Chapter 1

BuildHON^2: A Scalable Higher-Order Network

Contributed by Steven Krieg

[A note for reviewers of draft 2 (sequential implementation): I did not have access to the draft 1 feedback before submitting this version. If you reviewed draft 1 and provided feedback, please know that it will be considered for the next iteration, but those suggestions may not yet be included.]

1.1 Introduction

Networks are used to represent and analyze a variety of problems related to big data. However, some data is too complex to be accurately reprsented by a traditional first-order network. In the case of a first-order network, sequences of data are analyzed as Markov chains. For many applications, including transportation networks and anomaly detection, accurate analysis must take into account a series of events, not just one pair. A higher-order network (HON) representation is a creative solution to this problem that has demonstrated compelling increases in representative accuracy [7]. However, the trade-off for increased accuracy is increased network size and computational cost. In some cases, this trade-off may make HON an unattractive option. In this project, I will seek to provide a scalable implementation of a higher-order network. If successful, the implementation will provide HON's key representational benefits while minimizing their costs.

1.2 The Problem as a Graph

HON deals with data representing sequential interactions with multiple-levels of dependencies. Figure 1.1 illustrates a simple example.

Our data contains 16 sequences: 4 * (A, M), 4 * (B, M), 4 * (M, X), and 4 * (M, Y). Consider that our task is to determine the probability that a random walker beginning from node A will reach node X. A first-order network counts the number of pairwise interactions between all nodes and represents the probability of the interaction between both nodes as the edge weight. The random walker will go from A to M with 100% probability. From M, the walker has a 50% chance of moving to X and a 50% chance of moving to Y. From the visualization, we can see that this is not an accurate result. When A is the source, our data shows 75% termination at X. We will have the same representation problem when vertex B is the source. However, a first-order network

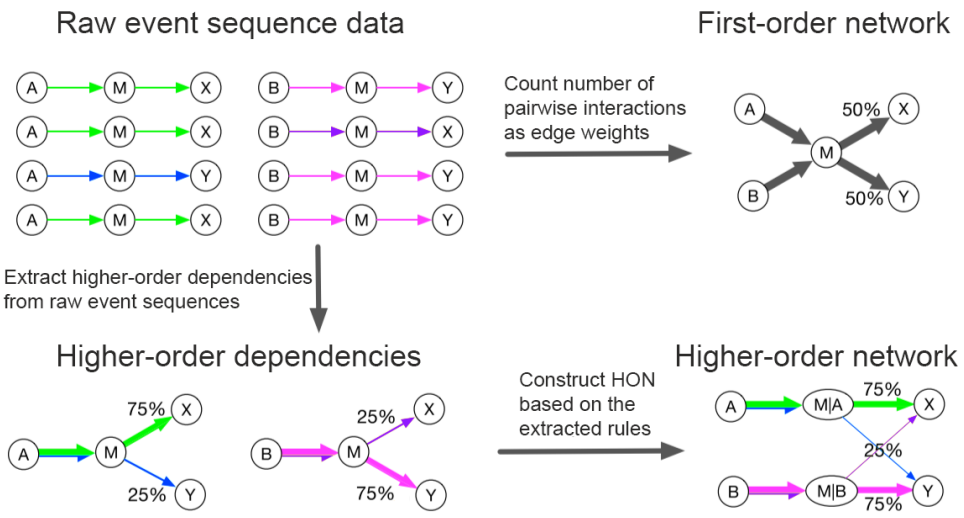

Figure 1.1: The HON Idea [5]

has no mechanism for chaining these sequences together, and thus once our walker reaches M it "forgets" its source.

A smarter algorithm can be a solution to this problem: for example, our random walker could be trained on rules. This could be costly at many levels. HON tackles the problem from another angle by choosing a better representation for the network itself. In the example above, HON responds to a dependency by splitting M into 2 nodes: M—A and M—B. The benefit from this solution is that our random walker does not need to be any smarter; the network representation solve the problem for it. This is a chief goal of HON: making the network structure more representative so existing tools can be utilized without modification. BuildHON+, the algorithm responsible for making these network modifications, will be discussed below.

