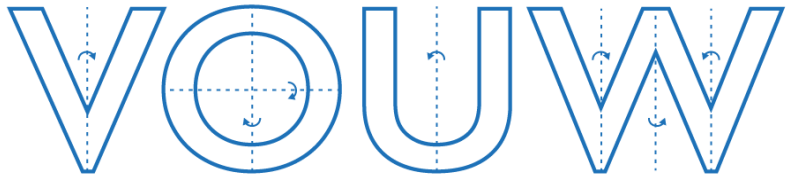


Pattern Mining on Matrices

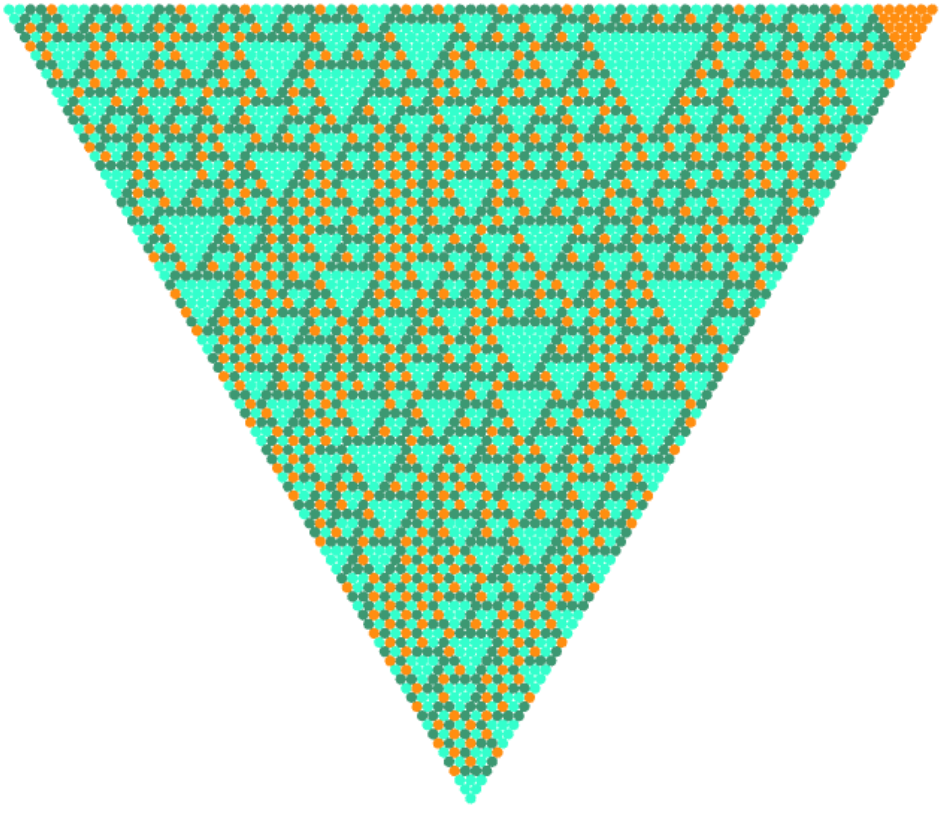
Micky Faas

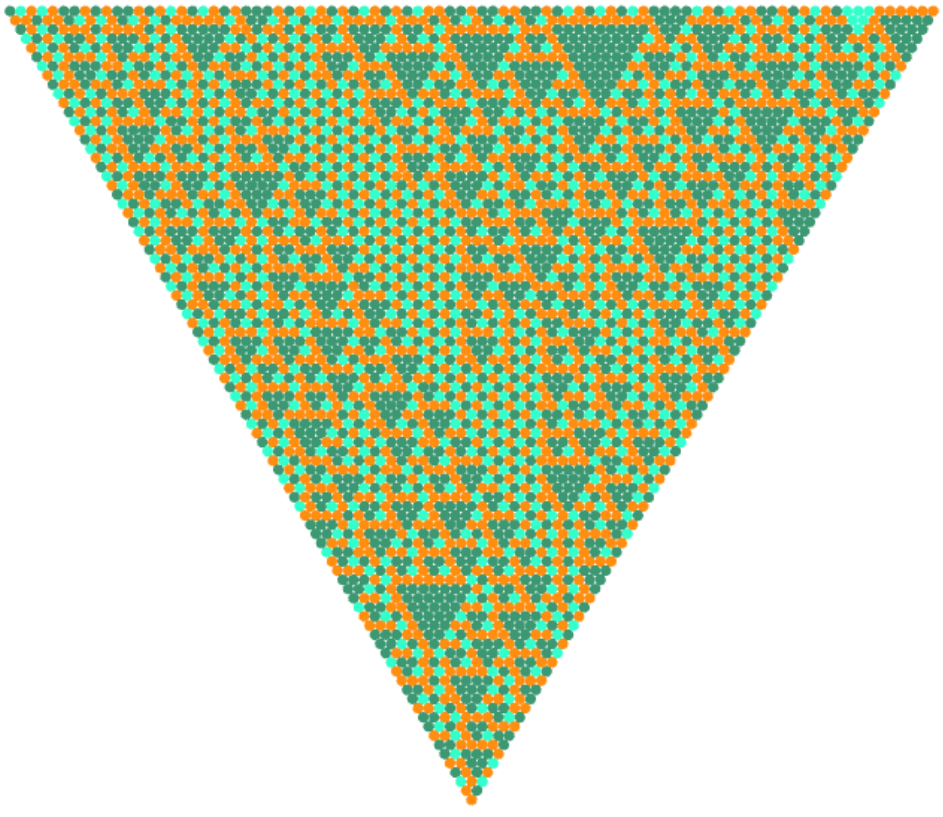
May 21, 2019

Overview

In this presentation I will:

* Introduce VOUW
* Give a brief description of the formal problem
* Explain the search method through some practical examples
* Tell you about my goals for the future





