GrowHON: A Scalable Algorithm for Growing Higher-order Networks of Sequences

Steven J. Krieg(✉), Peter M. Kogge, and Nitesh V. Chawla

University of Notre Dame, Notre Dame, IN 46556, USA
{skrieg,kogge,nchawla}@nd.edu

Abstract. Networks are powerful and flexible structures for expressing relationships between entities, but in traditional network models an edge can only represent a relationship between a single pair of entities. Higher-order networks (HONs) overcome this limitation by allowing each node to represent a sequence of entities, which allows edges to naturally express higher-order relationships. While HONs have proven effective in several domains, and previous works have been forced to choose between scalability and thorough detection of higher-order dependencies. To achieve both scalability and accurate encoding of higher-order dependencies, we introduce GrowHON, an algorithm that generates a HON by embedding the sequence input in a tree structure, pruning the non-meaningful sequences, and converting the tree into a network (Code available at https://github.com/sjkrieg/growhon). We demonstrate that GrowHON is scalable with respect to both the size of the input and order of the network, and that it preserves important higher-order dependencies that are not captured by prior approaches. These contributions lay an important foundation for higher-order networks to continue to evolve and represent larger and more complex data.

Keywords: Higher order networks · Sequence mining · Graph models

1 Introduction

Networks are powerful and flexible structures for expressing relationships between entities. However, some relationships are too complex to be represented by traditional network representations that typically rely on first order representation. In this **first-order network** (FON) representation, each entity in a sequence (typically a set of events or states that are ordered by time) is represented by a single node, with an edge connecting each pair of entities that are adjacent in the sequence. The resulting network assumes the first order Markov property, which means that at a given state, all necessary information about that state is available locally. For example, in Fig. 1, a random walker that arrives at node 2 could transition to nodes 3 or 4. But the underlying sequences exhibit higher-order dependencies: 2 is followed by 3 only when preceded by 1,

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2021
R. M. Benito et al. (Eds.): COMPLEX NETWORKS 2020, SCI 944, pp. 485–496, 2021.
https://doi.org/10.1007/978-3-030-65351-4_39

and followed by 4 when preceded by 3. This vital information is lost during network construction. Several recent works have shown that this limitation of FONs is problematic in a number of domains, including the study of user behaviors [3], citation networks [18], trade relations [9], human mobility and navigation patterns [14,22], information networks [21], the spread of invasive species [24], anomaly detection [20], and others [10,14].

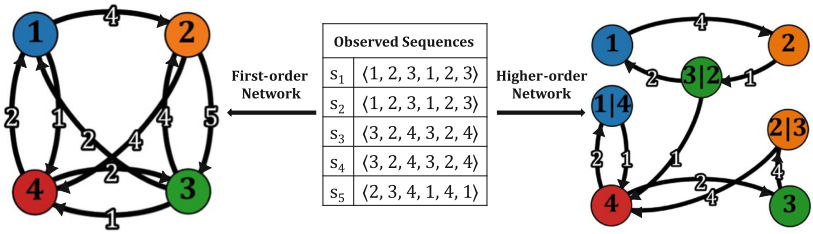

Fig. 1. A toy example of the differences between a FON and HON representation of a set of sequences.

A **higher-order network** (HON) representation, introduced as BuildHON by Xu et al. [24], is a solution to this problem that seeks to preserve dependencies by allowing each node to represent a variable-length sequence of entities, rather than a single entity. For example, in Fig. 1, node 3|2 (read as 3 *given* 2) represents the current state 3 conditioned on the prior state 2. This distinguishes it from node 3, and thus allows the network to more fully represent the underlying statistical patterns. The BuildHON framework is flexible because it allows nodes of varying order to coexist, and generalizable because it allows for existing network analysis tools, like clustering and random walking, to be used without modification. Finally and perhaps most importantly, a HON seeks to only preserve higher-order patterns that are statistically significant, which helps control the size of the network and prevent overfitting.

