BuildHON$^2$: A Scalable Method for Growing Higher Order Networks

STEVEN KRIEG

SCALABLE GRAPH ALGORITHMS

UNIVERSITY OF NOTRE DAME
Presentation Overview

1. Background & Motivation
2. Design & Implementation
3. Evaluation
4. Conclusions & Future Work
Higher Order Networks

How do we represent big data as a network, while accurately preserving dependencies?

Figure from [2]
Higher Order Networks

Raw event sequence data

First-order network

Count number of pairwise interactions as edge weights

Extract higher-order dependencies from raw event sequences

Higher-order dependencies

Construct HON based on the extracted rules

Higher-order network

Figure from [2]
Sequential Implementation

Published version written in Python [3]:

1. Extracts all 2 node sequences
2. Checks to see if adding a prefix to a sequence “significantly” changes the movement confidence
3. If so, preserve the 3 node sequence as a rule and try extending further
4. Repeat until convergence or max order is reached
Sequential Implementation

My version written in C++:

- Scaled horribly due to repeated searches of sequence data
- Python version uses an “indexing cache” to store the locations of sequences
  - Better performance
  - Large memory footprint
Enter Fp-Trees [7]

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{a, b}</td>
</tr>
<tr>
<td>2</td>
<td>{b, c, d}</td>
</tr>
<tr>
<td>3</td>
<td>{a, c, d, e}</td>
</tr>
<tr>
<td>4</td>
<td>{a, d, e}</td>
</tr>
<tr>
<td>5</td>
<td>{a, b, c}</td>
</tr>
<tr>
<td>6</td>
<td>{a, b, c, d}</td>
</tr>
<tr>
<td>7</td>
<td>{a}</td>
</tr>
<tr>
<td>8</td>
<td>{a, b, c}</td>
</tr>
<tr>
<td>9</td>
<td>{a, b, d}</td>
</tr>
<tr>
<td>10</td>
<td>{b, c, e}</td>
</tr>
</tbody>
</table>

Figure from [9]
## Fp-Trees

<table>
<thead>
<tr>
<th>Algorithmic Component</th>
<th>Helpful to HON?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Growth*</td>
<td>✋</td>
</tr>
<tr>
<td>Extraction**</td>
<td>✋</td>
</tr>
</tbody>
</table>

* No way to efficiently find rules of higher/lower order
** Extraction does not account for sequential ordering
Design & Implementation

1. Background & Motivation
2. Design & Implementation
3. Evaluation
4. Conclusions & Future Work
Modified Fp-Tree

<table>
<thead>
<tr>
<th>TID</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{1, 2, 3, 1, 2, 3}</td>
</tr>
<tr>
<td>2</td>
<td>{1, 2, 3, 1, 2, 3}</td>
</tr>
<tr>
<td>3</td>
<td>{1, 2, 4, 1, 2, 4}</td>
</tr>
<tr>
<td>4</td>
<td>{1, 2, 3, 1, 2, 3}</td>
</tr>
<tr>
<td>5</td>
<td>{2, 3, 4, 1, 2, 4}</td>
</tr>
<tr>
<td>6</td>
<td>{5, 2, 4, 5, 2, 4}</td>
</tr>
<tr>
<td>7</td>
<td>{5, 2, 4, 5, 2, 4}</td>
</tr>
<tr>
<td>8</td>
<td>{5, 2, 4, 5, 2, 3}</td>
</tr>
</tbody>
</table>

![Diagram of Modified Fp-Tree](image-url)
2D Header Table + Cousin Links
Pruning the Tree

Order 1

Order 2

Order 3

Candidate rule: 1 -> 2 -> 4
Pruning the Tree

Candidate rule: (1 -> 2 -> 4)
Corresponding lower order rule: (2 -> 4)
Pruning the Tree

Candidate rule: (1 → 2 → 4)
Corresponding lower order rule: (2 → 4)
if conf (1 → 2 → 4) !- conf (2 → 4):
prune (1 → 2 → 4)
Pruning the Tree
Pruning the Tree

Candidate rule: (1 -> 2 -> 4)
Corresponding lower order rule: (2 -> 4)
else: prune (2 -> 4)
Pruned Tree