1.3 Some Realistic Data Sets

HON processes data that can be represented as a weighted digraph. The two datasets utilized in previous implementations are global shipping routes traversed during several months in 2012 (real-world, 31,000 edges) [7], and synthetic clickstream data used for anomaly detection [6]. Both datasets are publicly available with the HON solution. I plan to begin with these data sets, and potentially incorporate others as the implementation scales [8].

1.4 BuildHON^2-A Key Graph Kernel

In this project I seek to develop BuildHON^2, a scalable version of the BuildHON network rewiring algorithm. BuildHON includes 2 major phases: rule extraction and network rewiring [7]. The second iteration, BuildHON+, significantly improved the runtime of the algorithm by implementing a lazy version of rule extraction [6]. However, BuildHON+ still has complexity $\theta(N)(2R_1 + 3R_2 + ...(i+1)R_i)$. While not expontential, as the number of nodes N and rules (dependencies) i increases in very large and comlex data sets, BuildHON+ will reach a point of unusability. In such cases, researchers must make a difficult trade-off between performance and the superior accuracy of a HON representation. The goal of BuildHON^{2}is to address this problem such the benefits of building a

HON will be well worth the costs.

A couple ideas are key to a more scalable implementation. First is implementation efficiency. BuildHON+ is currently implemented in Python, but an implementation in a more efficient paradigm will be more usable on large data sets. Second is parallelism. BuildHON+ could reap the distributed benefits of a distributed graph platform like Giraph [1] or Parallel Boost [3]. A more streaming-friendly graph platform like STINGER [2] may also enhance BuildHON's capabilities.

Note that I am not primarily seeking to improve the theoretical efficiency of the BuildHon+ algorithm. My focus is instead on the performance of concrete implementations.

1.5 Prior and Related Work

Graph researchers have long acknowledged the limitations of first-order networks. Though much research has sought to overcome these limitations, most of them focus on algorithms rather than representation. Some representation solutions have been proposed, such as a fixed second-order network. However, a fixed second-order introduces many unnecessary nodes and edges. Because the order is fixed to two, even interactions that are represented accurately with only a single order are forced to include a second. This may be appropriate for some networks, but most real-world networks are scale-free and thus the majority of higher orders are centralized to hubs [7]. This means dependencies for nodes near hubs will tend to be underrepresented and depedencies for nodes far from hubs will be overrepresented. Accuracy will not suffer but the efficiency of graph computations will.

BuildHON+, in response to this problem, is designed to be flexible. It only rewires nodes and edges where dependencies are found, and can specify an arbitrary maximum for levels of dependency. Thus it is almost always more accurate and more efficient than fixed-order networks when applied to real-world networks [7].

Many solutions have been proposed for scalable graph computations: new algorithms, architectures, engines, and more. These solutions may be very helpful in conjunction with BuildHON^2: if the data set is large enough to require a distributed solution for building a HON, it will likely require a distributed solution for computations on the HON. But the BuildHON^{2}implementation itself will focus on the actual generation of the network, which can then be processed as any other graph.

1.6 A Sequential Algorithm

The state-of-the-art sequential algorithm is detailed in Figure 1.2.