The trade-offs for the increased representative accuracy of such a HON are increased network size and the cost of network generation, which becomes computationally expensive as the order of the network and size of the input increases. This paper introduces GrowHON, an algorithm that offers two main advantages over the algorithms utilized in prior work (BuildHON/BuildHON+) [20,24]:

1. Increased scalability with respect to order and input size. By embedding sequences in a tree, GrowHON avoids repeated searches through the input and enables efficient computation.
2. More thorough detection of dependencies at higher orders. By testing higher-order nodes at first, GrowHON is able to preserve important sequences that are missed by other methods.

The rest of the paper proceeds as follows. First we survey related work (Sect. 2). Next we introduce the problem of HON generation and present GrowHON

(Sect. 3). Then we experimentally demonstrate GrowHON's increased scalability and ability to detect dependencies at higher orders. Finally we conclude and discuss opportunities for future work (Sect. 5).

2 Related Work

Several recent works share the conclusion that first-order Markov models are an inadequate representation of many real-world processes, which often exhibit complex spatiotemporal dependencies. While a multitude of approaches exist, we focus our discussion on studies that utilize network structures to model these dependencies alongside, or together with, other patterns of connectivity between entities [15]. These works span many domains, including the study of user behaviors [3], citation networks [18], trade relations [9], human mobility and navigation patterns [14,22], information networks [21], the spread of invasive species [24], anomaly detection [20], and more [10,14]. These works agree on the need to reach beyond the limits of first-order networks, but differ in methodological approach. We build on the model of Xu et al., which encodes dependencies of variable order into a single-layered network structure [20,24]. Compared with multi-layered networks [21] and models that rely on supplemental higher-order path information [14,18], this approach has the advantage that existing network analysis tools can be utilized without modification. Additionally, rather than inferring a fixed order for the entire model—which can produce a model that is overfit on some sequences and underfit on others—it allows nodes of variable order to coexist in the same space. Xu et al. accomplished this by testing each higher-order pattern and only preserving those that were expected to reduce the entropy of the network. Saebi et al. noted that this approach was combinatorially explosive and developed a lazy algorithm that starts with lower-order patterns and seeks to extend them one entity at a time [20]. This method, BuildHON+, successfully mitigated the combinatorial problem, but is still limited (by repeated searches through the input and lazy pattern testing) in its scalability and ability to thoroughly detect dependencies at higher-orders.

We additionally distinguish our task of higher-order representation from that of higher-order network analysis, which uses substructures like motifs [11] or graphlets [16] to model statistically significant patterns of connectivity between nodes. Higher-order analysis is important and has sparked new approaches to clustering [1,23,25], representation learning [17], and more, but is ultimately concerned with analyzing an existing network. In this work, however, we are concerned with the upstream task of creating a network—one that accurately represents the underlying data and thus enables meaningful analysis.

3 Methods

3.1 Problem Setting

We define a **sequence** $s = \langle a_0, a_1, ..., a_n \rangle$ as an ordered collection of elements. Each $a_i \in s$ represents a discrete state or entity, and each adjacent pair a_i

and a_{i+1} represents a transition from a_i to a_{i+1}. We call $s' = \langle a'_0, a'_1, ..., a'_m \rangle$ a substring[1] of s, denoted as $s' \sqsubseteq s$, if and only if all the elements in s' also appear in s in exactly the same order, i.e.

$$s' \sqsubseteq s \iff \exists j : \forall a'_i \in s', a'_i = a_{i+j}. \tag{1}$$

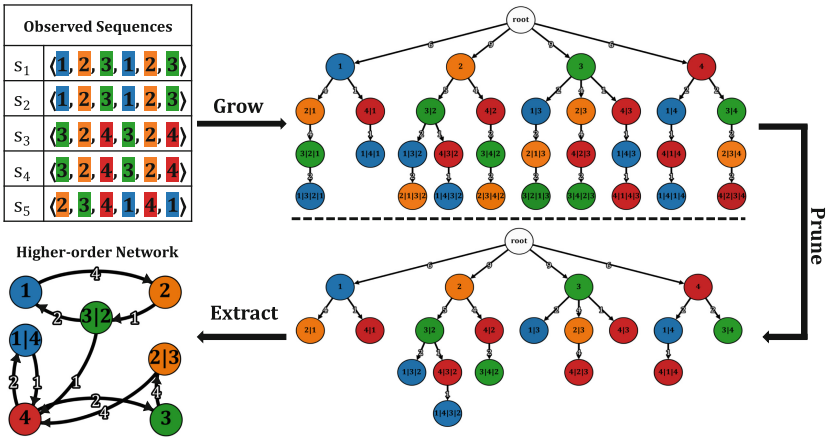

Fig. 2. An overview of GrowHON on toy data using $k = 3$. Nodes are colored according to their destination state.