Order 1
1
2 3 4 5

Order 2
2 4 3 4 1 1 5 2

Order 3
3 5 2 3 4 4

DESIGN & IMPLEMENTATION
Driver Code

```cpp
#include "hon_includes.h"

int main(int argc, char** argv) {
    std::string inf_path = (argc > 1) ? argv[1] : DEF_INF_PATH;
    int num_seq_prefixes = argc > 2 ? std::stoi(argv[2]) : DEF_NUM_SEQ_PREFIXES;
    int max_order = argc > 3 ? std::stoi(argv[3]) : DEF_MAX_ORDER;
    bool verbose = argc > 4 ? std::stoi(argv[4]) : DEF_VERBOSE;

    if (verbose) std::cout << "Building tree from file: " << inf_path << std::endl;
    FpTree fpt = FpTree(inf_path, num_seq_prefixes, max_order, verbose);
    if (verbose) std::cout << "Tree build complete!" << std::endl;
    fpt.prune();
    fpt.dump_rules();
    return 0;
}
## Complexity Analysis

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Growing</td>
<td>$O(k \times m)$</td>
<td>$O(n^m)$ - dense</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(n \times m)$ - sparse</td>
</tr>
<tr>
<td>Pruning</td>
<td>$O(n^m)$ - dense</td>
<td>$O(n \times m)$ - sparse</td>
</tr>
</tbody>
</table>

$k =$ total length of data set  
$m =$ max order  
$n =$ number of unique nodes  
Density/sparsity refers to the number of first-order (pairwise) relationships between nodes.
Implementation

- C++11 using only the standard library (lots of unordered maps - possible reallocation issue?)
- Two custom classes:
  - FpTree: ~300 lines of code
  - FpNode.cpp: ~100 lines of code
- ~480 total lines of code; remaining 80 are driver / header definitions
Evaluation

1. Background & Motivation
2. Design & Implementation
3. Evaluation
4. Conclusions & Future Work
## Synthetic Data Sets

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Number of Records</th>
<th>Record Length</th>
<th>Total Size</th>
<th>Unique Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SyntheticFull</td>
<td>10,000</td>
<td>100</td>
<td>1,000,000</td>
<td>100</td>
</tr>
<tr>
<td>SyntheticLarge</td>
<td>100,000</td>
<td>100</td>
<td>10,000,000</td>
<td>1,000</td>
</tr>
<tr>
<td>SyntheticGiant</td>
<td>1,000,000</td>
<td>100</td>
<td>100,000,000</td>
<td>1,000</td>
</tr>
<tr>
<td>SyntheticBalrog</td>
<td>10,000,000</td>
<td>100</td>
<td>1,000,000,000</td>
<td>1,000</td>
</tr>
</tbody>
</table>
### Scalability Experiment

<table>
<thead>
<tr>
<th>Application</th>
<th>Data Collection</th>
<th>Runtime Parameters</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>HON² (CHON)</td>
<td>Base Synthetic</td>
<td>max_order = 5</td>
<td>40 (10 runs * 4 data sets)</td>
</tr>
<tr>
<td>HON+ (Python)</td>
<td>Base Synthetic</td>
<td>max_order = 99</td>
<td>40 (10 runs * 4 data sets)</td>
</tr>
<tr>
<td>HON² (CHON)</td>
<td>Extended Synthetic</td>
<td>max_order = 5</td>
<td>40 (10 runs * 4 data sets)</td>
</tr>
<tr>
<td>HON² (CHON)</td>
<td>SyntheticGiant only</td>
<td>max_order = [3, 4, 5, 6, 7, 8, 9]</td>
<td>70 (10 runs * 7 parameters)</td>
</tr>
</tbody>
</table>

**Key metrics:**
- Runtime (wallclock)
- Max vmem usage
Results: Increasing Total Size

EVALUATION
Results: Increasing Total Size

EVALUATION

CRC shenaniganry?
Results: Baseline Comparison
# Synthetic Data Set Extensions

<table>
<thead>
<tr>
<th>Base Data Set</th>
<th>Number of Records</th>
<th>Record Length</th>
<th>Total Size</th>
<th>Unique Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SyntheticGiant</td>
<td>10,000</td>
<td>100</td>
<td>1,000,000</td>
<td>10,000 (x10)</td>
</tr>
<tr>
<td>SyntheticGiant</td>
<td>100,000</td>
<td>100</td>
<td>10,000,000</td>
<td>100,000 (x100)</td>
</tr>
<tr>
<td>SyntheticBalrog</td>
<td>1,000,000</td>
<td>10 (x0.1)</td>
<td>100,000,000</td>
<td>1,000</td>
</tr>
<tr>
<td>SyntheticLarge</td>
<td>100,000</td>
<td>1000 (x10)</td>
<td>100,000,000</td>
<td>1,000</td>
</tr>
</tbody>
</table>
Results: Varying Sequence Length
Results: Varying Unique Count
Results: Varying Max Order

EVALUATION
Real-world* Data

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Number of Records</th>
<th>Record Length</th>
<th>Total Size</th>
<th>Unique Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018 NY Taxi Data</td>
<td>4,306,477</td>
<td>52</td>
<td>223,387,351</td>
<td>266</td>
</tr>
</tbody>
</table>

Each item in the sequence is a “Taxi Zone” in which a trip begins or ends ([http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml](http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml))

* I had to artificially link sequences :-(

* I had to artificially link sequences :-(
Results: NY Taxi Data
Conclusions & Future Work

1. Background & Motivation
2. Design & Implementation
3. Evaluation
4. Conclusions & Future Work
Conclusions

Scalability achieved 🎉
Future Work

- Fix runtime wall (collisions?)
- Fix pruning accuracy (currently over-prunes)
- Vary graph density
- Parallelize
Future Work

- Fix runtime wall (collisions?)
- Fix pruning accuracy (currently over-prunes)
- Vary graph density
- Parallelize
References