Algorithm 1 HON+ rule extraction algorithm. Given the raw sequential data T, extracts arbitrarily high orders of dependencies, and output the dependency rules R. Optional parameters include $MaxOrder$, $MinSupport$, and $ThresholdMultiplier$

```
1: define global C as nested counter
2: define global D,R as nested dictionary
3: define global SourceToExtSource, StartingPoints as dictionary
4:
5: function ExtractRules(T, [MaxOrder, MinSupport, ThresholdMultiplier = 1])
6:     global MaxOrder, MinSupport, Aggresiveness
7:     BuildFirstOrderObservations(T)
8:     BuildFirstOrderDistributions(T)
9:     GenerateAllRules(MaxOrder, T)
10:
11: function BuildFirstOrderObservations(T)
12:     for t in T do
13:         for (Source, Target) in t do
14:             C[Source][Target] += 1
15:             IC.add(Source)
16:
17: function BuildFirstOrderDistributions(T)
18:     for Source in C do
19:         for Target in C[Source] do
20:             if C[Source][Target] < MinSupport then
21:                 C[Source][Target] = 0
22:             for Target in C[Source] do
23:                 if thenC[Source][Target] > 0
24:                     D[Source][Target] = C[Source][Target]/(∑ C[Source][∗])
25:
26: function GenerateAllRules(MaxOrder, T)
27:     for Source in D do
28:         AddToRules(Source)
29:         ExtendRule(Source, Source, 1, T)
30:
31: function KLDThreshold(NewOrder,ExtSource)
32:     return ThresholdMultiplier × NewOrder/log2(1 + ∑ C[ExtSource][∗])
33: function ExtendRule(Valid, Curr, order, T)
34:     if Order ≤ MaxOrder then
35:         AddToRules(Source)
36:     else
37:         Distr = D[Valid]
38:         if −log2(min(Distr[∗].vals)) < KLDThreshold(order + 1), Curr then
39:             AddToRules(Valid)
40:         else
41:             NewOrder = order + 1
42:             Extended = ExtendSource(Curr)
43:             if Extended = ∅ then
44:                 AddToRules(Valid)
45:             else
46:                 for ExtSource in Extended do
47:                     ExtDistr = D[ExtSource]
48:                     divergence = KLD(ExtDistr, Distr)
49:                     if divergence > KLDThreshold(NewOrder, ExtSource) then
50:                         ExtendRule(ExtSource, ExtSource, NewOrder, T)
51:                     else
52:                         ExtendRule(Valid, ExtSource, NewOrder, T)
```

Algorithm 1 *(continued)*

```
53: function AddToRules(Source):
54:     for order in [1..len(Source) + 1] do
55:         s = Source[0 : order]
56:         if not s in D or len(D[s]) == 0 then
57:             ExtendSource(s[1:])
58:         for t in C[s] do
59:             if C[s][t] > 0 then
60:                 R[s][t] = C[s][t]
61:
62: function ExtendSource(Curr)
63:     if Curr in SourceToExtSource then
64:         return SourceToExtSource[Curr]
65:     else
66:         ExtendObservation(Curr)
67:         if Curr in SourceToExtSource then
68:             return SourceToExtsource[Curr]
69:         else
70:             return ∅
71:
72: function ExtendObservation(Source)
73:     if length(Source) > 1 then
74:         if not Source[1 :] in ExtC or ExtC[Source] = ∅ then
75:             ExtendObservation(Source[1 :])
76:     order = length(Source)
77:     define ExtC as nested counter
78:     for Tindex, index in StartingPoints[Source] do
79:         if index − 1 ≤ 0 and index + order < length(T[Tindex]) then
80:             ExtSource = T[Tindex][index − 1 : index + order]
81:             ExtC[ExtSource][Target]+ = 1
82:             StartingPoints[ExtSource].add((Tindex, index−1))
83:     if ExtC = ∅ then
84:         return
85:     for S in ExtC do
86:         for t in ExtC[s] do
87:             if ExtC[s][t] < MinSupport then
88:                 ExtC[s][t] = 0
89:             C[s][t]+ = ExtC[s][t]
90:         CsSupport = ∑ ExtC[s][∗]
91:         for t in ExtC[s] do
92:             if ExtC[s][t] > 0 then
93:                 D[s][t] = ExtC[s][t]/CsSupport
94:                 SourceToExtSource[s[1 :]].add(s)
95:
96: function BuildSourceToExtSource(order)
97:     for source in D do
98:         if len(source) = order then
99:             if len(source) > 1 then
100:                 NewOrder = len(source)
101:                 for startingin[1..len(source)] do
102:                     curr = source[starting :]
103:                     if not curr in SourceToExtSource then
104:                         SourceToExtSource[curr] = ∅
105:                     if not NewOrder in SourceToExtSource[curr] then
106:                         SourceToExtSource[curr][NewOrder] = ∅
107:                     SourceToExtSource[curr][NewOrder].add(source)
```

Figure 1.2: The BuildHON+ Rule Exraction Algorithm [8]

1.7 A Reference Sequential Implementation

This simple implementation of the Rule Extraction portion of the BuildHon algorithm is written in C++ using only classes from the C Standard Library. The implementation includes several vectors and one unordered map (hash table) for data structures, and comprises just over 400 lines of code. The core of the driver code is shown in Code Segment 1.1.