In a FON $G_1 = (V_1, E_1)$, a set of sequences $S = \{s_0, s_1, ..., s_N\}$ is represented by a set of nodes, V_1, and a set of edges, E_1. Each entity in $\{\bigcup S\}$ maps to exactly one node in V_1, and two nodes u and v are joined by a directed edge $u \to v$ if u precedes v in some sequence $s \in S$, i.e. $\langle u, v \rangle \sqsubseteq s$. Edges are additionally weighted such that $w(u, v)$ is the number of times $\langle u, v \rangle$ appears across all sequences in S. Statistically significant patterns involving more than two entities, which we call **higher-order dependencies**, are thus lost in the construction of G_1.

A HON, on the other hand, seeks to preserve these dependencies. Let $G_k = (V_k, E_k)$ be a HON such that k is the maximum order, or the amount of history that can be encoded by each node. The set of possible nodes, V'_k, is the set of all substrings of length $k + 1$ or less:

$$V'_k = \bigcup_{m=1}^{k+1} \{\langle a'_{i-m+1}, ..., a'_i \rangle : (\exists s \in S) \wedge (\exists i > 0), \langle a'_{i-m+1}, ..., a'_i \rangle \sqsubseteq s\}. \tag{2}$$

In practice, we encode nodes in the following form in order to emphasize that the destination state a_m is conditioned on a sequence of prior states $a_{m-1}, ..., a_0$:

$$a_m|a_{m-1}|...|a_1|a_0 \equiv \langle a_0, ..., a_{m-1}, a_m \rangle, \tag{3}$$

[1] We use the term "substring" instead of "subsequence" because the formal definition of a subsequence allows for intermediate elements to be removed, and thus does not preserve adjacency from the original sequence.

In both cases, a_0 is the earliest element in the original sequence and a_m is the latest. For example, node 3|2 in Figs. 1 and 2 represents the substring $\langle 2, 3\rangle$. This definition of a node allows for edges in E_k, which are defined in the same way as in E_1 but generalized to allow for the fact that u and v are substrings, to naturally represent higher-order interactions between nodes. For example, the edge $3|2 \rightarrow 1$ represents the substring $\langle 2, 3, 1\rangle$ that cannot be represented by a single edge in E_1.

We define a node's **order** as the number of states—current and prior—it represents. A HON can contain nodes with variable order, so we use cardinality to denote the order of a node, such that for a given node $u = a_m|a_{m-1}|...|a_0$, we have $|u| = m + 1$. This heterogeneity with respect to node order means that there are many possible representations of a given sequence. For example, $1 \rightarrow 2$ and $1 \rightarrow 2|1$ are both valid representations of $\langle 1, 2\rangle$. The key objective in generating a HON is to produce a representation, i.e. a final set of nodes V_k and edges E_k, that preserves the statistically significant higher-order sequences in a generalizable (avoids overfitting) and concise form. GrowHON accomplishes this through three phases:

1. Grow a sequence tree, which is a compact and connected representation of the original sequences that enables efficient pruning and extraction.
2. Prune statistically insignificant sequences. Pruning is performed in-place and top-down (with respect to order) in order to ensure all higher-order dependencies are preserved.
3. Extract sequences by converting them to edges in a manner that preserves the topological integrity of the network.

The following sections detail each of the three phases, and Fig. 2 illustrates the output of each phase on a toy data set.

