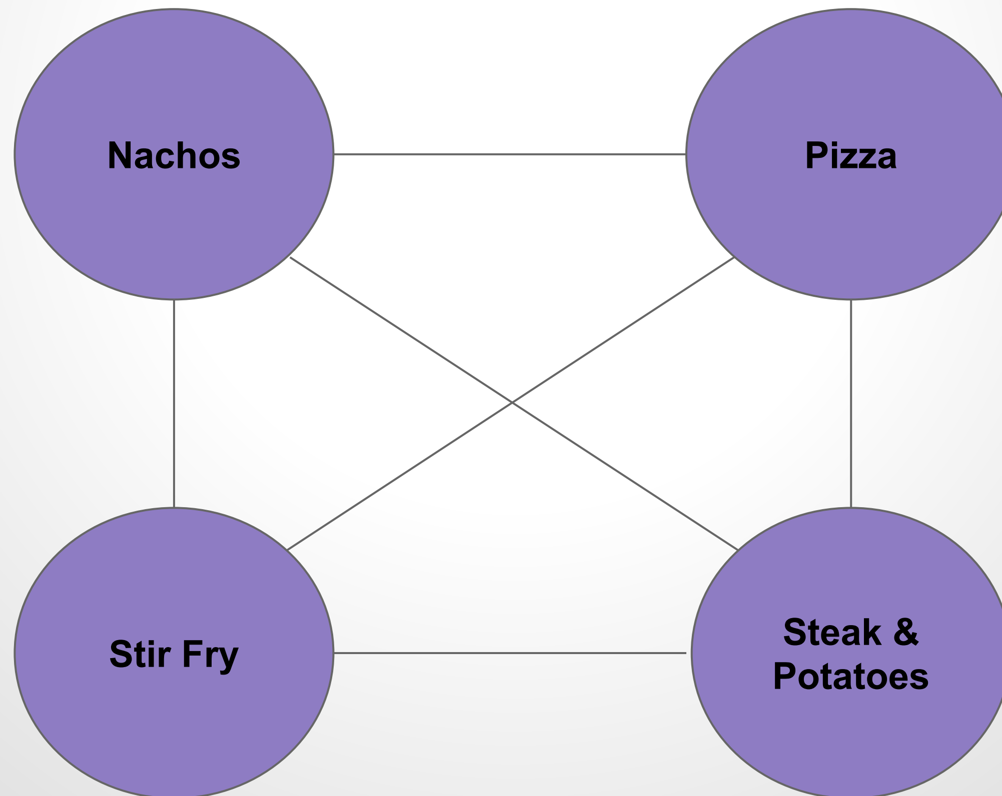
Higher Order Networks & BuildHon+

Steven Krieg

The Problem

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

Quite a problem, indeed...



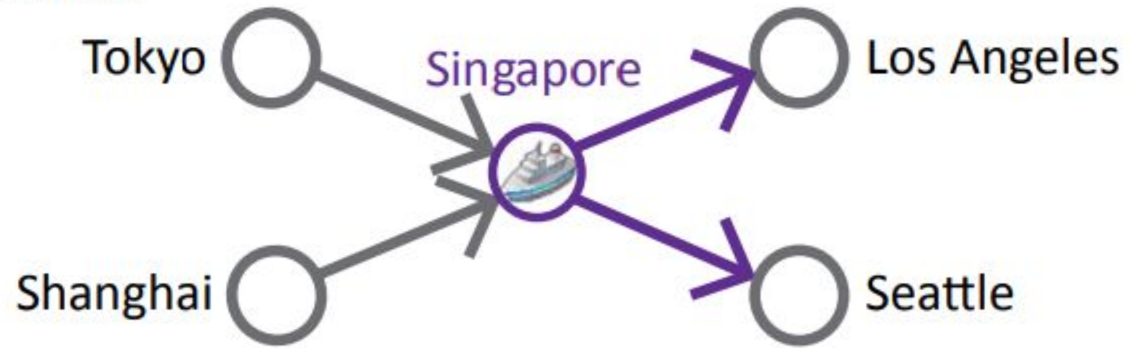
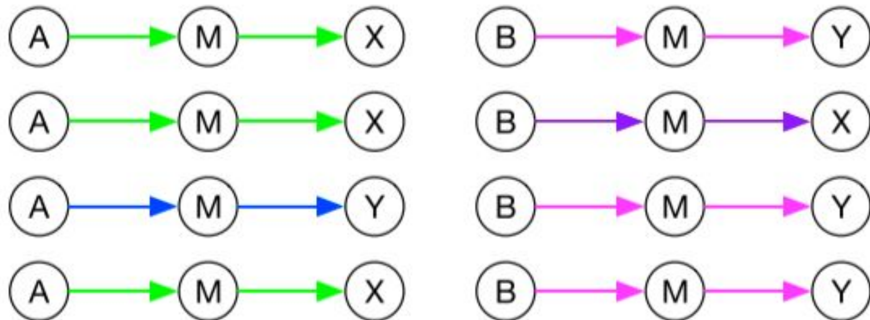