Originally: similarity in Cellular Automata

Introduction

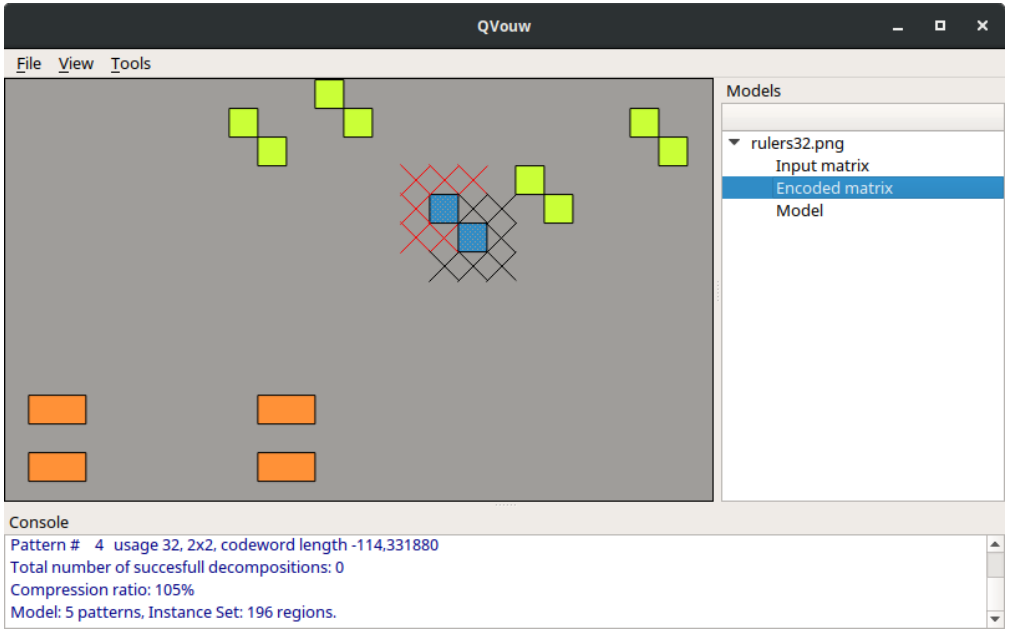
Theoretical problem: given some $M \times N$ matrix A, we want to discover and extract recurring structure. As a means of

- 'explaining' the data
- similarity measure or clustering

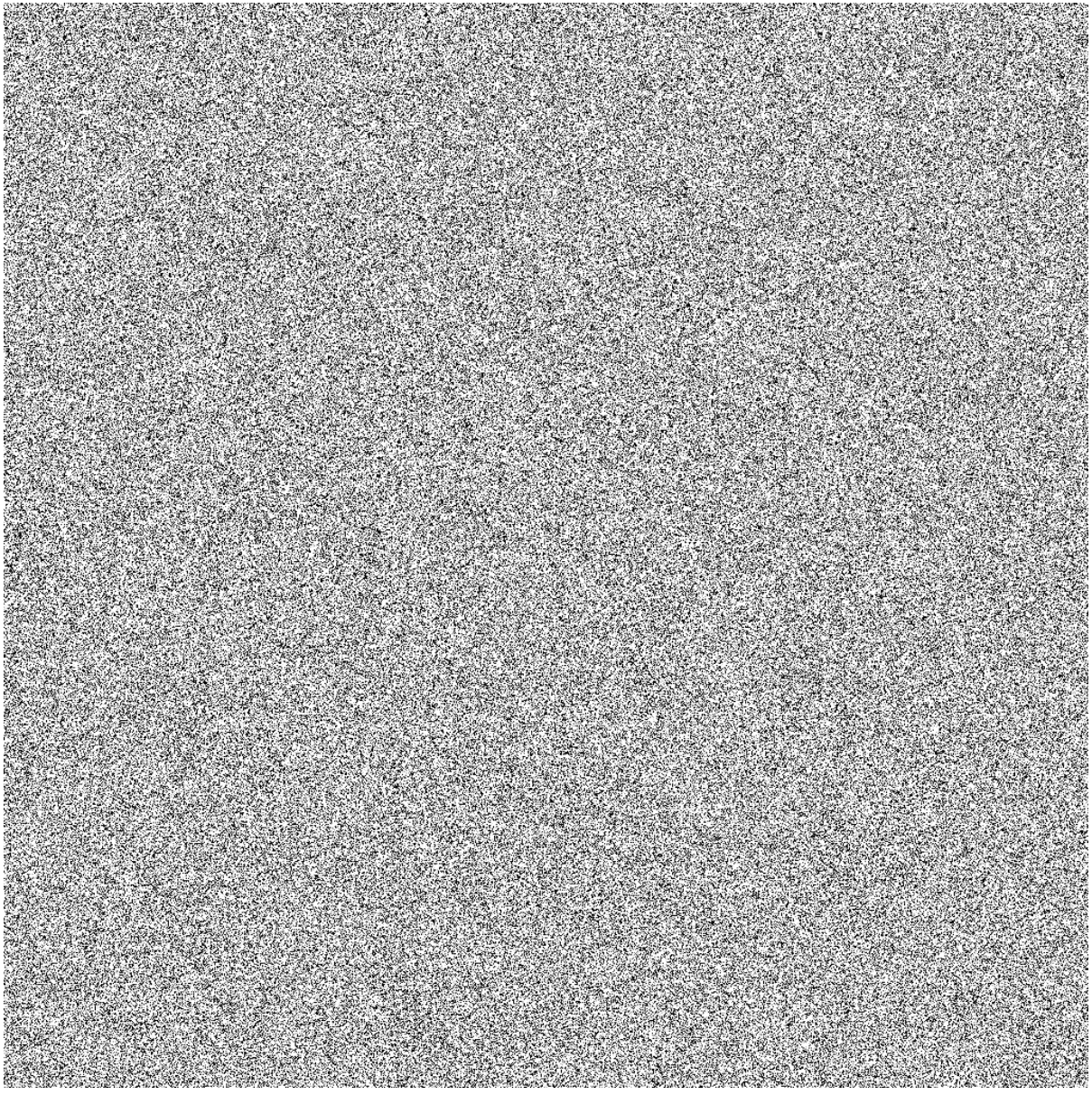
When compared to other pattern mining problems, we are specifically looking for

- Spatial relation/structure

Practical implementation: `libvouw` (library) and QVouw (GUI).



