1 Theoretical Framework

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). 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 occurring in A.

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

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

1.1 Patterns and Instances

▷ *We define a **pattern** as a $M_X \times N_X$ submatrix X of the original matrix A. This submatrix is not necessarily complete (elements may be $\cdot$, the empty element), which gives us the ability to precisely 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.*

From this definition we see that the dimensions $M_X \times N_X$ give essentially the smallest rectangle around X (the *bounding box).* As a more useful measure we therefore 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.

One element in each pattern is given the special function of **pivot**: $pivot(X)$ of pattern X is the first non-empty element of X in the first non-empty column of the first (non-empty) row of X. A pivot can be thought of as a fixed point in a pattern X which we can use to position its elements in relation to A. The precise translation we apply to a particular pattern we call an **offset**. An 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 way that the pivot element is at (i, j). We formalize this concept with the **instantiation operator** $\otimes$:

▷ *In the context of an $M \times N$ matrix A, we define the **instance** $X \otimes q$ as the incomplete $M \times N$ matrix containing all elements of X such that* $\text{pivot}(X)$ *is at index (i, j) and the distances between all original 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 a* $M \times N$ *matrix 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.

Given a set of patterns H over A, we would like to have a set of 'instructions' of where instances of each pattern should be positioned in order to obtain A. When looking at the example in Figure 1, 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 with the set of possible elements being* H*, i.e.* $I_{i,j} \in H$ *for all* (i, j)*. For all non-empty elements* $I_{i,j}$ *it holds that* $I_{i,j} \otimes (i, j)$ *is an instance of* $I_{i,j}$ *in* A *that is not non-overlapping with any other instance* $I_{i',j'} \otimes (i', j')$.

The above definition creates the interesting proposition that the offset to each instance is unique. Given that a pattern's pivot is placed exactly at one offset and that instances must be non-overlapping, makes this indeed believable. Later when we inductively define the set I of all instantiation matrices, this will be shown to be true more formally.

1.2 The Problem and its Solution Space

Constructing Patterns Patterns can be joined to create increasingly large patterns and, as we will later see, this concept forms the basis of the search algorithm. However, two patterns could be joined in any different number of ways. We can define the exact way two patterns should be joined by enumerating the distance of their respective pivots. We use pattern instances as an intermediate step.

Recall that instances are simply patterns projected on an $M \times N$ matrix, containing the same elements as the patterns at their original distances. This makes $\otimes$ trivially reversible 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 can now define a sum on two instances to get a new instance that combines both operands. Even though the result is not a pattern, we can always obtain it by using $\oslash$. The example in Figure 2 illustrates this process. We start by instantiating X and Y with offsets $(1, 0)$ and $(1, 1)$ respectively. This yields the non-overlapping x and y, which we simply add up to obtain z. The new pattern contained in z can be easily identified by removing the top empty row. We formally describe this mechanism in Theorem 1. As we will see later, this will become the fundamental operation of our algorithm.

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

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

Fig. 2: Example of Theorem 1. The 2×3 input matrix is not shown.

Theorem 1. *Given two non-overlapping instances $x = X \otimes q$ and $y = Y \otimes r$, the sum of the matrices $x + y$ is another instance. We observe that pattern $Z = \oslash(x + y)$ such that $x + y = Z \otimes q$.*

Notice that this sum has the limitation that two instances can only be summed if they do not overlap. While this is a serious limitation, we will show in the next subsection that it is not of any practical relevance.

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 of $\mathcal{H}$, namely the set $\mathcal{H}_A$ of all possible models for A, and its counterpart, the set $\mathcal{I}_A$ of all possible instantiations to these models. We will also take H_A^0 to be the model with only singleton patterns (patterns of length 1). 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 $\mathcal{I}_A$ of all instantiations of all models over 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

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

In this definition of I_A^0 we inductively replace two instances with their sum. Indeed we can add any two instances together in any order and eventually this results in just one big instance that is equal to A. The elegance in this is that, by this inductive definition, the instances never overlap and thus their sum is always a valid instance on A. Note that when we take two elements $I_{i,j}, I_{k,l} \in I$ we force $(i,j) \leq (k,l)$ to be in lexicographical order, 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 instantiation matrices also implicitly defines the corresponding models. While this may seem odd — defining models for instantiations instead of the other way around — note that there is no unambiguous way to find an instantiation matrix for a given model. Instead we find the following trivial definition by applying the inductive construction rule above:

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

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 instantiation: increasing the complexity of one decreases that of the other. When plotting this construction a tightly connected lattice appears such as that of Figure 3.

