Fortunately, we need only pairs of patterns and offsets that actually occur in the instance matrix. This means at each step we can directly enumerate candidates from the instantiation matrix and never even look at the original data.

The **support** of a candidate, written $\sup(X, Y, \delta)$, tells how often it is found in the instance matrix. Computing support is not completely trivial, as one candidate occurs multiple times in 'mirrored' configurations, such as (X, Y, δ) and $(Y, X, -\delta)$, which are equivalent but can still be found separately. Furthermore, due to the definition of a pattern, many potential candidates cannot be considered by the simple fact that their elements are not adjacent.

Peripheries. For each instance x we define its *periphery*: the set of instances which are positioned such that their union with x produces a valid pattern. This set is split into the *anterior*- $\mathrm{ANT}(X)$ and *posterior* $\mathrm{POST}(X)$ peripheries, containing instances that come before and after x in lexicographical order, respectively. This enables us to scan the instance matrix once, in lexicographical order. For each instance x, we only consider the instances $\mathrm{POST}(x)$ as candidates, thereby eliminating any (mirrored) duplicates.

Self-overlap. Self-overlap happens for candidates of the form (X, X, δ). In this case, too many or too few copies may be counted. Take for example a straight line of five instances of X. There are four unique pairs of two X's, but only two can be merged at a time, in three different ways. Therefore, when considering candidates of the form (X, X, δ), we also compute an *overlap coefficient*. This coefficient e is given by the equation $e = (2N_X + 1)\delta_i + \delta_j + N_X$. This equation essentially transforms δ into a one-dimensional coordinate space of all possible ways that X could be arranged *after* and *adjacent* to itself. For each instance x_1 a vector of bits $V(x)$ is used to remember if we have already encountered a combination x_1, x_2 with coefficient e, such that we do not count a combination x_2, x_3 with an equal e. This eliminates the problem of incorrect counting due to self-overlap.

Algorithm 1 Find Candidates

Input: I
Output: C
for all $x \in I$ **do**
  for all $y \in \text{POST}(x)$ **do**
    $X \leftarrow \oslash(x),\ Y \leftarrow \oslash(y)$
    $\delta \leftarrow \text{dist}(X, Y)$
    if $X = Y$ **then**
      if $V(x)[e] = 1$ **continue**
      $V(y)[e] \leftarrow 1$
    end if
    $C \leftarrow C\ \cup\ (X, Y, \delta)$
    $\sup(X, Y, \delta)$ += 1
  end for
end for

Algorithm 2 Vouw

Input: $H,\ I$
$C\ \leftarrow$ Find Candidates
$(X, Y, \delta) \in C : \forall_{c \in C} \Delta L((X, Y, \delta)) \leq \Delta L(c)$
$\Delta L_{best} = \Delta L((X, Y, \delta))$
if $\Delta L_{best} > 0$ **then**
  $Z \leftarrow \oslash(X \otimes (0, 0) + (Y \otimes \delta))$
  $H \leftarrow H \cup \{Z\}$
  for all $x_i \in I \mid \oslash(x_i) = X$ **do**
    for all $y \in \text{POST}(x_i) \mid \oslash(y) = Y$ **do**
      $x_i \leftarrow Z,\ y \leftarrow \cdot$
    end for
  end for
end if
repeat until $\Delta L_{best}\ <\ 0$

4.2 Gain computation

After candidate search we have a set of candidates C and their respective supports. The next step is to select the candidate that gives the best *gain*: the improvement in compression by merging the pattens in the candidate. For each candidate $c = (X, Y, \delta)$ the gain $\Delta L(A', c)$ is comprised of two parts: (1) the negative gain of adding the union pattern Z to the model H, resulting in H' and (2) the gain of replacing all instances x, y with relative offset δ by Z in I, resulting in I'. We use the length functions L_1, L_2 to derive an equation for gain:

$$\begin{aligned} \Delta L(A', c) &= \Big(L_1(H') + L_2(I')\Big) - \Big(L_1(H) + L_2(I)\Big) \\ &= L_0(|H|) - L_0(|H| + 1) - L_p(Z) + \Big(L_2(I') - L_2(I)\Big) \end{aligned} \tag{3}$$

