MORK-MINER

Ben Goertzel

July 21, 2025

Abstract

We present a low-level, MORK-native pattern mining approach that leverages the hierarchical PathMap index for extremely efficient discovery of frequent substructures. Two examples are briefly elaborated: wavelet-transform hierarchies and factor graphs stored in MORK. In the wavelet case, hidden-state vectors are decomposed via discrete wavelet transform and each coefficient is stored under a key path encoding layer, time step, scale, and subband. Seed patterns are extracted by scanning local subtrees, grown by prefix-proximity across adjacent scales, and support counts are maintained in-place via lightweight counters. In the factor-graph case, edges between factors and variables are encoded as key paths so that neighborhoods can be enumerated by prefix scans and pattern supports tracked at each prefix. By performing seed generation, pattern expansion, and support counting entirely within MORK's PathMap, we achieve likely orders-of-magnitude speedups over generic AtomSpace mining while preserving the same conceptual pipeline of frequent pattern discovery.

Contents

1 Introduction

We describe a pattern miner that runs entirely inside MORK's PathMap index, exploiting prefix-tree locality to extract seeds, grow patterns, and count supports without external graph matching.

The key idea is simply locality in the PathMap. Instead of treating the AtomSpace as a flat graph, we leverage MORK's PathMap index to organize nodes into prefix-tree neighborhoods. Any two entries whose key paths share a long common prefix live nearby in PathMap and often correspond to semantically related data (for example adjacent wavelet coefficients or adjacent factor-graph nodes).

Here we outline the basic idea and then sketch two applications: to wavelet-transform and factor-graph Atomspaces. In each of these cases, the way data is stored in the MORK PathMap closely reflects the actual semantics of the data.

2 MORK-Native Pattern Mining Architecture

This section describes a low-level pattern miner that runs entirely within MORK's PathMap index, exploiting its hierarchical key structure and in-place counters to extract, grow, and score patterns without external graph matching.

The miner treats each PathMap key as a node in a prefix tree. Nodes that share long key prefixes are considered "nearby" and likely to participate in the same pattern. By operating on these prefix neighborhoods, we avoid costly global searches and subgraph isomorphism checks.

2.1 Seed Pattern Extraction

For each new data insertion, the miner identifies seed patterns by scanning a fixed-depth subtree under a selected key prefix. A seed pattern may consist of any small combination of sibling keys (for example two adjacent subkeys under the same parent). This scan is implemented as a single subtree traversal in PathMap.

2.2 Pattern Growth via Prefix Proximity

To expand a seed into a larger pattern, the miner locates additional keys whose paths share the same higher-level prefix and whose depth differs from the seed by at most one level. All candidate extensions are found by a single prefix scan that returns the full set of nearby keys. New pattern clauses are formed by combining the seed with each candidate.

2.3 In-Place Support Counting

Whenever a new data item is stored, the miner increments an integer counter at each prefix node along its key path. The support of any pattern (represented by a prefix path) is then obtained by reading the counter at its prefix node. No separate matching or counting pass is required.

2.4 Pipeline Summary

The complete miner consists of three stages, all implemented as PathMap operations:

1. Extract seed patterns by scanning a subtree under a given prefix.
2. Grow each seed by combining it with nearby keys found via prefix proximity.
3. Score each pattern by reading the in-place counter at its prefix node.

In this way we obtain a pattern miner that runs entirely inside MORK's PathMap in time proportional to $\text{branch\_factor} \times \text{pattern\_size}$ per pattern instead of costly subgraph matching. The conceptual architecture of frequent or surprising patterns up to N-clauses is preserved while achieving orders-of-magnitude speedups by exploiting the built-in hierarchical index and locality of MORK.

3 Mining Patterns in Wavelet-Transform Hierarchies

We describe application of this MORK-MINER to MORK Atomspaces containing wavelet transforms extracted from neural net states.

At training time, each hidden-state vector is transformed via a discrete wavelet transform into a tree of coefficient subvectors and stored under PathMap keys of the form:

```
layer_l/step_t/level_k/subband_j
```

Here `layer_l` tags the transformer block, `step_t` the token index, `level_k` the wavelet scale, and `subband_j` the subband name. To extract local seed patterns, scan the subtree under

```
layer_l/step_t/level_k
```

and form small combinations of adjacent subbands (for example `low` and `high1`). To grow these seeds, perform prefix scans under the same

```
layer_l/step_t
```

prefix but restrict to entries whose level index is within plus or minus one of the seed level. Support for any prefix pattern is maintained by a simple integer counter stored at that prefix node and incremented whenever a new wavelet snapshot is inserted.

4 Mining Patterns in Factor Graphs

We describe application of this MORK-MINER to MORK Atomspaces containing PLN factor graphs.

Encode factor-graph edges as PathMap entries:

```
graphX/factor/FID/var_VID
graphX/var/VID/factor_FID
```

For each factor `FID`, seed patterns are its immediate variable neighbors, obtained by listing subkeys under

```
graphX/factor/FID
```

To grow patterns that involve pairs of factors sharing a variable, scan the subtree under

```
graphX/var/VID
```

and collect frequent factor pairs. Support counts for these patterns are maintained at each prefix when new edges are inserted.