Is this just noise?



Patterns and Instances

Term?	Form	Encodes...
Pattern	Submatrix of A	...relative positions of elements
Offset	$(i, j) \in M \times N$	...position of entire pattern
Instance	$M \times N$ matrix	...absolute positions of elements
Instantiation	$M \times N$ matrix	...what pattern's instance should be at which index

$$\underbrace{\begin{bmatrix} 1 & \cdot \\ \cdot & 1 \end{bmatrix}}_{\text{Pattern}}, \underbrace{\begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix}}_{\text{Instance with offset (2,2)}}$$

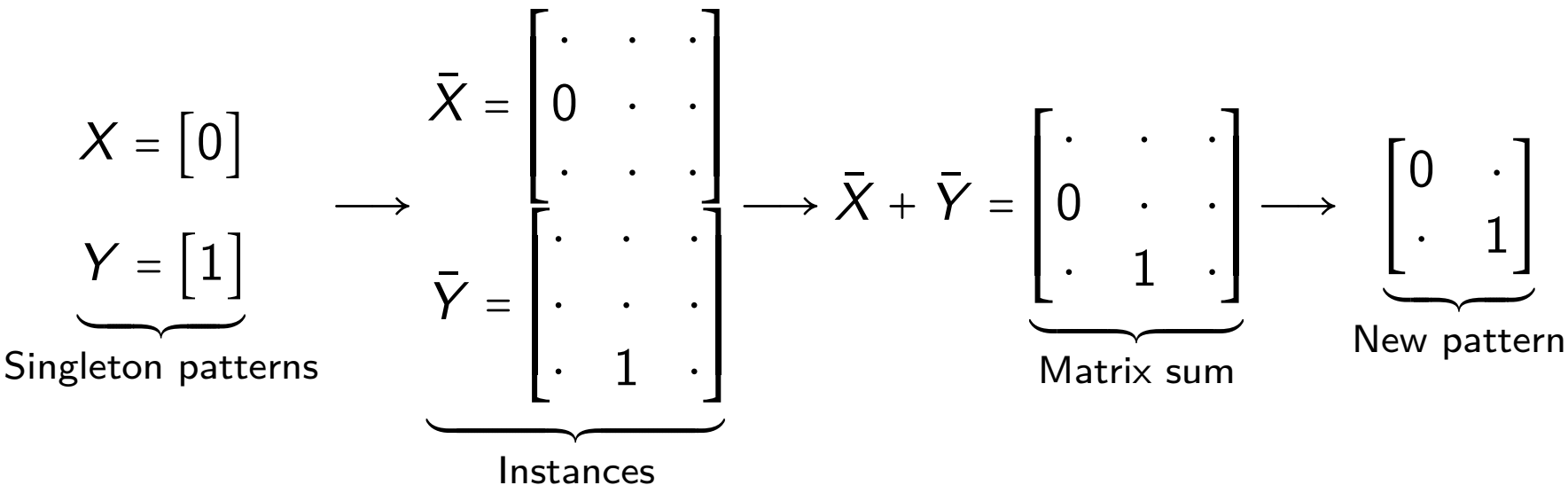
Example

$$A = \overbrace{\begin{bmatrix} 1 & . & . & . & . & 1 \\ . & 1 & . & . & . & . \\ 1 & . & . & . & 1 & . \\ . & 1 & . & . & . & 1 \\ 1 & . & 1 & . & . & . \\ . & 1 & . & 1 & . & . \end{bmatrix}}^{\text{Original matrix}}, G = \overbrace{\begin{bmatrix} X & . & . & . & . & Y \\ . & . & . & . & . & . \\ X & . & . & . & X & . \\ . & . & . & . & . & . \\ X & . & X & . & . & . \\ . & . & . & . & . & . \end{bmatrix}}^{\text{Instantiation}}$$

$$\overbrace{H = \{X = \underbrace{\begin{bmatrix} 1 & . \\ . & 1 \end{bmatrix}}_{\text{Pattern}}, Y = \underbrace{\begin{bmatrix} 1 \end{bmatrix}}_{\text{Pattern}} \}}^{\text{Model}}$$

Building Patterns

We can construct complex patterns by repeatedly combining simpler ones. We use instances for this as they encode the position of one pattern relative to another.