Image from [1]

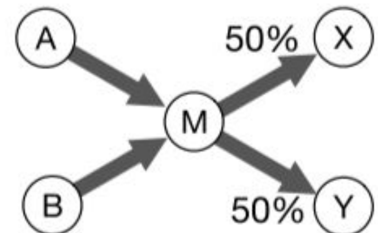
A Solution!

Raw event sequence data



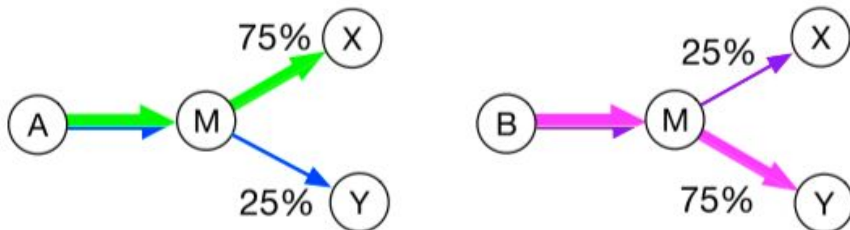
First-order network

Count number of pairwise interactions as edge weights



Extract higher-order dependencies from raw event sequences

Higher-order dependencies



Higher-order network

Construct HON based on the extracted rules

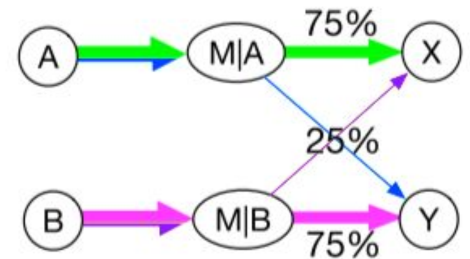


Image from [2]

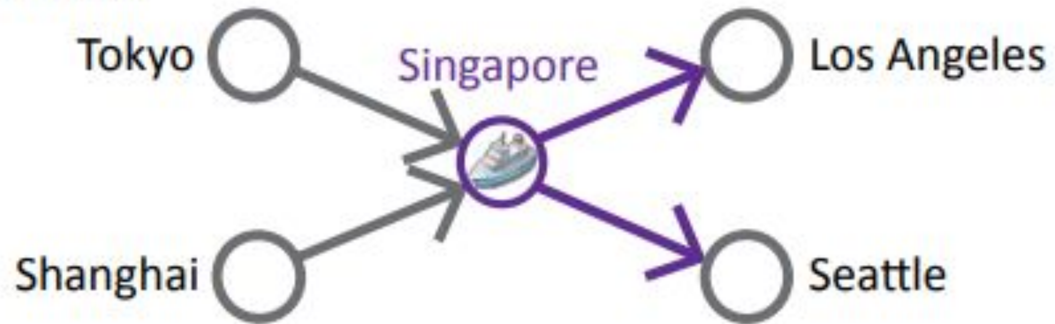


Image from [1]

