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.*
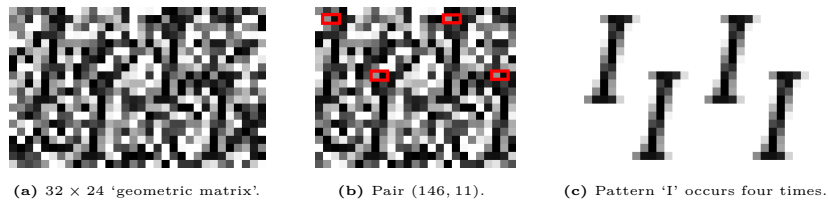

(a) 32 × 24 'geometric matrix'.   (b) Pair (146, 11).   (c) Pattern 'I' occurs four times.

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 describe 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 develop 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). Each element $a_{i,j} \in S$, where row $i \in [0;N)$, column $j \in [0;M)$, and S is a finite set of symbols, i.e., the alphabet of A.

According to the MDL principle, the shortest (optimal) description of A reveals all structure of A in the most succinct way possible. This optimal description is only optimal if we can unambiguously reconstruct A from it and nothing more—the compression is both minimal and lossless. Figure 2 illustrates how an example matrix could be succinctly described using patterns: matrix A is decomposed into patterns X and Y. A set of such patterns constitutes the **model** for a matrix A, denoted H_A (or H for short when A is clear from the context). In order to reconstruct A from this model, we also need a mapping from the H_A back to A. This mapping represents what (two-part) MDL calls the **the data given the model** H_A. In this context we can think of this as a set of all instructions required to rebuild A from H_A, which we call the **instantiation** of H_A and is denoted by I in the example. These concepts allow us to express matrix A as a decomposition into sets of local and global spatial information, which we will next describe in more detail.

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

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

3.1   Patterns and Instances

▷ *We define a **pattern** as an $M_X \times N_X$ 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 realise this translation by placing all elements of X in an empty $M \times X$ size matrix such that the pivot element is at (i, j). We formalise 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 $\mathrm{pivot}(X)$ is at index (i, j) and the distances between all elements are preserved. The resulting matrix contains no additional non-empty elements.*

Since this does not yield valid results for arbitrary offsets (i, j), we enforce two constraints: (1) an instance must be **well-defined**: placing $\mathrm{pivot}(X)$ at index (i, j) must result in an instance that contains all elements of X, and (2) elements of instances cannot *overlap*, i.e., each element of A can be described only once.

▷ *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 onto 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 a set of patterns H, the **instantiation (matrix)** I is an $M \times N$ matrix such that $I_{i,j} \in H \cup \{\cdot\}$ for all (i, j), where $\cdot$ denotes the empty element. For all non-empty $I_{i,j}$ 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

Larger patterns can be naturally constructed by joining (or merging) smaller patterns in a bottom-up fashion. To limit the considered patterns to those 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

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

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

$\oslash$ 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** $\mathcal{H}$ as the set of all possible models for all possible inputs. Without any prior knowledge, this would be the search space. To simplify the search, however, we only consider the more bounded subset $\mathcal{H}_A$ of all possible models for A, and $\mathcal{I}_A$, the set of all possible instantiations for these models. To this end we first define H_A^0 to be the model with only singleton patterns, i.e., $H_A^0 = S$, and denote its corresponding instantiation matrix by I_A^0. Given that each element of I_A^0 must correspond to exactly one element of A in H_A^0, we see that each $I_{i,j} = a_{i,j}$ and so we have $I_A^0 = A$.