Model Space

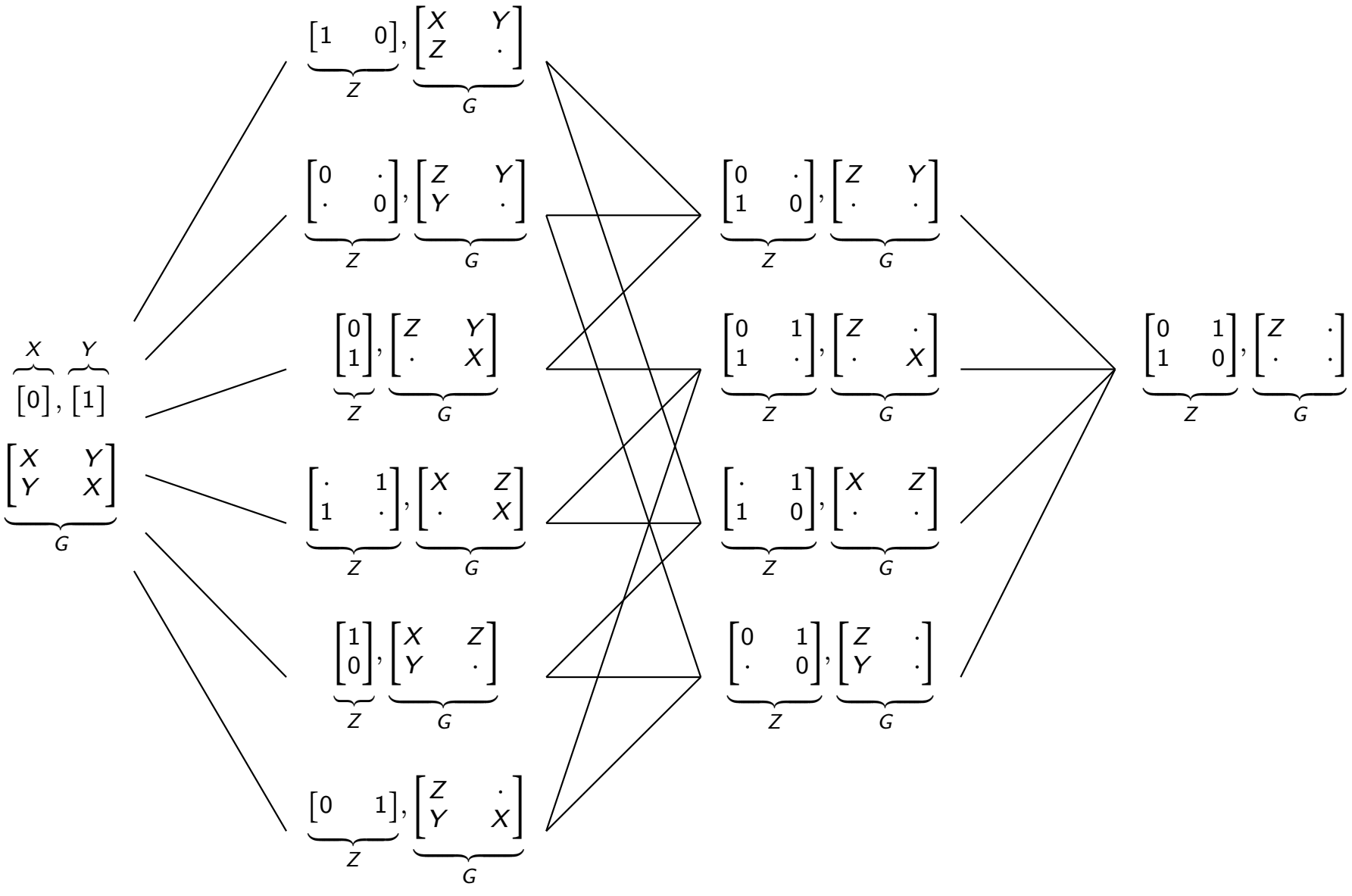
Idea: we inductively define the model space by starting with singletons for each element of A. Then we repeatedly merge instances until we finally obtain a pattern that equals A.

- At the beginning we have the completely underfit model
- We end up with a completely overfit model

Everything in between is a possible solution, but *we want the solution that describes A best.*

For example $A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$: what is the space of possible solutions?

2x2 Model Space Lattice



Worst-case complexity

How complex is this lattice? Very complex!
Idea: pick two instances to combine $MN-1$ times.

$$\prod_{n=0}^{MN-2} \binom{MN-n}{2} = \prod_{n=0}^{MN-2} \frac{MN-n}{2(MN-n-2)!} = \frac{(MN)!(MN-1)!}{2^{MN-1}}.$$

We will need to use heuristics.

Optimization using MDL

Idea: the best solution is a balance of model and instantiation complexity. We use two-part MDL:

$$\underbrace{L(H)}_{\text{Model}} + \underbrace{L(A|H)}_{\text{Data given model}}$$

In this case:

$$L\left(\underbrace{\overbrace{\begin{bmatrix} 1 & \cdot \\ \cdot & 1 \end{bmatrix}}^{\text{Pattern}}, \overbrace{\begin{bmatrix} 1 \end{bmatrix}}^{\text{Pattern}}}_{\text{Model}}\right) + L\left(\underbrace{\overbrace{\begin{matrix} X & \cdot & \cdot & \cdot & \cdot & Y \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ X & \cdot & \cdot & \cdot & X & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ X & \cdot & X & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{matrix}}^{\text{Instantiation Matrix}}}_{\text{Data given model}}\right)$$

Motivation

Why does this work? MDL is founded on these principles:

Kolmogorov Complexity of given data is the shortest computer program to produce that data.

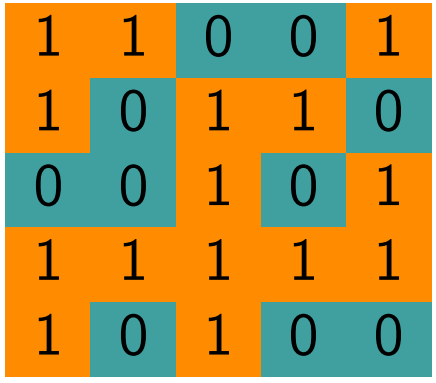
Occam's Razor . Given competing hypotheses, pick the one with the fewest assumptions.

No-hypercompression theorem . Data with no inherent structure, cannot be compressed.

Kraft Inequality . One-to-one correspondence between code lengths and probabilities.

A practical algorithm

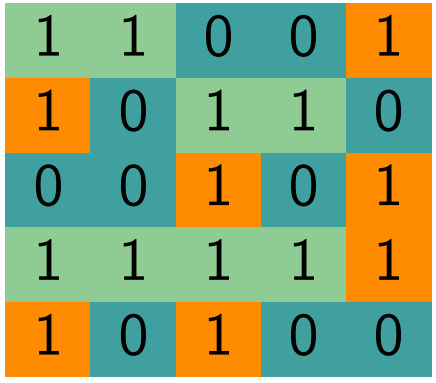
Instantiation



Patterns ('code table')

A practical algorithm

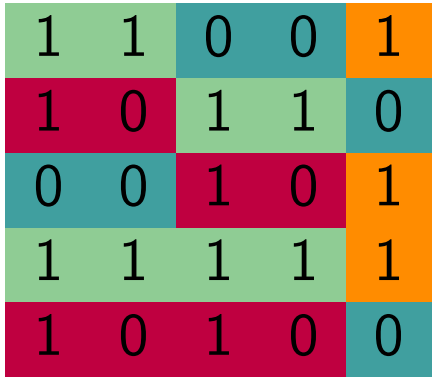
Instantiation



Patterns ('code table')