3.2 Phase 1: Grow

Grow processes each sequence by embedding the observed substrings of length $m \leq k + 1$ as branches in a tree. For example, in Fig. 2, the first substring in s_1, $\langle 1, 2, 3, 1\rangle$, produces the leftmost branch on the grown tree: $1 \rightarrow 2|1 \rightarrow 3|2|1 \rightarrow 1|3|2|1$. Algorithm 1 details the procedure, which is somewhat similar to growing a frequent-pattern tree (FP-tree) [6]. The main benefit to this structure is that it is very efficient to compute transition probability distributions during the prune phase, since all of a node's outgoing neighbors are represented as its children in the tree.

Alongside the tree, we utilize a hash table called the *nmap* (**n**ode **map**) to store object references and thus enable efficient node lookups. For a given node $u = a_m|a_{m-1}|...|a_0$, we call any node $u' = a_m|a_{m-1}|...|a_z$ for some $0 < z < m$ a **lower-order counterpart** of u, since it represents the same destination state a_m but with z fewer prior states. We express this relationship using the logical shift operator $\gg$:

$$a_m|a_{m-1}|...|a_0 \gg z = a_m|a_{m-1}|...|a_z \tag{4}$$

Algorithm 1. GrowHON Phase 1: Grow

```
1: function Grow(S, k)
2:     Q ← an empty queue with length k + 1
3:     t.root ← a dummy node
4:     t.nmap ← an empty hash table
5:     for each sequence s in S do
6:         Prime Q with the first k + 1 elements in s
7:         for each remaining element a in s do
8:             parent ← t.root
9:             for each element u in Q do
10:                if u in parent.children then
11:                    child ← parent.GetChild(u)    ▷ Retrieve child from hash table
12:                    child.indeg ← child.indeg + 1
13:                else
14:                    child ← parent.AddChild(u)    ▷ Store child in a hash table
15:                    child.indeg ← 1
16:                    child.outdeg ← 0
17:                    t.nmap.Insert(child)
18:                parent.outdeg ← parent.outdeg + 1
19:                parent ← child
20:            Q.pop()
21:            Q.push(a)
22:        Repeat lines 9-20 until Q is empty
23:    return t
```

Every higher-order node has exactly one lower-order counterpart for each $0 < z < |u'|$. This is trivially proven by the fact that if there exists some $s \in S$ such that $\langle a_0, a_1, ..., a_m \rangle \sqsubseteq s$, then $\langle a_z, a_{z+1}, ..., a_m \rangle \sqsubseteq s$. This means that we can look up a node's lower-order counterpart in constant time using the *nmap*, rather than traversing the tree from the root, which could require up to k lookups.

3.3 Phase 2: Prune

Given a fully grown tree, Prune decides which nodes to preserve in the HON. Algorithm 2 details the procedure. We follow Saebi et al. in seeking to preserve a given node u if the Kullback-Leibler divergence (relative entropy) of its outgoing transition probabilities, measured with respect to its lower-order counterpart u', exceeds a threshold function f [20]. We define both as follows:

$$D_{KL}(u \parallel u \gg 1) = \sum_{v \in \mathcal{N}(u)} P(u \to v) log_2 \frac{P(u \to v)}{P(u \gg 1 \to v \gg 1)}, \tag{5}$$

$$f(u, \tau) = \frac{\tau |u|}{log_2(1 + indeg(u))}, \tag{6}$$

where $\mathcal{N}$ represents the set of outgoing neighbors or children in the grown tree, $P(u \to v) = \frac{w(u,v)}{outdeg(u)}$ represents the probability of transition from node u to

Algorithm 2. GrowHON Phase 2: Prune

```
1: function Prune(t, τ)
2:     for i ← t.height − 1 down to 1 do
3:         for each node u in t where |u| = i do
4:             if u.marked or D_KL(u ‖ u ≫ 1) > f(u, τ) then
5:                 u.parent.marked = true          ▷ Ensure ancestors are preserved
6:             else
7:                 u.outdeg ← 0
8:                 for each child c in u.children do
9:                     c.indeg ← 0
```