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

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:

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

$$I'_{k,l} := \cdot$$

This shows we can add any two instances together, in any order, as they are by definition always non-overlapping and thus valid in A, and hereby obtain another 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 so 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:

$$\mathcal{H}_A = \bigl\{\{\oslash(x) \mid x \in I\} \mid I \in \mathcal{I}_A\bigr\}. \tag{1}$$

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 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 models in $\mathcal{H}_A$ we want to select the model that describes A best. Two-part MDL [4] tells us to choose that model that minimises the sum of
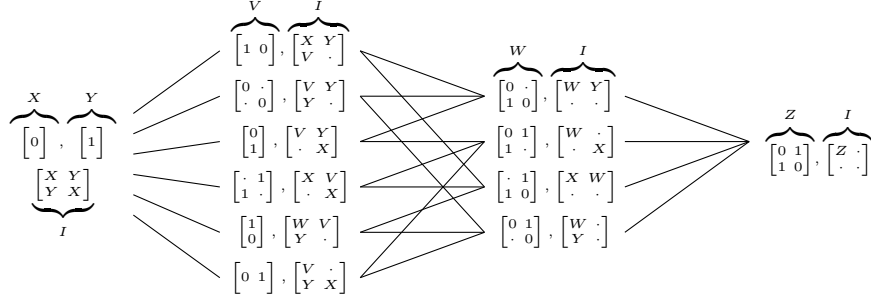
Fig. 4: Model space lattice for a 2×2 Boolean matrix. The V, W, and Z columns show which pattern is added in each step, while I depicts the current instantiation.

$L_1(H_A) + L_2(A|H_A)$, where L_1 and L_2 are two functions that give the length of the model and the length of 'the data given the model', respectively. In this context, the data given the model is given by I_A, which represents the accidental information needed to reconstruct the data A from H_A.

In order to compute their lengths, we need to decide how to encode H_A and I. As this encoding is of great influence on the outcome, we should adhere to the conditions that follow from MDL theory: (1) the model and data must be encoded losslessly; and (2) the encoding should be as concise as possible, i.e., it should be optimal. Note that for the purpose of model selection we only need the length functions; we do not need to actually encode the patterns or data.

Code length functions. Although the patterns in H and instantiation matrix I are all matrices, they have different characteristics and thus require different encodings. For example, the size of I is constant and can be ignored, while the sizes of the patterns vary and should be encoded. Hence we construct different length functions[1] for the different components of H and I, as 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 the **prequential plug-in code** [4] to encode I enables us to omit encoding these usages separately, which would create unwanted bias. The prequential plug-in code gives us the following length function for I. We use $\epsilon = 0.5$ and elaborate on its derivation in Appendix A[2].

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

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 $\frac{1}{MN}$, so that $\log MN$ encodes an arbitrary index of A. Next we encode the number of elements that are non-empty. For patterns

---

[1] We calculate code lengths in bits and therefore all logarithms have base 2.
[2] The appendix is available on https://arxiv.org/abs/1911.09587.

Table 1: Code length definitions. Each row specifies the code length given by the first column as the sum of the remaining terms.

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

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 assume that each unique value in A occurs with equal probability; without other prior knowledge, using the uniform distribution has minimax regret and is therefore optimal. For the instance matrix, which encodes symbols to patterns, the prequential code is used as demonstrated before. Note that L_N is the universal prior for the integers [8], which can be used for arbitrary integers and penalises larger integers.

4 The Vouw Algorithm

Pattern mining often yields vast search spaces and geometric pattern mining is no exception. We therefore use a heuristic approach, as is common in MDL-based approaches [11, 9, 6]. We devise a greedy algorithm that exploits the inductive definition of the search space as shown by the lattice in Figure 4. We start with a completely underfit model (leftmost in the lattice), where there is one instance for each matrix element. Next, in each iteration we combine two patterns, resulting in one or more pairs of instances to be merged (i.e., we move one step right in the lattice). In each step we merge the pair of patterns that improves compression most, and we repeat this until no improvement is possible.