Random Walker Results*

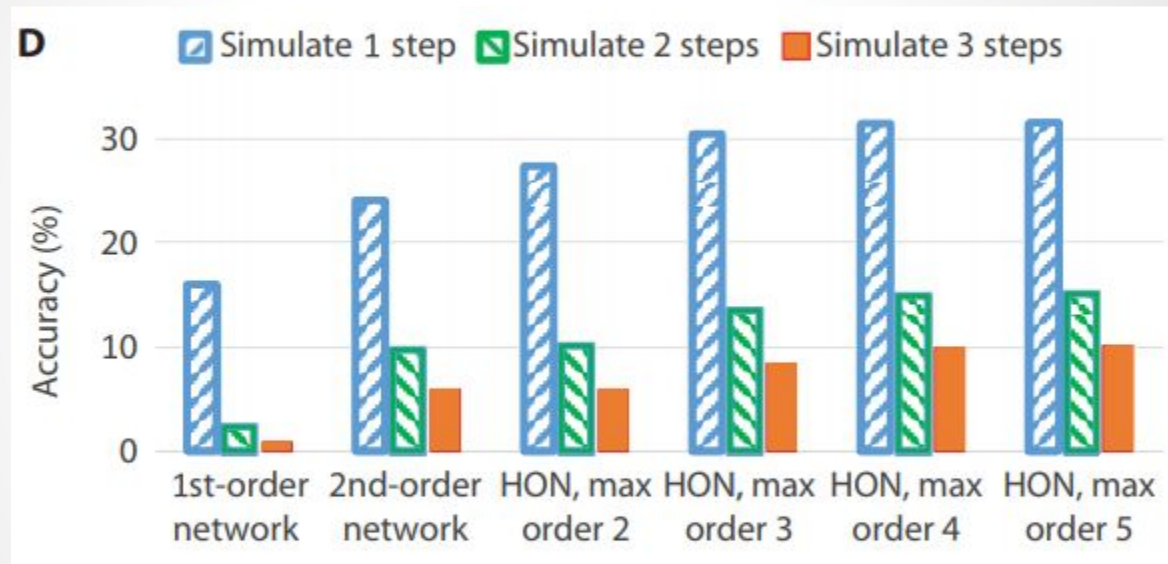


Image from [1]

*The goal of this experiment was not link prediction but to demonstrate the **improvement in representation** using HON.

HON Highlights

-is a network representation of a weighted digraph
-rewires the existing network so that nodes represent a series rather than a singular entity
-enables higher accuracy without needing new analysis tools/algorithms

HON Applications

-ranking (PageRank, etc.)
-anomaly detection
-NLP
-social
-biology
-etc...

The Kernel: BuildHon

Algorithm used to construct the HON

Has 2 main steps:

1. Rule extraction
2. Network rewiring

Step 1: Rule Extraction

1. Count the number of sequential node interactions at the first-order (basically the normal network)
2. Normalize the distributions for each pairwise interaction
3. For each fork node, add the preceding step and see how that changes the distribution of the sequence
4. If the change is “significant” (above a selected threshold), add a second-order dependency and repeat the process recursively to determine higher orders

Step 2: Network Rewiring

1. Construct a conventional first-order network
2. For every second order rule, add the corresponding node;
   Rewire the previous first-order link to connect to the new higher-order node; then repeat the process for third order rules and so on
3. Once we finish the highest order, rewire all out-edges from that order to connect to nodes with the highest orders possible.

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)
```

Scalability :S

Network representation (global shipping data)	Number of edges	Number of nodes	Network density	* Clustering time (mins)	** Ranking time (s)
Conventional first-order	31,028	2,675	4.3×10^{-3}	4	1.3
Fixed second-order	116,611	19,182	3.2×10^{-4}	73	7.7
HON, max order two	64,914	17,235	2.2×10^{-4}	45	4.8
HON, max order three	78,415	26,577	1.1×10^{-4}	63	6.2
HON, max order four	83,480	30,631	8.9×10^{-5}	67	7.0
HON, max order five	85,025	31,854	8.4×10^{-5}	68	7.6

* Using MapEquation with 1000 iterations
** Using PageRank

Source: [2]

Time Complexity

$$L * N * \sum_{i=1}^{k} ((i + 1) R_i)$$

where L is the count of records in the raw data;
N is the number of unique nodes in the raw data;
k is the maximum order of dependency;
R_i is the count of dependencies at order i

(*the theoretical upper bound is exponential but is not really helpful for real data sets, in which orders of dependency tend to follow an inverse power law)

Data Sets

- Synthetic web clickstreams (11 billion nodes)
- Global shipping data (3,415,577 voyages made by 65,591 ships between May 1st , 2012 and April 30th, 2013)

Related Work

- Going beyond Markov decisions to higher-order dependencies is not a new idea, but most work has focused on stochastic processes rather than on the representation problem.
- VOM (variable-order Markov) models for predictions are related, but occupy a different niche.

Further Resources

- [1] Xu, J., Wickramarathne, T., & Chawla, N. (2016). Representing higher-order dependencies in networks. Science Advances, 2(5), E1600028.
- [2] http://www.higherordernetwork.com/
- [3] https://github.com/xyjprc/hon
- [4] Xu, J., Saebi, M., Ribeiro, B., Kaplan, L., & Chawla, N. (2017). Detecting Anomalies in Sequential Data with Higher-order Networks.
- [5] Cui Jiao, Guo Jun, Zhang Cangsong, & Chang Xiaojun. (2012). Implementation of random walk algorithm by parallel computing. Fuzzy Systems and Knowledge Discovery (FSKD), 2012 9th International Conference on, 2477-2481.