Algorithm 3. GrowHON Phase 3: Extract

```
1: function Extract(t)
2:     E ← an empty edgelist
3:     Extract-Helper(E, t.root)
4:     return E

5: function Extract-Helper(E, u)
6:     if u.indeg > 0 then
7:         if |u| > 1 then                  ▷ Nodes at level 1 cannot be destinations
8:             v ← u
9:             while v.outdeg = 0 and |v| > 1 do   ▷ Handle the dead-end (leaf) case
10:                v ← v ≫ 1
11:            E.insert(u.parent, v, u.indeg)
12:        for child in n.children do
13:            Extract-Helper(E, child)
```

node v, and τ is a free parameter (we use a default of 1.0). f increases monotonically as τ and $|u|$ increases and decreases monotonically as $indeg(u)$ increases, which helps control for the fact that D_{KL} is biased at higher orders, where edge weights are sparser and transition probabilities are noisier.

The key advantage afforded by Prune is that higher-order nodes are tested first. This means that when a higher-order node is preserved, we can also ensure that its ancestors are preserved (line 22)—otherwise it may have no in-edges in the resulting network. Previous approaches, limited by computational complexity, tested higher-order dependencies in bottom-up fashion [20,24]. This bottom-up testing implicitly assumes that a node u can only have a dependency if $u \gg 1$ also had a dependency, which is often not the case.

3.4 Phase 3: Extract

The final step is to convert the pruned tree into a HON. Extract, detailed in Algorithm 3, recursively traverses the pruned tree and converts each tree node into an edge in the HON. Because each tree node represents the final element in a substring, the parent/child relationship is directly converted to a

Table 1. A summary of the data sets used for evaluation. N and $\bar{n}$ represent the number of sequences and average sequence length, respectively. The Airport data served as the source for five synthetic sets of sequences.

Name	N	$\bar{n}$	$N\bar{n}$	$\lvert V_1\rvert$	$\lvert E_1\rvert$	$\frac{\lvert E_1\rvert}{\lvert V_1\rvert}$
Shipping	54,892	151.15	8,296,770	5,590	369,965	66.18
T2D (ICD)	913,475	39.18	35,792,618	914	481,020	526.28
T2D (CCS)	913,475	31.54	28,809,027	304	85,025	279.69
Airport	—	—	{1,2,3,4,5}00,000,000	1,922	31,491	16.38

source/destination relationship in the HON for all non-leaf nodes. When a leaf node is detected, Extract attempts to redirect the edge to a lower-order counterpart. This seeks to maximize the preserved higher-order information while ensuring that tree leaves do not produce nodes with no out-edges, which would disrupt the flow of information.

3.5 Asymptotic Complexity

Grow requires n queue pushes and n pops for a sequence of n elements. For each substring of length $k+1$, each element induces either a weight increment (for an existing child) or the insertion of a new child—both of which are constant-time operations[2]. In total, Grow considers $n-k$ substrings with length $k+1$ and k substrings with length $< k+1$ (line 22). The time complexity of processing a single sequence is thus bounded by $O(kn)$. If there are N total sequences with an average length of $\bar{n}$, then the function's complexity is bounded by $O(kN\bar{n})$. Prune requires a constant number of operations for each tree node, since each node above order 1 could be considered in a calculation of D_{KL}, and each node below order k will be looked up once in *nmap*. Extract similarly requires a constant number of operations, on average, for each node. If a leaf node u requires more than one lookup to find an appropriate lower-order destination, that means $u \gg 1$ was pruned and will be skipped by Extract. Thus the complexity of both Prune and Extract is bound by the number of nodes in the tree, i.e. $|V_k'|$. While it is possible to derive an upper bound for $|V_k'|$, GrowHON's complexity will always be dominated by Grow, and is bounded by $O(kN\bar{n})$.

4 Experimental Results

GrowHON is implemented in Python 3.7.3, and all code is publicly available[3]. We evaluated the scalability of GrowHON and the representation of the resulting networks using eight data sets—three real and five synthetic—each of which are summarized in Table 1. The real data includes the set of global shipping

[2] In practice, creating a new node takes much longer than a weight increment, so the number of unique substrings has a significant effect on runtime.