As we can see, the terms with L_1 are simplified to $-L_p(Z)$ and the model's length because L_1 is simply a summation of individual pattern lengths. The equation of L_2 requires the recomputation of the entire instance matrix' length, which is expensive considering we need to perform it for *every candidate*, *every iteration*. However, we can rework the function L_{pp} in Equation (2) by observing that we can isolate the logarithms and generalize them into:

$$\log_G(a, b) = \log \frac{\Gamma(a + b\epsilon)}{\Gamma(b\epsilon)} = \log \Gamma(a + b\epsilon) - \log \Gamma(b\epsilon) \tag{4}$$

Which can be used to rework the second part of Equation (3) in such way that the gain equation can be computed in constant time complexity.

$$\begin{aligned} L_2(I') - L_2(I) = &\log_G(U(X), 1) + \log_G(U(Y), 1) \\ &- \log_G(U(X) - U(Z), 1) - \log_G(U(Y) - U(Z), 1) \\ &- \log_G(U(Z), 1) + \log_G(|I|, |H|) - \log_G(|I'|, |H'|) \end{aligned} \tag{5}$$

Notice that in some cases the usages of X and Y equal that of Z, which means additional gain is created by removing X and Y from the model.

4.3 Mining a Set of Patterns

In the second part of the algorithm we select the candidate (X, Y, δ) with the best gain and merge X and Y to form Z, as explained in Section 3.2. We linearly traverse I to replace all instances x and y with relative offset δ by instances of Z. (X, Y, δ) was constructed by looking in the posterior periphery of all x to find Y and δ, which means that Y always comes after X in lexicographical order. The pivot of a pattern is the first element in lexicographical order, therefore $\text{pivot}(Z) = \text{pivot}(X)$. This means that we can replace all matching x with an instance of Z and all matching y with $\cdot$. This concludes the baseline version of the algorithm, which is listed in Algorithm 2.

4.4 Improvements

Local search. Given a matrix containing some pattern X, the algorithm can arrive to X in different ways. Exploring these combinatorics can tell us how efficiently the algorithm arrives at X. By definition we know that the fundamental operation is to combine exactly two patterns on each step. Given this, the number of steps in which X can be constructed lies between $\log_2 |X|$ and $|X| - 1$.

To improve efficiency on large patterns without sacrificing the objectivity of the original heuristics, we add an optional local search. It is a result of the observation that the algorithm generates a large pattern X by adding small elements to a incrementally growing pattern, resulting in a behaviour that approaches $|X| - 1$ steps. Instead of taking all $|X| - 1$ steps to arrive at X, we can try to predict which elements will be added to X and merge them directly. Given that we selected candidate (X, Y, δ) and merged X and Y into Z, now for all m resulting instances $z_i \in z_0, \ldots, z_{m-1}$ we try to find pattern W and relative offset δ such that holds:

$$\forall_{i \in 0 \ldots m} \exists_w \in \text{ANT}(z_i) \cup \text{POST}(z_i) \ \cdot \ \oslash(w) = W \wedge dist(z_i, w) = \delta \qquad (6)$$

This yields zero or more candidates (Z, W, δ), which are then treated as any candidate: candidates with the highest gain are merged first until none exists with positive gain. This essentially means that we run the baseline algorithm only on the peripheries of all z_i, with the condition that the support of the candidates is equal to that of Z. Therefore we only expand Z during local search and we do not create new patterns.

Reusing candidates. We can improve performance by reusing the candidate set and slightly changing the search heuristic of the algorithm. The **Best-N** heuristic selects multiple candidates on each iteration as opposed to the baseline **Best-1** heuristic that only selects a single candidate with the highest gain. Best-N selects candidates in descending order of gain until no candidates with positive gain are left. Furthermore we only consider candidates that are all *disjoint*, because

when we merge candidate (X, Y, δ), remaining candidates with X and/or Y have unknown support and therefore unknown gain.