Code Segment 1.1: CHONDriver.cpp

```
1  int main() {
2      cur_ord = 1;
3      seqs = get_raw_sequences();
4      first_order = build_observations(seqs, cur_ord);
5      rules.append(first_order);
6
7      while (rules.last != empty AND current_order < MAX_ORDER) {
8          next_cands = get_next_order_candidates(rules.last);
9          next_ord_obs = build_observations(seqs, cur_ord, next_cands);
10         next_rule = check_and_extend(rules.last, next_obs);
11         rules.append(next_rule);
12     }
13     return 0
14 }
```

This driver relies on three sub-functions: *get_next_order_candidates*, *build_observations*, and *check_and_extend*. My current plans for an enhanced implementation will involve replacing these sub-functions and rewriting the main driver, so I have decided not to include the code for them in this draft. I describe my future implementation plans below.

1.8 Sequential Scaling Results

I executed a small series of experiments to determine a baseline performance for the sequential implementation. Each iteration executed on a Ubuntu 18.04 virtual machine with 5GB of memory. Table 1.1 lists the average results of 30 total runs: 15 on the sequential C++ implementation and 15 on the published Python version. Of the 15 total iterations, 5 each were given data sets of size 1 million, 5 million, and 10 million sequential steps.

Number of Pairs	C++ Exec Time (s)	C++ # Rules	Python Exec Time (s)	Python # Rules
1m	22.32	440	26.01	212
5m	328.52	2,200	116.12	1160
10m	1321.26	4400	–	–

Table 1.1: Results from the sequential C++ implementation when compared with the published Python version [8]. Numbers displayed are the average of 5 iterations on different size sets of the synthetic web clickstream data. The Python implementation exceeded available memory when processing the largest data set.

On the smallest data set, the C++ version runs faster while generating more rules. However, the C++ version does not scale well with an increasing input size. I believe this is due to the data structures used in the simple implementation (mostly vectors for easy implementation and fast iteration). In fact, the C++ version's performance scales in accordance with what is predicted by

the theoretical performance: $\theta(L * N * \sum_{i=1}^{k}((i+1)R_i))$, where L is the number of transactions in the raw data, N is the number of unique nodes in the raw data, k is the order of the most complex rule, and R_i is the count of rules at order i. The summation becomes the biggest factor due to the need for repeated iterations through the sequential data to extract deeper rules. Python's lists and dictionaries allow for better scaling in terms of execution speed, but the implementation consumes much more physical memory and will not complete on the largest data set.

