Chapter 1

BuildHON^2: A Scalable Higher-Order Network

Contributed by Steven Krieg

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 [6]. 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 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

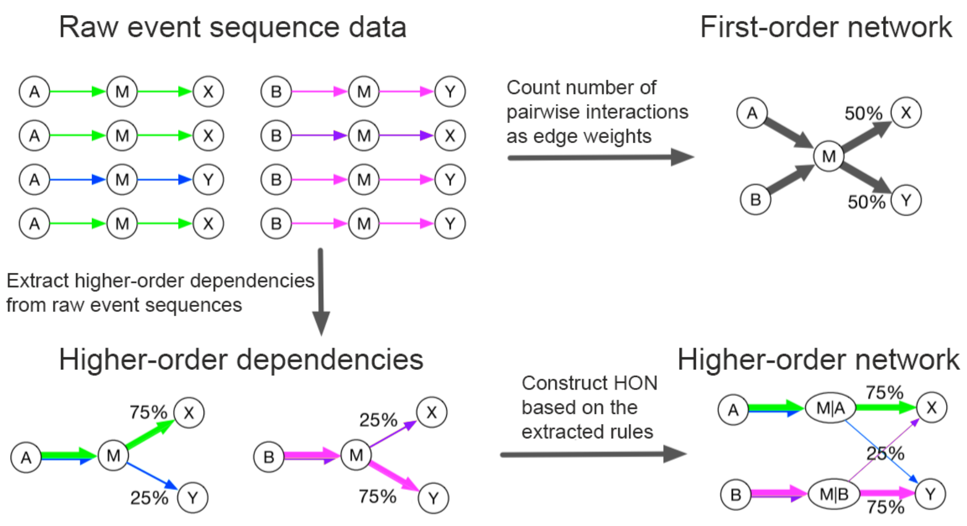
Raw event sequence data
First-order network
Count number of pairwise interactions as edge weights
50%
50%
Extract higher-order dependencies from raw event sequences
Higher-order dependencies
75%
25%
25%
75%
Construct HON based on the extracted rules
Higher-order network
75%
25%
75%
A
B
M
X
Y
M|A
M|B

Figure 1.1: The HON Idea [4]

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) [6], and synthetic clickstream data used for anomaly detection [5]. 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 [7].

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 [6]. The second iteration, BuildHON+, significantly improved the runtime of the algorithm by implementing a lazy version of rule extraction [5]. 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 [6]. 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 [6].

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.

1.7 A Reference Sequential Implementation

Discuss here your implementation of the basic sequential code. Include what language/paradigm you used for the code.

1.8 Sequential Scaling Results

Discuss here results from your sequential implementation. Include software and hardware configuration, where the input graph data sets came from, and how input data set characteristics were varied. Did the performance as a function of size vary as you predicted?

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 $ThresholdMultiplier$

```
define global $C$ as nested counter
define global $D$,$R$ as nested dictionary
define global $SourceToExtSource$, $StartingPoints$ as dictionary

function ExtractRules($T$, [$MaxOrder$, $MinSupport$, $ThresholdMultiplier = 1$])
    global $MaxOrder$, $MinSupport$, $Aggresiveness$
    BuildFirstOrderObservations($T$)
    BuildFirstOrderDistributions($T$)
    GenerateAllRules($MaxOrder$, $T$)

function BuildFirstOrderObservations($T$)
    for $t$ in $T$ do
        for $(Source, Target)$ in $t$ do
            $C[Source][Target]$ += 1
            $IC$.add($Source$)

function BuildFirstOrderDistributions($T$)
    for $Source$ in $C$ do
        for $Target$ in $C[Source]$ do
            if $C[Source][Target] < MinSupport$ then
                $C[Source][Target] = 0$
            for $Target$ in $C[Source]$ do
                if then$C[Source][Target] > 0$
                    $D[Source][Target] = C[Source][Target]/(\sum C[Source][*])$

function GenerateAllRules($MaxOrder$, $T$)
    for $Source$ in $D$ do
        AddToRules($Source$)
        ExtendRule($Source$, $Source$, 1, $T$)

function KLDThreshold($NewOrder$,$ExtSource$)
    return $ThresholdMultiplier \times NewOrder/log_2(1 + \sum C[ExtSource][*])$
function ExtendRule($Valid$, $Curr$, $order$, $T$)
    if $Order \leq MaxOrder$ then
        AddToRules($Source$)
    else
        $Distr = D[Valid]$
        if $-log_2(min(Distr[*].vals)) <$ KLDThreshold$(order + 1), Curr$ then
            AddToRules($Valid$)
        else
            $NewOrder = order + 1$
            $Extended$ = ExtendSource($Curr$)
            if $Extended = \emptyset$ then
                AddToRules($Valid$)
            else
                for $ExtSource$ in $Extended$ do
                    $ExtDistr = D[ExtSource]$
                    $divergence =$ KLD$(ExtDistr, Distr)$
                    if $divergence >$ KLDThreshold$(NewOrder, ExtSource)$ then
                        ExtendRule$(ExtSource, ExtSource, NewOrder, T)$
                    else
                        ExtendRule$(Valid, ExtSource, NewOrder, T)$
```