A practical algorithm

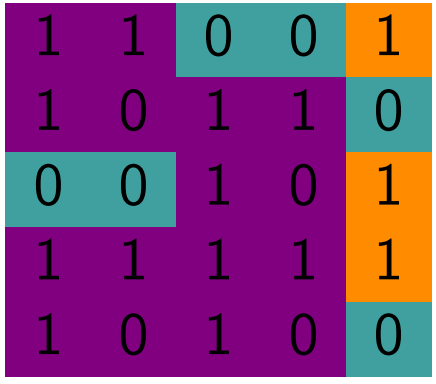
Instantiation



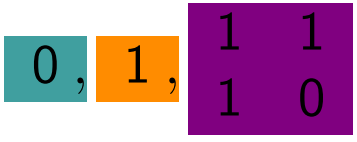
Patterns ('code table')

A practical algorithm

Instantiation



Patterns ('code table')

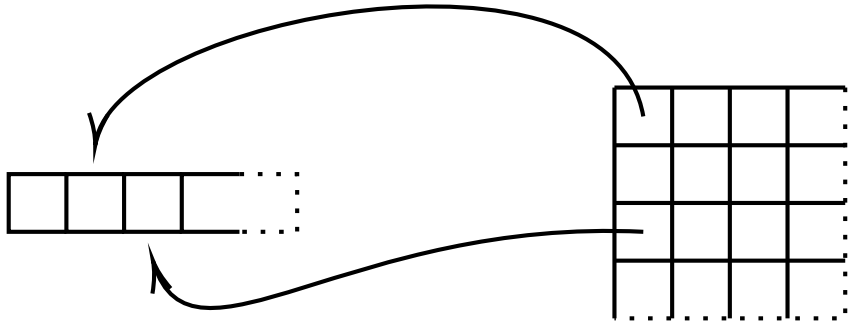


"Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. (...)" - Rob Pike

The instantiation matrix is central to the whole algorithm. How to implement it in actual code?

InstanceVector

InstanceMatrix



Contiguous memory

Sparse matrix

Datastructures (2)

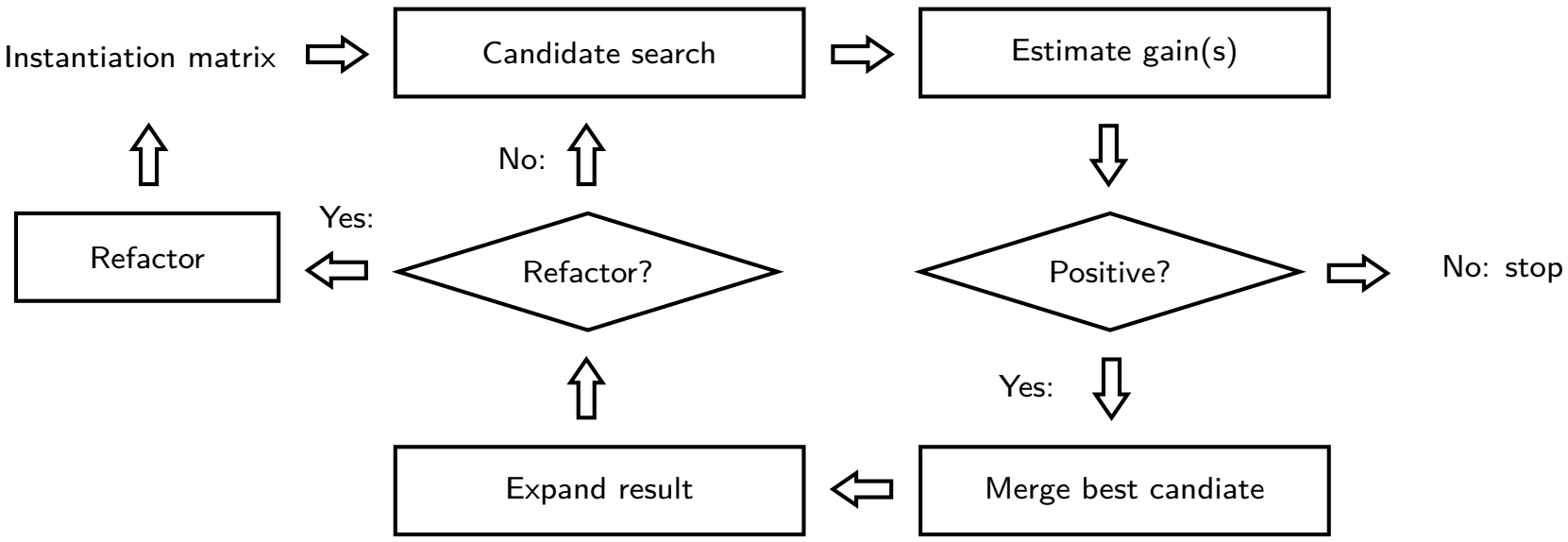
Although there is some redundant storage, we combine the best of both storage types:

InstanceVector Constant-time access by memory index, fast traversing in lexicographic order **if sorted**.