5 Experiments and Results

To asses the practical performance of the Vouw algorithm, we will primarily use the synthetic dataset generator Ril that was developed specifically for this purpose. Ril utilizes random walks to populate a matrix with patterns of a given size and prevalence, up to a specified density, while filling the remainder of the matrix with noise. Both the pattern elements and the noise are picked from the same uniform random distribution on the interval $[0; 255]$. The *signal-to-noise ratio* (SNR) of the data is defined as the number of pattern elements over the matrix size MN. The objective of the resulting experiment is that we try to find all of the signal (the patterns) and none of the noise. Figure 6 gives an overview of what the generated data looks like, how it is mined and evaluated.

Implementation. The implementation[2] used here, consists of the Vouw algorithm written in vanilla C/C++, a GUI and the synthetic benchmark Ril.

Evaluation. Completely random data (noise) is unlikely to be compressed. The SNR tells us how much noise is present in the data and thus conveniently gives us an upper bound of how much compression could be achieved. We use the ground truth SNR versus the resulting compression ratio as a benchmark to tell us how close we are in finding all the structure in the ground truth.

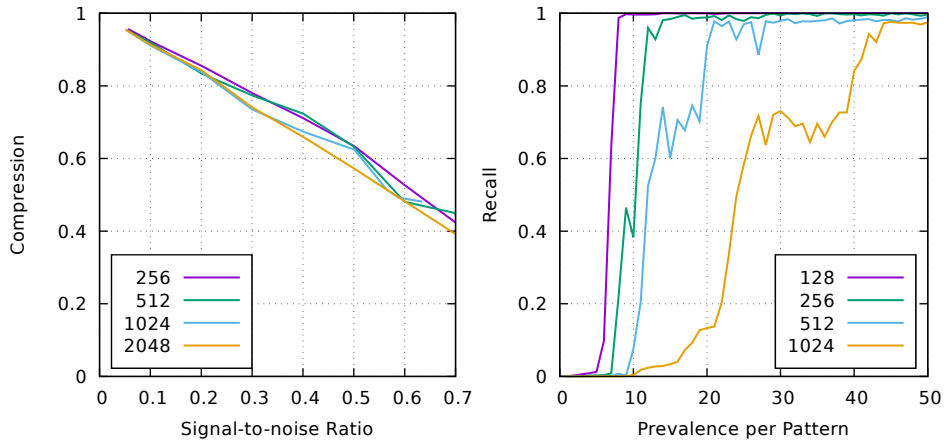
Because the compression ratio alone does not tell us the quality of the results, we also compare the ground truth matrix with the compressed result. We use the notion that instances of singleton patterns do not yield any compression and these elements must therefore be noise. Therefore we reconstruct the original matrix from the compressed result, while omitting any singleton patterns. This essentially gives us a matrix of 'positives' (signal) and 'negatives' (noise). By comparing each element with the corresponding element in the ground truth matrix, traditional figures for *precision* and *recall* can be calculated.

Figure 5a shows the influence of input SNR on compression ratio for different matrix sizes. We expect the compression ratio to be close to $1 - \text{snr}$. In Figure 5b shows that patterns with a low prevalence have a lower probability of being 'detected' by the algorithm as they are more likely to be accidental/noise. We see that increasing the matrix size also increases this threshold. In Table 2 we look at the influence of the two improvements upon the baseline algorithm as described in Section 4.4. In terms of quality, local search can improve the results quite substantial while Best-N notably *lowers* precision. Both improve speed by an order of magnitude, although the improvements given by Best-N are superior.