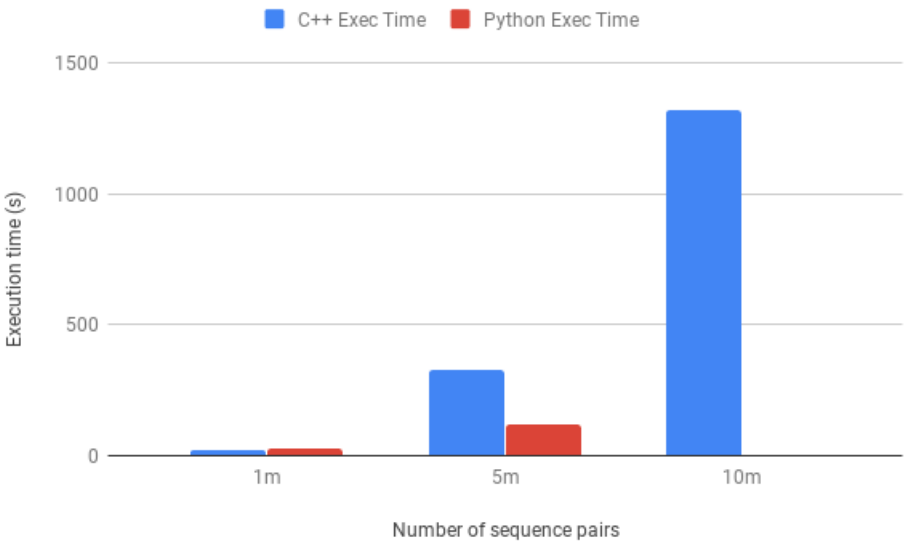

Figure 1.3: Visualization of sequential results listed in Table 1.1

The Python version includes a "minsupport" parameter which filters out some low-frequency rules, which is likely the cause of the discrepancies in number of rules. This likely causes a significant increase in execution time (much higher than a factor of 2), because each additional rule increases the likelihood that the maximum order of the rule set will increase. Every order 2 rule generated must be checked against each preceding node to see if adding a prefix will generate an order 3 rule. Rules continue to grow until further growth is no longer statistically significant, or the number of sequences falls below the "minsupport" parameter. I expect that adding this threshold would decrease the C++ runtime by an order of magnitude. However, this would still not guarantee satisfactory scaling results.

1.9 An Enhanced Algorithm

I believe the primary bottleneck to BuildHON's scaling is the way it processes the sequential data. The lazy rule generation described in Section 1.5 allows for much more efficient execution than the greedy algorithm proposed by the first iteration of BuildHON. However, the algorithm still does not handle sequential data in a scalable fashion.

Here I believe research in sequential pattern mining may offer a solution. An FP-tree is an innovative data structure that enables the compression of transactional data [4] for pattern mining.

However, the FP-trees and the associated FP-growth algorithm have previously been applied to a different class of data. In the seminal paper, transaction data were unordered, and thus could be sorted or otherwise manipulated for more efficient processing. This is not true of the sequential data in which HON is interested. Bridging the application gap may be tricky. To date, I have not found an implementation of an FP-tree for an application with the same parameters as HON. However, I think it may be very useful. First, it will greatly increase the rate at which sequence patterns can be searched and translated into rules. Second, it may actually allow us to skip the rule extension process altogether by building a HON directly from the pattern tree. Third, it should (in most cases) reduce the storage requirements. Finally, the FP-tree as a structure is well-suited to processing on distributed graph systems.

An enhanced BuildHON algorithm would include the following two supersteps.

Algorithm 1 buildhon_from_fptree

1: Build FP-tree from raw sequence data
2: Build HON directly from FP-tree using depth-first search

The tricky part of this implementation will be combining depth-first search with HON's lazy rule generation. A more complete version of the algorithm will be included in the final presentation of this project.

1.10 A Reference Enhanced Implementation

Discuss here an implementation of the enhanced algorithm. Include what language/paradigm you used for the code.

1.11 Enhanced Scaling Results

Discuss here results from the enhanced algorithm. Include software and hardware configuration, where the input graph data sets came from, and how input data set characteristics were varied. Ideally plots of performance vs BOTH problem size changes AND hardware resources are desired. Did the performance as a function of size vary as you predicted?

1.12 Conclusion

Summarize your paper. Discuss possible future work and/or other options that may make sense.

1.13 Response to Reviews

This will be included only in the second and third iterations, and will be a summary of what you learned from the reviews you received from the prior pass, and how you modified the paper accordingly.

Bibliography

[1] Ching Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 11(3):5–9, 2011.

[2] David A Bader, Jonathan Berry, Adam Amos-Binks, Daniel Chavarría-Miranda, Charles Hastings, Kamesh Madduri, and Steven C Poulos. Stinger: Spatio-temporal interaction networks and graphs (sting) extensible representation. *Georgia Institute of Technology, Tech. Rep*, 2009.

[3] Douglas Gregor and Andrew Lumsdaine. The parallel bgl: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)*, 2:1–18, 2005.

[4] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *ACM sigmod record*, volume 29, pages 1–12. ACM, 2000.

[5] iCeNSA. Higher Order Networks. http://www.higherordernetwork.com/, 2018 (accessed Sept. 30, 2018).

[6] Jian Xu, Mandana Saebi, Bruno Ribeiro, Lance M Kaplan, and Nitesh V Chawla. Detecting anomalies in sequential data with higher-order networks. *arXiv preprint arXiv:1712.09658*, 2017.

[7] Jian Xu, Thanuka L Wickramarathne, and Nitesh V Chawla. Representing higher-order dependencies in networks. *Science advances*, 2(5):e1600028, 2016.

[8] Jian Xu, Thanuka L. Wickramarathne, and Nitesh V. Chawla. Higher Order Networks Repository, 2018 (accessed Sept. 30, 2018).