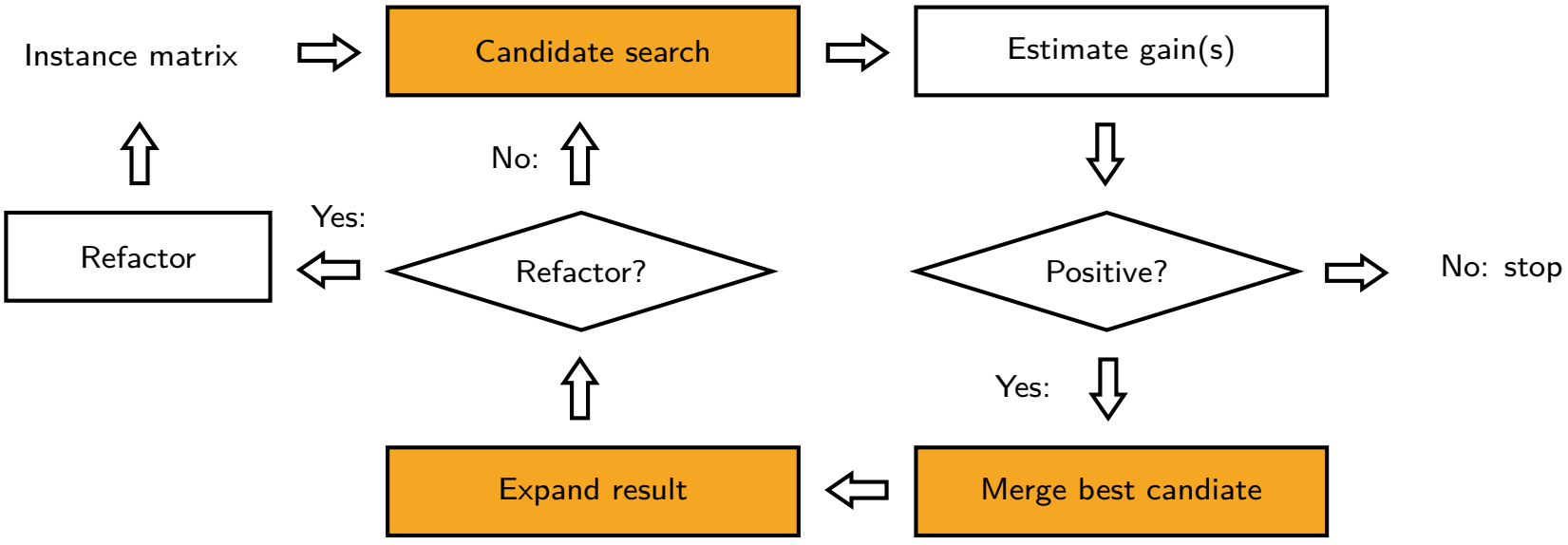
InstanceMatrix Constant-time access by coordinate.

Need to manually manage the memory of the InstanceVector: adding/deleting/sorting changes indices and thus InstanceMatrix needs to be (completely) rebuild as well. Hence the need for **refactoring**

A practical algorithm (2)

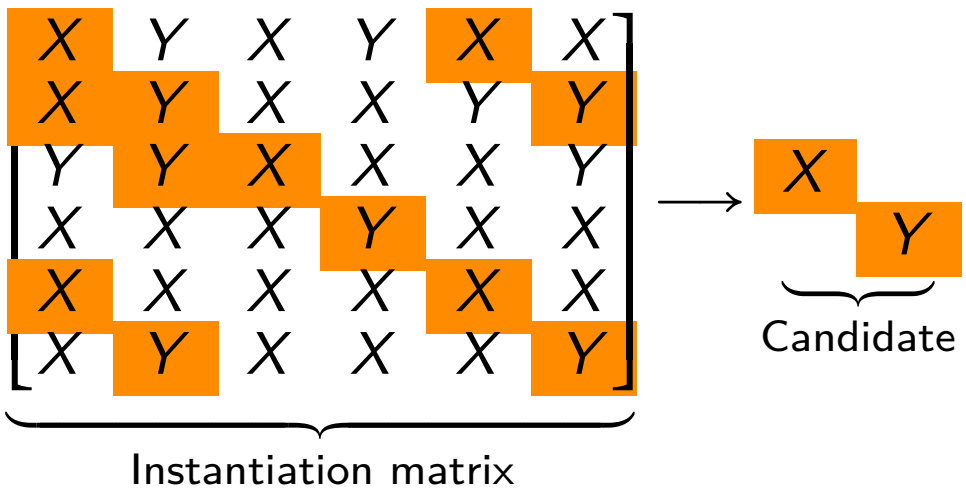


A practical algorithm (2)



Candidate search

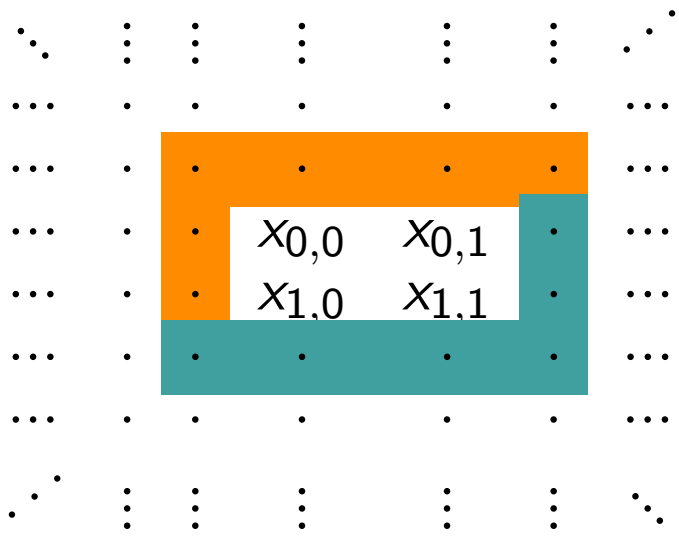
A candidate can be defined as a tuple $< X, Y, \delta >$: two patterns and a relative offset between them. We derive a list of candidates from the instantiation matrix and count their **usage**.



Limitation: we only pick instances whose *peripheries* overlap...

Peripheries

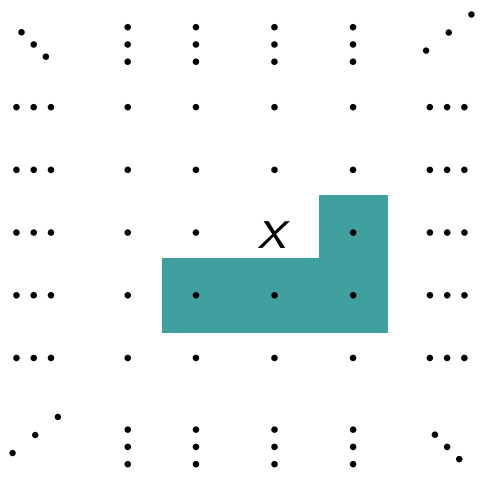
Along with its elements, we store each pattern's **periphery**:



The periphery elements are used to look in the neighbourhood of an instance.

Peripheries (2)

During candidate search, we are only looking in the **posterior periphery**. Consider the first iteration (only singletons):



Given a total number of unique elements n, we can produce up to $4n^2$ candidates (phew!). So for example, our noise image could give us $4(256)^2 = 262144$ candidates in the first iteration alone...

Counting candidates is not so simple, however.