6 Conclusion

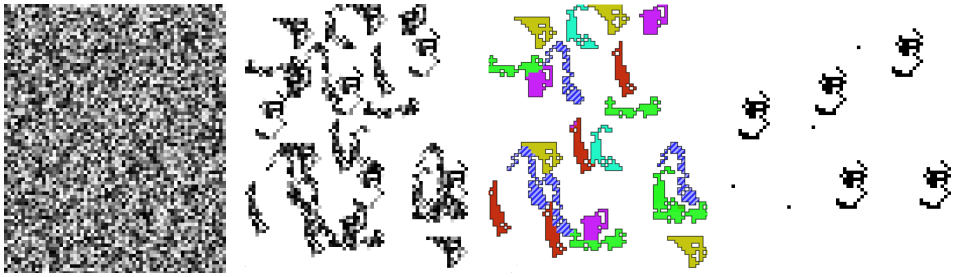
We have introduced geometric pattern mining, the problem of finding recurring structures in matrices or raster-based data, that we had not previously seen

[2] https://github.com/mickymuis/libvouw



(a) The influence of SNR in the ground truth **(b)** Prevalence versus recall

Fig. 5: Qualitative results for increasingly large matrices (sizes are squared).



(a) Generated matrix **(b)** Ground truth **(c)** Found patterns **(d)** Difference

Fig. 6: Synthetic patterns are added to a matrix filled with noise. The elements that belong to the ground truth are marked and compared to the matrix reconstructed by the algorithm. The difference is used to compute precision and recall.

Table 2: Performance measurements for the baseline algorithm and its optimizations. It can be seen that the optimizations influence both the speed and the quality of the results. Experiments were performed on an Intel Xeon-E2630v3 with 512GB RAM.

		Precision/Recall				Average time			
Size	SNR	None	Local	Best-N	Both	None	Local	Best-N	Both
256	.05	.98/.98	.99/.99	.93/.98	.95/.99	29s	1s	2s	1s
	.3	.99/.8	.99/.88	.96/.82	.99/.89	2m 32s	9s	5s	5s
512	.05	.98/.97	.99/.99	.87/.97	.93/.98	5m 26s	8s	20s	6s
	.3	.97/.93	.99/.99	.94/.91	.97/.90	26m 52s	2m 32s	24s	65s
1024	.05	.97/.98	.99/.99	.84/.98	.92/.96	21m 34s	44s	37s	34s
	.3	.98/.98	.99/.99	.93/.96	.98/.97	116m 4s	7m 31s	1m 49s	3m 31s

in literature. Compared to most pattern mining problems, it adds a layer of encoding geometric relations of data elements. Furthermore the problem can be split into three classes, of which the first class has been the focus of this paper.

We have also presented Vouw, an algorithm based on the MDL principle. The baseline algorithm is capable of generating high-quality results from synthetic datasets produced with our own dataset generated Ril. We have shown that the compression ratio achieved by the algorithm is on-par with what we would expect given the density (SNR) of the input data. By default, the algorithm has a bottleneck in the candidate search, which can be alleviated by the two demonstrated optimizations of the heuristics. It should also be mentioned that more optimizations on part of the implementation are possible (most notably parallelization), but these are outside the scope of this paper.

In future work we would like to generalize the formal definition and notation to n-dimensional data. Moreover, the framework can be expanded to cover all three problem classes. For the algorithm, we believe that it can benefit from more refinement on part of the encoding scheme in an effort to lower the threshold of pattern detection in larger matrices. Lastly we believe that expanding Vouw into similarity measure and clustering such demonstrated by [2], holds a very interesting premise.