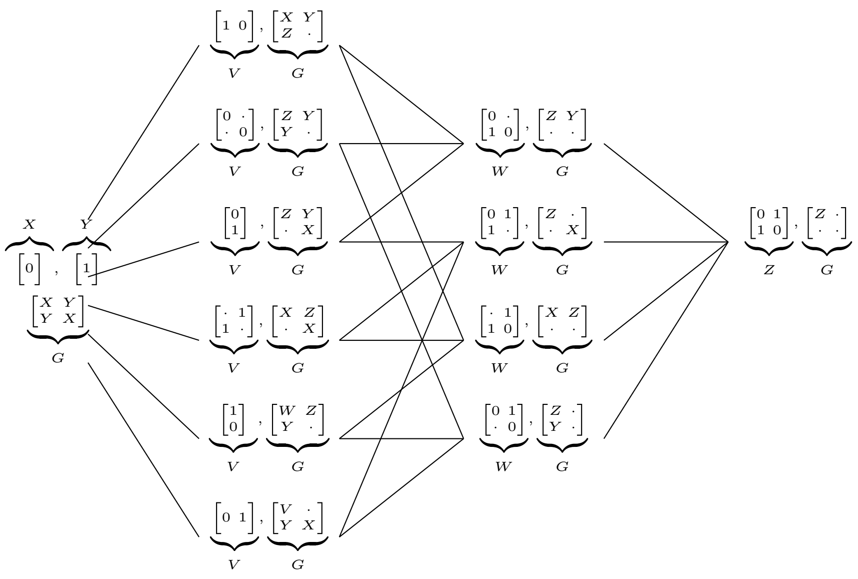

Fig. 3: The model space lattice for a 2×2 Boolean matrix. Here X and Y are the initial singleton patterns and G is the instance matrix. V and W are intermediate patterns and Z is the final, completely over fit, pattern.

1.3 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 the accidental information needed to reconstruct the data from H_A — which in this case is the instantiation matrix I_A.

In order to give said length functions, 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. The decision for this encoding is obviously prone to bias, which is one of the few disadvantages of two-part MDL. Fortunately, 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 instance matrices, but we do not actually need to encode them.

The fictional encoder sequentially sends each symbol in the datastream (either pattern set or instance matrix) to the 'decoder' using a code word. Information theory tells us that we optimal length of a code word is given by $-\log(p)$, 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.

To encode the instance matrix we will use the **prequential plug-in code** [**?**]. 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:

$$P_{plugin}(y_i = I_i \mid y^{i-1}) = \frac{|\{y \in y^{i-1} \mid y = I_i\}| + \epsilon}{\sum_{X \in H} |\{y \in y^{i-1} \mid y = X\}| + \epsilon} \tag{2}$$

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 Definition 1.3 to be able to encode items in arbitrary order.

This produces the length function of the instance matrix I as follows:

$$\begin{aligned} L_{pp}(I \mid P_{plugin}) &= \sum_{i=1}^{|I|} -\log \frac{|\{y \in y^{i-1} \mid y = I_i\}| + \epsilon}{\sum_{X \in H} |\{y \in y^{i-1} \mid y = X\}| + \epsilon} \\ &= -\log \frac{\prod^{X_i \in H} \prod_{j=0}^{\text{usage}(X_i)} j + \epsilon}{\prod_{j=0}^{|I|-1} j + \epsilon |H|} \\ &= -\sum_{X_i \in h}^{|H|} \left[\log \frac{\Gamma(\text{usage}(X_i) + \epsilon)}{\Gamma(\epsilon)} \right] + \log \frac{\Gamma(|I| + \epsilon|H|)}{\Gamma(\epsilon|H|)} \end{aligned} \tag{3}$$

The length function for incomplete matrices. To losslessly encode A' we have to encode both H and I individually. Recall that both instances and patterns are both matrices. It is therefore tempting to utilize the same length function for both. Empirical evidence has shown that this is not a good idea though and the main reason for this is the fact that prequential plug-in code behaves different for small sequences (patterns) than it does for large sequences (the instance matrix). Furthermore, 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.

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*	$L_p(X \in H)$
$L_2(I)$	Inst. mat.	*constant*	$\log(MN)$	*implicit*	$L_{pp}(I)$

Each length function has four (optional) 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 distribution $\log(MN)$. This is simply an uniform distribution that encodes an arbitrary index of A with equal lengths for each index. Next we encode the number of elements that are non-empty. Notice how for patterns, this value is encoded together with the third term, namely the positions of the non-empty elements. Because we have encoded $M_X N_X$ in the first term, we may now use it as a constant. We use it in the binominal function to enumerate the number of ways we can place the non-empty elements ($|X|$) 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[**?**] that can be used to encode integers on an arbitrary range.