How many times do you see $\begin{matrix} 1 \\ & 1 \end{matrix}$

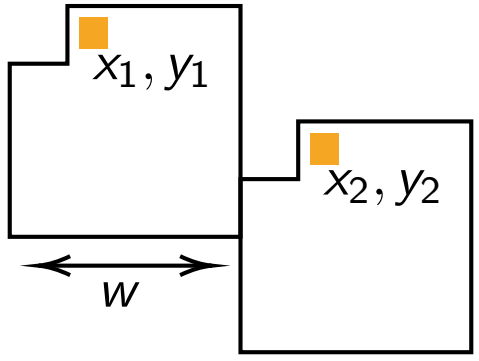
$$\begin{matrix}
\ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \iddots \\
\cdots & 1 & \cdot & \cdot & \cdot & \cdot & \cdots \\
\cdots & \cdot & 1 & \cdot & \cdot & \cdot & \cdots \\
\cdots & \cdot & \cdot & 1 & \cdot & \cdot & \cdots \\
\cdots & \cdot & \cdot & \cdot & 1 & \cdot & \cdots \\
\cdots & \cdot & \cdot & \cdot & \cdot & 1 & \cdots \\
\iddots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{matrix}$$

Four can be counted, but at most two pairs can be merged.

Solution: storing per-instance **overlap coefficients**.

Overlap (2)

Computing the overlap coefficient c.



$$c = w + x_2 - x_1 + (y_2 - y_1)(2w + 1)$$

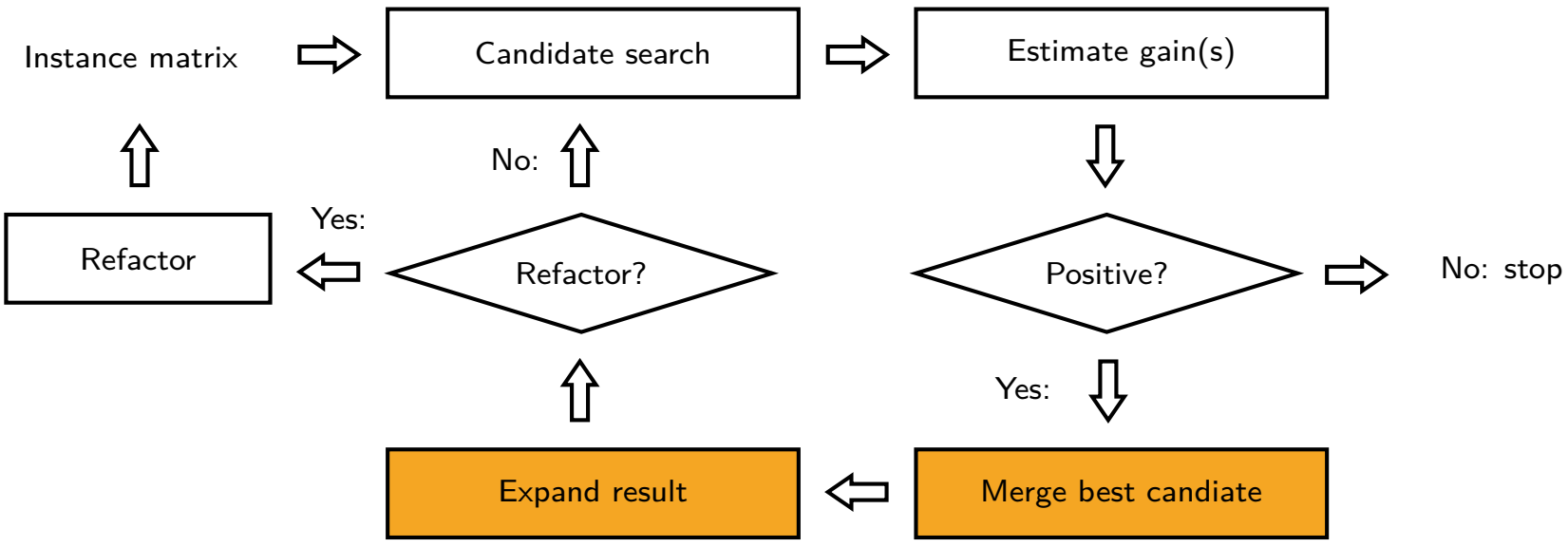
Overlap (3)

To each instance $\bar{X}$ we add an **overlap bitvector** $V()$. The candidate search algorithm now looks like this.

We visit every instance $\bar{X} \in \bar{H}$ in lexicographical order. Then for each $\bar{X}$ we look at all instances $\bar{Y}$ that we can reach from the posterior periphery of X:

1. if $X \neq Y$, goto 6.
2. With the offset between $\bar{X}$ and $\bar{Y}$ we obtain candidate $< X, Y, \delta >$:
3. Compute the overlap coefficient c from δ
4. If $\bar{X}$ has $V(c)$ set to 1, skip this candidate
5. Otherwise set $V(c)$ on $\bar{Y}$ to 1
6. Increment the usage of $< X, Y, \delta >$ by one

A practical algorithm (3)



Merging candidates

Assume that the pattern we are trying to find contains N elements, it takes ...

- At most $N-1$ merges,
- At least $\log_2(N)$ merges ...

... to construct it from singletons.

Unfortunately, in practice we get $N-1$ rather than $\log_2(N)$.