References

1. Charu C. Aggarwal and Jiawei Han. *Frequent Pattern Mining.* Springer, 2014.
2. Bilson JL Campana and Eamonn J Keogh. A compression-based distance measure for texture. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 3(6):381–398, 2010.
3. Aristides Gionis, Heikki Mannila, and Jouni K. Seppänen. Geometric and combinatorial tiles in 0-1 data. In *Proceedings of PKDD 2004*, pages 173–184, 2004.
4. Peter D Grünwald. *The minimum description length principle.* MIT press, 2007.
5. Ming Li and Paul Vitányi. *An introduction to Kolmogorov complexity and its applications*, volume 3. Springer, 2008.
6. Hugo M Proença and Matthijs van Leeuwen. Interpretable multiclass classification by mdl-based rule lists. *arXiv preprint arXiv:1905.00328*, 2019.
7. Jorma Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465–471, 1978.
8. Jorma Rissanen. A universal prior for integers and estimation by minimum description length. *The Annals of statistics*, pages 416–431, 1983.
9. Koen Smets and Jilles Vreeken. Slim: Directly mining descriptive patterns. In *Proceedings of the 2012 SIAM International Conference on Data Mining*, pages 236–247. SIAM, 2012.
10. Nikolaj Tatti and Jilles Vreeken. Discovering descriptive tile trees - by mining optimal geometric subtiles. In *Proceedings of ECML PKDD 2012*, pages 9–24, 2012.
11. Jilles Vreeken, Matthijs van Leeuwen, and Arno Siebes. Krimp: mining itemsets that compress. *Data Mining and Knowledge Discovery*, 23(1):169–214, 2011.

A Appendix

A.1 Prequential Plugin-Code

To encode the instance matrix we use the **prequential plug-in code** [4]. The prequential plug-in code is defined for sequences of one item at a time and updates the probability of each item as it is encoded, such that the probability need not be known in advance. It has the favorable property of being asymptotically equal to the optimal code for large sequences. Say we want to encode all elements $I_i \in I$, we define:

$$P_{plugin}(y_i = I_i \mid y^{i-1}) = \frac{|\{y \in y^{i-1} \mid y = I_i\}| + \epsilon}{\sum_{X \in H} |\{y \in y^{i-1} \mid y = X\}| + \epsilon} \tag{7}$$

Here y_i is the i-th element to be encoded and y^{i-1} is the sequence of elements encoded so far. We initialize the base case (no element has been sent yet) with a pseudocount ϵ, which gives $P_{plugin}(y_1 = I \mid y^0) = \frac{\epsilon}{\epsilon|H|}$. We pick $\epsilon = 0.5$ as it is used generally with good results.

Let us adapt this principle to the problem of encoding patterns. The first step here is to determine the probability that each unique element (instance of a pattern) in I occurs.

▷ *Given a set of instances I, we define* $\text{usage}(X) = |\{I_i \in I \mid I_i = X\}|$.

From this definition we see that the **usage** of a pattern is a sum of how often it occurs as an instance. We can use this function to simplify things a little by realizing that we actually know the precise number of instances per pattern on the side of the decoder, but not as the decoder. This information can be used to slightly rephrase Equation 2 to be able to encode items in arbitrary order. This produces the length function of the instance matrix I as follows[3]:

$$\begin{aligned}
L_{pp}(I \mid P_{plugin}) &= \sum_{i=1}^{|I|} -\log \frac{|\{y \in y^{i-1} \mid y = I_i\}| + \epsilon}{\sum_{X \in H} |\{y \in y^{i-1} \mid y = X\}| + \epsilon} \\
&= \sum_{X_i \in h}^{|H|} -\log \prod_{j=0}^{\text{usage}(X_i)-1} \frac{j + \epsilon}{\sum_{k=1}^{i-1} U(X_k) + j + \epsilon|H|} \\
&= -\log \frac{\prod^{X_i \in H} \prod_{j=0}^{\text{usage}(X_i)} j + \epsilon}{\prod_{j=0}^{|I|-1} j + \epsilon|H|} \\
&= -\sum_{X_i \in h}^{|H|} \left[\log \frac{\Gamma(\text{usage}(X_i) + \epsilon)}{\Gamma(\epsilon)}\right] + \log \frac{\Gamma(|I| + \epsilon|H|)}{\Gamma(\epsilon|H|)}
\end{aligned} \tag{8}$$

[3] Here we use the fact that we can interchange sums of logarithms with logarithms of products and that those terms can be moved around freely. Moreover we convert the real-valued product sequences to the Gamma function Γ, which is the factorial function extended to real and complex numbers such that $\Gamma(n) = (n-1)!$.