1 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 1 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.

1.1 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

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

Fig. 1: Example decomposition of A into instantiation I and patterns X, Y.

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

Fig. 2: Example of joining patterns X and Y to construct a new pattern Z.

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* 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 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 1 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.*

1.2 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 2 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** $\mathcal{H}$ 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

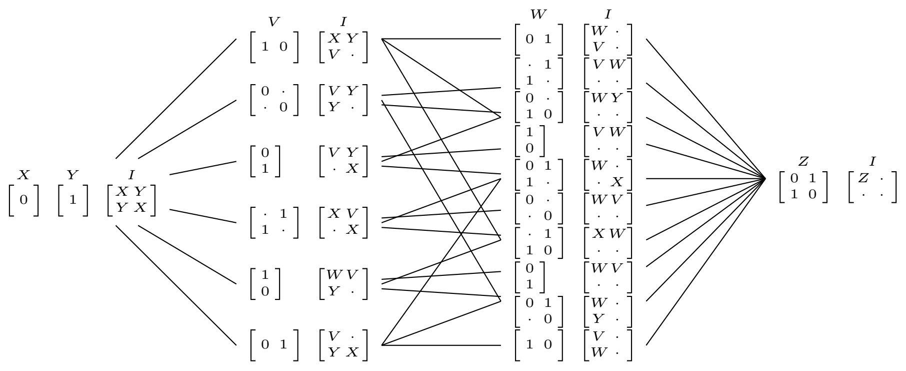
Fig. 3: Model space lattice for a 2×2 Boolean matrix. The V, W, and Z columns show which pattern is added in each step, while I depicts the current instantiation.

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:

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

This shows we can add any two instances together, 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:

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

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 instantiation: increasing the complexity of one decreases that of the other. This construction gives a tightly connected lattice as shown in Figure 3.

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	$\log(MN)$	$L_N\binom{M_X N_X}{\lvert X\rvert}$		$\lvert X\rvert \log(\lvert S\rvert)$
$L_1(H)$	Model	*N/A*	$L_N(\lvert H\rvert)$	*N/A*	$\sum_{X \in H} L_p(X)$
$L_2(I)$	Instantiation	*constant*	$\log(MN)$	*implicit*	$L_{pp}(I)$

1.3 Encoding Models and Instances

From all models in $\mathcal{H}_A$ we want to select the model that describes A best. Two-part MDL [**?**] tells us to choose that model that minimises the sum of $L_1(H_A) + L_2(A|H_A)$, 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** [**?**] 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 the Appendix[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 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

[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.

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 [**?**], which can be used for arbitrary integers and penalises larger integers.