Algorithm 1 *(continued)*

```
function AddToRules(Source):
    for $order$ in [1..len($Source$) + 1] do
        $s = Source[0 : order]$
        if not $s$ in $D$ or len($D[s]$) == 0 then
            ExtendSource($s$[1:])
        for $t$ in $C[s]$ do
            if $C[s][t] > 0$ then
                $R[s][t] = C[s][t]$

function ExtendSource($Curr$)
    if $Curr$ in $SourceToExtSource$ then
        return $SourceToExtSource[Curr]$
    else
        ExtendObservation($Curr$)
        if $Curr$ in $SourceToExtSource$ then
            return $SourceToExtsource[Curr]$
        else
            return $\emptyset$

function ExtendObservation($Source$)
    if length($Source$) $> 1$ then
        if not $Source[1:]$ in $ExtC$ or $ExtC[Source] = \emptyset$ then
            ExtendObservation($Source[1:]$)
    $order = length(Source)$
    define $ExtC$ as nested counter
    for $Tindex, index$ in $StartingPoints[Source]$ do
        if $index - 1 \leq 0$ and $index + order < length(T[Tindex])$ then
            $ExtSource = T[Tindex][index - 1 : index + order]$
            $ExtC[ExtSource][Target] += 1$
            $StartingPoints[ExtSource]$.add($(Tindex, index - 1)$)
    if $ExtC = \emptyset$ then
        return
    for $S$ in $ExtC$ do
        for $t$ in $ExtC[s]$ do
            if $ExtC[s][t] < MinSupport$ then
                $ExtC[s][t] = 0$
            $C[s][t] += ExtC[s][t]$
        $CsSupport = \sum ExtC[s][*]$
        for $t$ in $ExtC[s]$ do
            if $ExtC[s][t] > 0$ then
                $D[s][t] = ExtC[s][t]/CsSupport$
                $SourceToExtSource[s[1:]].add(s)$

function BuildSourceToExtSource($order$)
    for $source$ in $D$ do
        if $len(source) = order$ then
            if $len(source) > 1$ then
                $NewOrder = len(source)$
                for $startingin[1..len(source)]$ do
                    $curr = source[starting:]$
                    if not $curr$ in $SourceToExtSource$ then
                        $SourceToExtSource[curr] = \emptyset$
                    if not $NewOrder$ in $SourceToExtSource[curr]$ then
                        $SourceToExtSource[curr][NewOrder] = \emptyset$
                    $SourceToExtSource[curr][NewOrder].add(source)$
```

Figure 1.2: The BuildHON+ Rule Exraction Algorithm [7]

1.9 An Enhanced Algorithm

Discuss here the outlines of an enhanced algorithm. This could be a parallel code, a code with some significant heuristics, or a code written in a non-traditional programming paradigm. Pseudocode is fine. Discuss what you think is the computational complexity.

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

[5] 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.

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

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