Figure 1: VOUW logo

# VOUW Project Journal

## Status quo up to February 2019 (updated 24-2-2019)

In order to assess uncoming modifications, I decided to benchmark and report on the current sitution first. I will briefly decribe the components of the algorithm with their current implementation decisions.

- Candidate search. Lots of effort went into making this fast. Currently, we search for pairs *p1* and *p2* occuring in the instance list. Each pair is hashed so that we can quickly count multiple occurences as we go through the list. The main output of the candidate search is for all pairs *p1,p2* their current usage. One big problem is that if *p1=p2*, we can get all kinds of weird artifacts that lead to double counting. To combat this (and have exact gain computation) each time we count identical patterns, I compute their spatial configuration in what I call an *overlap coefficient*. This coeffecient is stored efficiently in a bitvector of the instance belonging to *p2*. When we later detect the same configuration in some instance belonging to a *p1*, we know we have already seen this thing shifted by one. Thus we ignore this occurence.
- Gain computation. In this step we compute gain for all candidates depending on their usage. Currently the computation is exact, so we need to do a lot of steps to get it right, because merging two patterns changes the entire codetable and all its codelengths. It is algebraically optimized such that little computations are needed and is therefore generally the fastest step.
- Merge the candidate. Here we simply merge the two patterns from the best candidate and replace all instances of the separate patterns with the same configuration, with the newly created pattern. This operation can account for a third of the computation time, depending on the usage of the new pattern. Effort decreases exponentially near the end because the instance matrix becomes much smaller.
- Pruning. Pruning can either be the pruning of zero-use patterns or the decomposition of used patterns. The latter is very complex and although it takes little time, it has not (yet) proven to be very usefull.

### Shortcomings

The current approach is reasonably fast, but has several shortcomings. The most important are: (1) the encoded result during the process is not the same as when one would encode the input matrix with the same codetable from scratch, (2) there is no noise-rebustness and (3) an unwanted artifact occurs that leads to sub-optimal encodings that I call *small-pair preference.*

### Small-pair preference

This artifact is a result of taking shortcuts when merging the best candidate patterns. Say we have two strings of elements over the alphabet *1,2*:

```
1 2 1 2 1 2
1 2 1 2 1
```

We ignore the horizontal relation for a moment here. One would expect to find `1 2 1 2 1` twice along a lone `2` in the first string. However, we will in fact find five patterns `1 2` with a lone `1` in the second string. We could solve this problem by re-encoding the entire input matrix at every step. However, this would mean our gain computation is off as well as slow down each step considerably.

### Benchmark

These tests provide some insight into the current performance of the algorithm. These tests are not representative of any real-world problem nor are they particulary robust. I do hope they give some baseline to compare future solutions to.

| Test set | Ratio | Patterns | Iterations | Time | Subjective quality |
|---|---:|---:|---:|---:|---|
| triangles32_32 | 43.14% | 5 | 10 | 3 ms | Excellent |
| sonnet18 | 17.87% | 41 | 258 | 1857 ms | Good |
| smileys512 | 8.586% | 21 | 943 | 21505 ms | Mediocre |
| rule73rand | 39.64% | 30 | 85 | 193 ms | Excellent |
| shapes40 | 31.82% | 10 | 106 | 106 ms | Good |
| checkboard256 | 2.542% | 3 | 16 | 401 ms | Mediocre |
| noise512 | - | - | - | - | Crash |
| noisytriangles64 | 97.2% | 14 | 7 | 23 ms | Mediocre |
| smallpair16 | 71.12% | 6 | 9 | 2 ms | Mediocre |
| rulers32 | 93.64% | 8 | 8 | 4 ms | Mediocre |

All tests were performed with 8 quantization levels and 'show progress' unchecked for unbiased timings. Note that the last three tests were especially crafted to make the current algorithm perform at its worst.