[3] https://github.com/sjkrieg/growhon.

routes over 15 years (1997–2012) that Xu et al. studied in their seminal HON manuscript [24], and a set of diagnoses sequences for type-2 diabetes (T2D) patients under two distinct mapping schemas: the first using the ninth revision of the International Classification of Diseases (ICD) and the second using the Clinical Classification Software (CCS) taxonomy[7]. Both represent the same set of real patients, but with diagnoses aggregated at different levels of granularity and density. We generated the synthetic data by constructing a first-order network of U.S. airport travel in 2018 based on data from the Bureau of Transportation Statistics [2]. From this network we used a random walker to generate five sets of sequences, each of a different size.

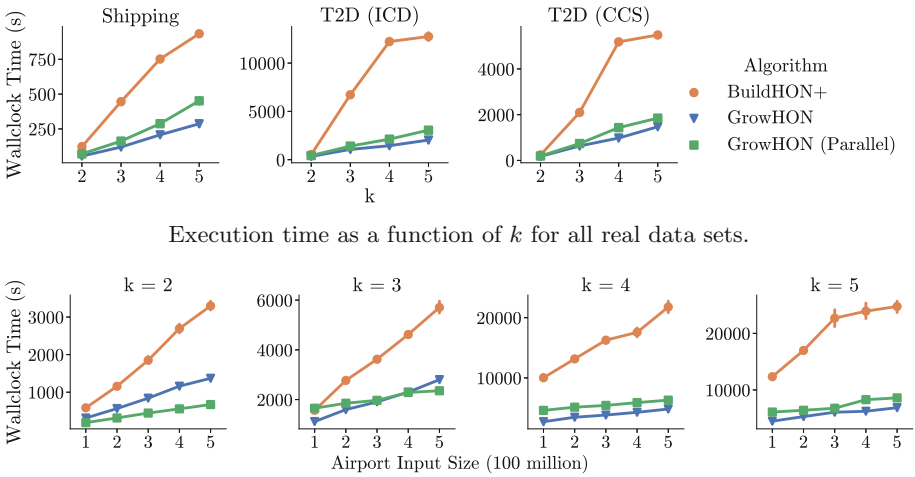

Fig. 3. Execution times, measured as wallclock time, of each algorithm on all data sets using $k = 2..5$. The reported values are the means of 10 iterations, and error bars represent standard deviations.

Figure 3 shows the execution times of each algorithm using $k = 2..5$ for 10 iterations on each data sets. All Shipping and Airport experiments utilized Intel Xeon E5-2680 v3 @2.50GHz CPUs, and all T2D experiments (due to data sensitivity) utilized Intel Xeon E5-2686 v4 @2.30GHz CPUs. GrowHON was faster than BuildHON+ in all experiments, in many cases by almost an order of magnitude—especially at higher k. Perhaps more importantly, GrowHON demonstrates greater scalability, with respect to both k and input size.

In addition to the base version of GrowHON, we used Ray, a Python framework for efficient distributed computation [12], to implement a parallel version of Grow. This modified version utilizes a driver for growing the tree and a group of workers (four, in this case) for enumerating substrings from the input sequences. Despite Ray's efficiency, our experiments show that, when $k > 3$,

Table 2. Differences between network sizes when $k = 5$.

Algorithm	Shipping		T2D (ICD)		T2D (CCS)	
	$\|V_5\|$	$\|E_5\|$	$\|V_5\|$	$\|E_5\|$	$\|V_5\|$	$\|E_5\|$
BuildHON+	2,010,511	5,937,933	17,918,723	50,272,035	11,620,878	36,423,805
GrowHON	2,596,214	6,893,689	21,683,241	54,378,822	15,285,534	40,461,838

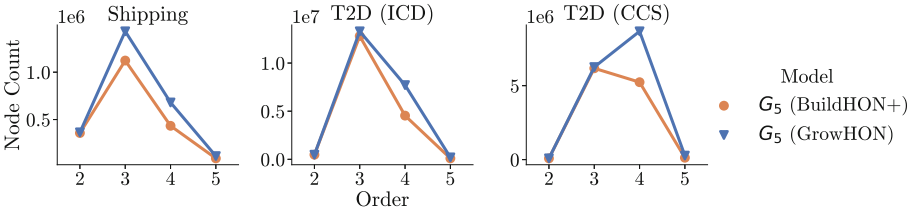

Fig. 4. Distribution of nodes by order for the networks produced by both algorithms.