4.1 Finding candidates

The first step is to find the 'best' **candidate** pair of patterns for merging (Algorithm 1). A candidates is denoted as a tuple (X, Y, δ), where X and Y are patterns and δ is the relative offset of X and Y as they occur in the data. Since we only need to consider pairs of patterns and offsets that actually occur in the instance matrix, we can directly enumerate candidates from the instantiation matrix and never even need to consider 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,

Algorithm 1 FindCandidates	**Algorithm 2** Vouw
Input: I	**Input:** H, I
Output: C	1: $C \leftarrow$ FindCandidates(I)
1: **for all** $x \in I$ **do**	2: $(X, Y, \delta) \in C : \forall_{c \in C} \Delta L((X, Y, \delta)) \leq \Delta L(c)$
2: **for all** $y \in$ POST(x) **do**	3: $\Delta L_{best} = \Delta L((X, Y, \delta))$
3: $X \leftarrow \oslash(x),\ Y \leftarrow \oslash(y)$	4: **if** $\Delta L_{best} > 0$ **then**
4: $\delta \leftarrow \text{dist}(X, Y)$	5: $Z \leftarrow \oslash(X \otimes (0, 0) + (Y \otimes \delta))$
5: **if** $X = Y$ **then**	6: $H \leftarrow H \cup \{Z\}$
6: **if** $V(x)[e] = 1$ **continue**	7: **for all** $x_i \in I \mid \oslash(x_i) = X$ **do**
7: $V(y)[e] \leftarrow 1$	8: **for all** $y \in$ POST$(x_i) \mid \oslash(y) = Y$ **do**
8: **end if**	9: $x_i \leftarrow Z, y \leftarrow \cdot$
9: $C \leftarrow C\ \cup\ (X, Y, \delta)$	10: **end for**
10: $\sup(X, Y, \delta)\ += 1$	11: **end for**
11: **end for**	12: **end if**
12: **end for**	13: **repeat until** $\Delta L_{best}\ <\ 0$

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*- ANT(X) and *posterior* 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 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 $e = (2N_X + 1)\delta_i + \delta_j + N_X$, which 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.

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 candidate pair of patterns. 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 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 generalise 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 are equal to 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, listed in Algorithm 2, we select the candidate (X, Y, δ) with the largest 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 $\cdot$.

4.4   Improvements

Local search. To improve the efficiency of finding large patterns without sacrificing the underlying idea of the original heuristics, we add an optional local search. Observe that without local search, Vouw generates a large pattern X by adding small elements to an incrementally growing pattern, resulting in a behaviour that requires up to $|X| - 1$ steps. To speed this up, we can try to

(a) Generated matrix   **(b)** Ground truth   **(c)** Found patterns   **(d)** Difference
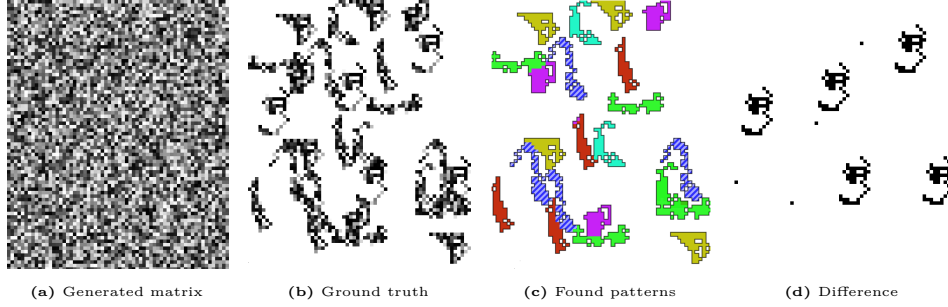
Fig. 5: Synthetic patterns are added to a matrix filled with noise. The difference between the ground truth and the matrix reconstructed by the algorithm is used to compute precision and recall.

'predict' which elements will be added to X and add them immediately. After selecting candidate (X, Y, δ) and merging X and Y into Z, for all m resulting instances $z_i \in z_0, \dots, z_{m-1}$ we try to find pattern W and offset δ such that

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

This yields zero or more candidates (Z, W, δ), which are then treated as any set of candidates: candidates with the highest gain are iteratively merged until no candidates with positive gain exist. 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.