Flood-fill

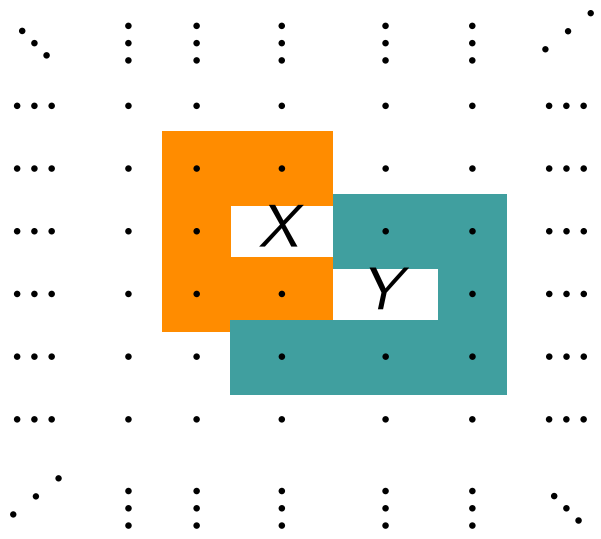
We can try to approach $\log_2(N)$ by implementing a local search. Say we merge candidate $< X, Y, \delta >$ into pattern Z, now for all resulting instances $\bar{Z}_i \in \bar{Z}_0, \bar{Z}_1, \ldots \bar{Z}_{m-1}$:

$$W = \ominus(\bar{H}_{\delta_0+c}) \iff \forall \bar{Z}_i \cdot \ominus(\bar{H}_{\delta_i+c}) = W$$

Where δ_i is the offset of $\bar{Z}_i$ and c is an element from the periphery of Z. If such W exists, replace all instances $\bar{Z}$ with the respective unions $\bar{Z} + \bar{W}$.

Flood-fill (2)

So when merging X and Y below, we look at the *resulting* periphery to see what adjacent pattern(s) *all instances* have in common.



We simply add all resulting matches, recursively. This result in a BFS, or 'flood fill'.

Merging candidates (2)

Candidate search is expensive, so we try to merge more candidates per iteration.

We keep a set of used patterns P_{used} and initialize it to $\varnothing$. Now for each candidate $< X, Y, \delta >$ in the list of candidates:

- If either $X \in P_{used}$ or $Y \in P_{used}$, skip
- Otherwise, merge the candidate
- Add X and Y to P_{used}

This is indeed a different heuristic that leads to slightly different results.

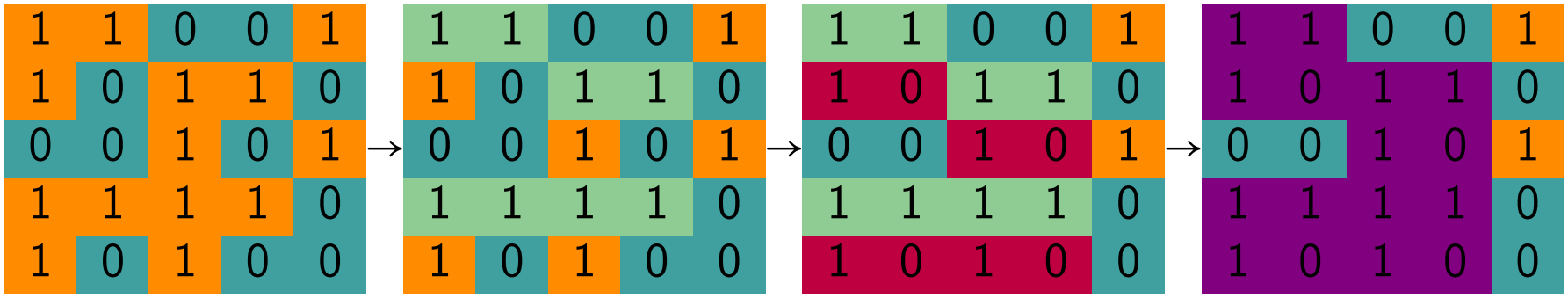
Pre-processing

VOUW utilized two simple data pre-processing steps.

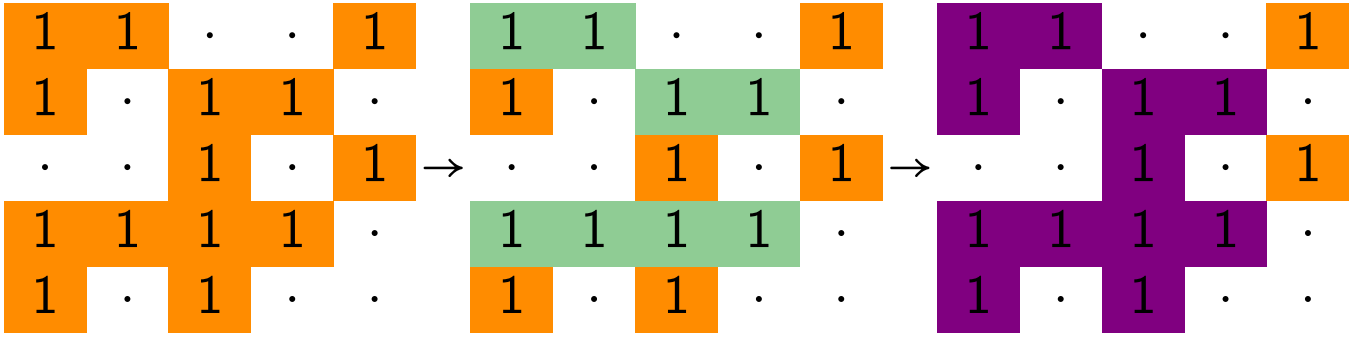
- Quantization: limit the number of singletons to begin with
- 'Tabu' patterns: exclude very prevalent singletons

Both steps have a profound effect on the results and there is no 'one size fits all'.

Tabu patterns



With '0' removed from the matrix



Problem classes

Not all matrices and patterns are equally difficult for VOUW. They can roughly be divided in five classes.

Class 1a Exact duplicates in a sparse matrix.

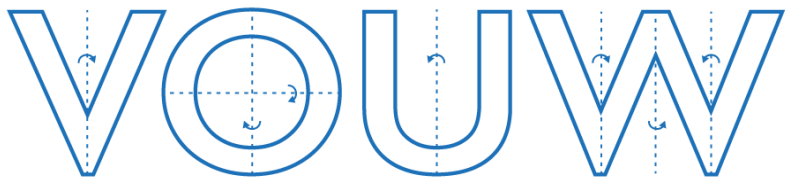
Class 1b Exact duplicates in (differently distributed) noise.

Class 1c Exact duplicates in noise with equal distribution.

Class 2 Approximate duplicates

Class 3 Transformed duplicates

Future goals



- Implement at least the ability to solve the ‘class 2’ problem.
- Rethink the encoding scheme, it is still not good enough ;-)
- Finish writing the paper.

Thank you for your attention!