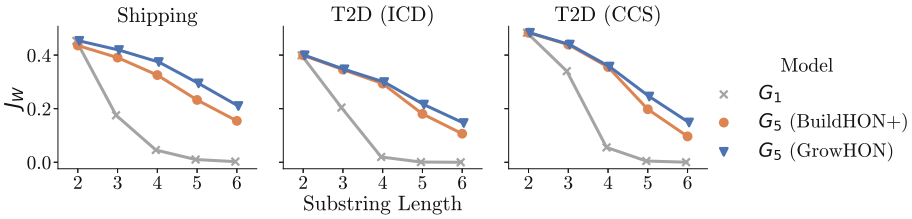

Fig. 5. Results of using a random walker to reproduce the original sequences. Each plot shows weighted Jaccard similarity (J_W) as a function of substring length (i.e. number of steps taken by the random walker). The reported values are the means of 10 samples, and standard deviations were $< .0001$ in all cases.

the combinatorial explosion of substrings causes the cost of message passing to outweigh its benefits.

Table 2 shows the difference in network sizes at $k = 5$, and Fig. 4 visualizes the distributions of nodes by order. These differences are attributable to GrowHON's top-down pruning procedure, which allowed it to preserve higher-order dependencies that BuildHON+'s lazy testing did not capture.

To quantify the effect these additional nodes have on the representational quality, we utilized a random walker to generate samples of synthetic sequences from each network. Since random walking is core to many network applications like PageRank [13], community detection [5], and network embedding [4,19], it is critical that such walks reflect real patterns in the underlying data. We compared each synthetic sample to the real sequences using weighted Jaccard (Ruzicka) similarity J_W, which returns a score between 0.0 (sets are totally disjoint) and 1.0 (sets are identical with respect to both membership and frequency) [8]. Figure 5 shows the results for $m = 2..6$ (substring length) on a FON (G_1) and the HONs generated with $k = 5$ (G_5) by both algorithms. At $m = 2$, where a substring is an

edge in E_1, all models performed similarly. As m increased, G_1's performance deteriorated quickly. While BuildHON+'s G_5 significantly outperformed G_1 at all $m > 2$, GrowHON's G_5 was best able to better reproduce the sets of longer substrings. This supports the conclusion that the additional higher-order dependencies preserved by GrowHON are important in representing the underlying data.

5 Conclusions and Future Work

Higher-order networks (HONs) overcome the Markovian limitations of first-order networks by allowing each node to represent part of a sequence, rather than a single entity. This allows edges to naturally encode higher-order relationships between entities and improves the representative quality of the network with respect to the original sequences. However, the process of enumerating and testing for higher-order dependencies is computationally complex, especially as the order of the network increases, and previous approaches have been limited by the trade-off between efficient computation and thorough detection of dependencies. We introduced GrowHON, an algorithm that grows a HON by embedding the input in a tree, pruning the non-meaningful sequences, and converting the preserved sequences into an edgelist. We demonstrated that GrowHON is scalable with respect to both the size of the input and order of the network, and that its top-down pruning procedure preserves important higher-order dependencies that are missed by prior approaches.

While our work has mostly focused on the computational procedure of growing a HON, there are still many opportunities for advancing the HON framework. At present, GrowHON only captures information about sequence order, but this could be extended to consider additional information like distance in time or space. Additionally, it does not test the assumption that sequences are strictly ordered, i.e. that 1|2|3 is different than 1|3|2, and is limited in its ability to compute how relevant each entity is to the overall sequence. GrowHON could also be extended to include heterogeneous information, and even use this information in deciding whether a given sequence is statistically meaningful. We believe that GrowHON lays an foundation for studying these problems and creating even more meaningful representations of large and complex data.