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

To asses Vouw's practical performance we primarily use Ril, a synthetic dataset generator developed for this purpose. Ril utilises 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 experiment is to assess whether Vouw recovers all of the signal (the patterns)

and none of the noise. Figure 5 gives an overview of what the generated data looks like, and how it is mined and evaluated.

Implementation. The implementation[3] used 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 of the data is noise 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 to finding all the structure in the ground truth.

In addition, we also compare the ground truth matrix to the obtained model and instantiation. As singleton patterns do not yield any compression over the baseline model, we reconstruct the matrix omitting any singleton patterns. Ignoring the actual values, this gives us a Boolean matrix with 'positives' (pattern occurrence=signal) and 'negatives' (no pattern=noise). By comparing each element in this matrix with the corresponding element in the ground truth matrix, *precision* and *recall* can be calculated and evaluated.

Figure 6 (left) shows the influence of ground truth SNR on compression ratio for different matrix sizes. Compression ratio and SNR are clearly strongly correlated. Figure 6 (right) shows that patterns with a low prevalence (i.e., number of planted occurrences) have a lower probability of being 'detected' by the algorithm as they are more likely to be accidental/noise. 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 substantially while Best-N notably *lowers* precision. Both improve speed by an order of magnitude.

6   Conclusions

We introduced geometric pattern mining, the problem of finding recurring structures in discrete, geometric matrices, or raster-based data. Further, we presented Vouw, a heuristic algorithm for finding sets of geometric patterns that are good descriptions according to the MDL principle. The baseline algorithm is capable of accurately recovering patterns from synthetic data, and the resulting compression ratios are on par with the expectations based on the density of the data. Of the two improvements, especially the local search appears valuable as it improves precision and recall as well as runtime. For the future, we think that extensions to fault-tolerant patterns and clustering have large potential.

References

1. Charu C. Aggarwal and Jiawei Han. *Frequent Pattern Mining.* Springer, 2014.

---

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

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:

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

Table 2: Performance measurements for the baseline algorithm and its optimizations. Experiments were performed on an Intel Xeon-E2630v3 with 512GB RAM.

Size	SNR	Precision/Recall				Average time			
		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

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[4]:

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

---

[4] 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 $\Gamma(n) = (n-1)!$.
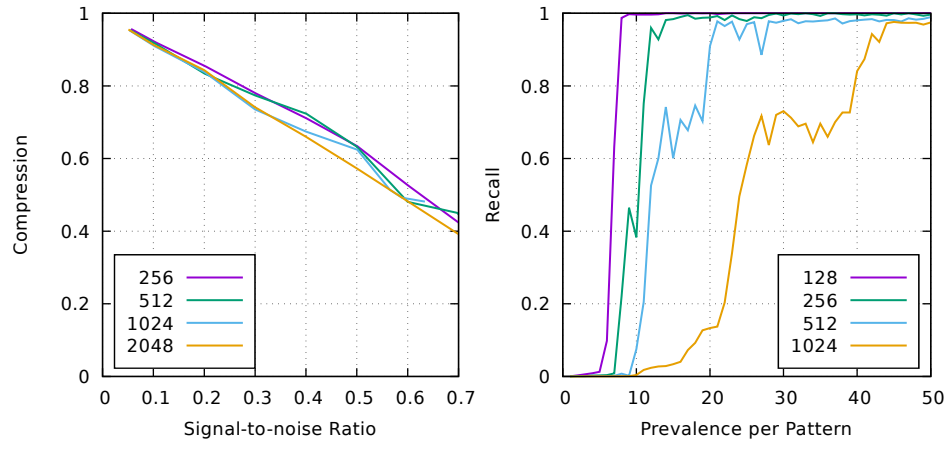
Fig. 6: The influence of SNR in the ground truth (left) and prevalence on recall (right).