References

1. Benson, A.R., Gleich, D.F., Leskovec, J.: Higher-order organization of complex networks. Science **353**(6295), 163–166 (2016)
2. Bureau of Transportation Statistics: Transtats. https://www.transtats.bts.gov/. Accessed 30 Sep 2019
3. Chierichetti, F., Kumar, R., Raghavan, P., Sarlos, T.: Are web users really Markovian? In: Proceedings of the 21st International Conference on World Wide Web, pp. 609–618 (2012)
4. Cui, P., Wang, X., Pei, J., Zhu, W.: A survey on network embedding. IEEE Trans. Knowl. Data Eng. **31**(5), 833–852 (2018)

5. Fortunato, S.: Community detection in graphs. Phys. Rep. **486**(3–5), 75–174 (2010)
6. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. ACM Sigmod Rec. **29**(2), 1–12 (2000)
7. Healthcare Cost and Utilization Project (HCUP): Clinical classification software, March 2017. http://www.hcup-us.ahrq.gov. Accessed 8 Jan 2020
8. Ioffe, S.: Improved consistent sampling, weighted minhash and l1 sketching. In: 2010 IEEE International Conference on Data Mining, pp. 246–255. IEEE (2010)
9. Koher, A., Lentz, H.H., Hövel, P., Sokolov, I.M.: Infections on temporal networks-a matrix-based approach. PloS ONE **11**(4), e0151209 (2016)
10. Lambiotte, R., Rosvall, M., Scholtes, I.: From networks to optimal higher-order models of complex systems. Nat. Phys. **15**(4), 313–320 (2019)
11. Milo, R., Shen-Orr, S., et al.: Network motifs: simple building blocks of complex networks. Science **298**(5594), 824–827 (2002)
12. Moritz, P., Nishihara, R., et al.: Ray: A distributed framework for emerging AI applications. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pp. 561–577 (2018)
13. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: bringing order to the web. Technical report, Stanford InfoLab (1999)
14. Peixoto, T.P., Rosvall, M.: Modelling sequences and temporal networks with dynamic community structures. Nat. Commun. **8**(1), 582 (2017)
15. Porter, M.A.: Nonlinearity+ networks: a 2020 vision. In: Emerging Frontiers in Nonlinear Science, pp. 131–159. Springer, Cham (2020)
16. Pržulj, N., Corneil, D.G., Jurisica, I.: Modeling interactome: scale-free or geometric? Bioinformatics **20**(18), 3508–3515 (2004)
17. Rossi, R.A., Ahmed, N.K., Koh, E.: Higher-order network representation learning. In: Companion Proceedings of the The Web Conference 2018, pp. 3–4. International World Wide Web Conferences Steering Committee (2018)
18. Rosvall, M., Esquivel, A.V., Lancichinetti, A., West, J.D., Lambiotte, R.: Memory in network flows and its effects on spreading dynamics and community detection. Nat. Commun. **5**, 4630 (2014)
19. Saebi, M., Ciampaglia, G.L., Kaplan, L.M., Chawla, N.V.: Honem: learning embedding for higher order networks. Big Data **8**(4), 255–269 (2020)
20. Saebi, M., Xu, J., Kaplan, L.M., Ribeiro, B., Chawla, N.V.: Efficient modeling of higher-order dependencies in networks: from algorithm to application for anomaly detection. EPJ Data Sci. **9**(1), 15 (2020)
21. Scholtes, I.: When is a network a network? In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1037–1046. ACM (2017)
22. Scholtes, I., et al.: Causality-driven slow-down and speed-up of diffusion in non-Markovian temporal networks. Nat. Commun. **5**, 5024 (2014)
23. Tsourakakis, C.E., Pachocki, J., Mitzenmacher, M.: Scalable motif-aware graph clustering. In: Proceedings of the 26th International Conference on World Wide Web, pp. 1451–1460 (2017)
24. Xu, J., Wickramarathne, T.L., Chawla, N.V.: Representing higher-order dependencies in networks. Sci. Adv. **2**(5), e1600028 (2016)
25. Yin, H., Benson, A.R., Leskovec, J., Gleich, D.F.: Local higher-order graph clustering. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 555–564. ACM (2017)