

Chapter 1

BuildHON²: A Scalable Method for Growing Higher Order Networks

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 represented by traditional network methods, which are referred to in this paper as first-order networks. In a first-order network, data is represented as a Markov chain. This means that at a given state (or node) in the network, we assume the Markov property: all necessary information about that state is available locally. As an example, consider two flight sequences. One sequence begins in Detroit and travels through Chicago to Tokyo. Another sequence begins in Los Angeles and travels through Chicago to London. A first-order network, as pictured in Figure 1.1, joins the sequences on Chicago, the node common to both. However, this union results in the loss of valuable information. It is extremely inefficient, and probably quite uncommon, to fly from Los Angeles to Tokyo through Chicago! However, we cannot infer this from the first-order network itself. This is the Markov property in action: the first-order network does not capture the fact that a traveler in Chicago who has just come from Los Angeles is a very unlikely candidate to board a plane to Tokyo.

For many applications which process sequential data, including transportation networks and anomaly detection, a first-order network representation is insufficient. Accurate analysis must take into account an ordered series of events, not just several pairwise sequences joined on common nodes. A higher-order network (HON) representation is a creative solution to this problem that has demonstrated compelling increases in representative accuracy [13]. However, the trade-off for increased accuracy is increased network size and computational cost.

Two key cost components are the time and memory required to process sequential data and generate the network. Even the most accurate network representation model is of no use if it cannot be generated efficiently. While this does not cause serious concern for small data sets, many real-world data sets not small. During tests on the time-shared workers at the Center for Research and Computing (CRC) at the University of Notre Dame, BuildHON+, the state-of-the-art HON generator was unable to process synthetic data sets larger than 100 million items. In this paper I present BuildHON² a method for growing higher order networks at a greatly reduced time and memory cost. The key innovation in BuildHON² is a data structure for efficiently mining sequential rules, which I call an **SqTree** (Sequence Tree). The first set of experiments using BuildHON²

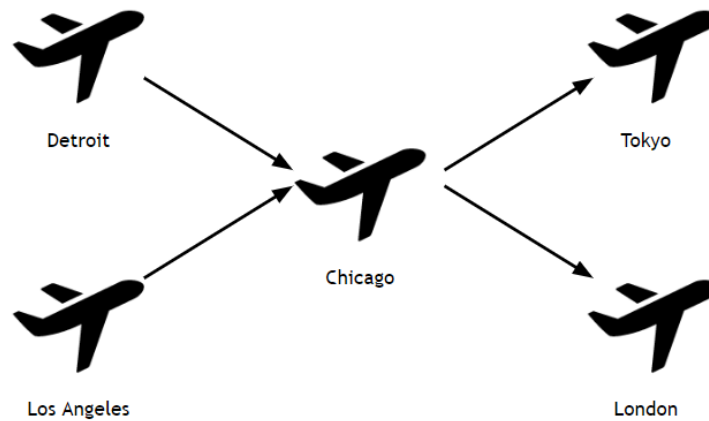


Figure 1.1: An example of a first-order network.

demonstrate superior runtime and memory usage when compared to BuildHON+. BuildHON² also shows great promise in scaling to greater than sequential data sets larger than 1 billion items. My grand vision for BuildHON² is to package the representational benefits of HON in a solution so efficient that it is unquestioningly worth the cost.

1.2 The Problem as a Graph

A HON deals with data representing sequential interactions with multiple-levels of dependencies. A sequential **dependency** refers to any point or node in the sequential data for which the Markov property fails to produce an accurate result. From our example in Figure 1.1, Chicago exhibits a dependency, because accurate processing is dependent on additional information. We can then use these dependencies to generate a set of sequential **rules**, which identify the sequences underlying the network. Figure 1.2 illustrates a more detailed example.

Our sequential data contains 8 sequential records, 3 * (A>M>X), 1 * (A>M>Y), 1 * (B>M>X), and 3 * (B>M>Y). To construct a first-order network representation, we would essentially decompose these 8 records into 16 pairwise sequences: 4 * (A>M), 4 * (B>M), 4 * (M>X), and 4 * (M>Y). A, B, M, X, and Y are added as nodes in the network, and nodes that appear sequentially in the raw data are connected with an edge. Each edge is assigned a weight equal to the frequency with which the sequence appears in the raw data (or, as in Figure 1.2 it can be expressed as a percentage).

Imagine that A, B, M, X, and Y are all cities, and consider that our task is to determine, based on the first-order network, the probability that an airplane beginning at A will travel through M to X. Flying from A to M has a 100% probability, since A does not connect to any other cities. From M, the plane has a 50% chance of flying to X and a 50% chance of moving to Y; however, we can easily infer from the raw data that this is not an accurate result. When A is the original source, our data 75% of the planes will eventually reach X. This is the Markov property inherent in all first-order networks: once the plane reaches M, it "forgets" whether it came from A or B, despite the fact that this information makes a significant difference to the end result.

Solutions to this problem can be generalized as follows:

1. Smarter graph algorithms. We could train an algorithm on the sequential rules, which would enable it to provide more accurate results. However, this approach is developmentally costly

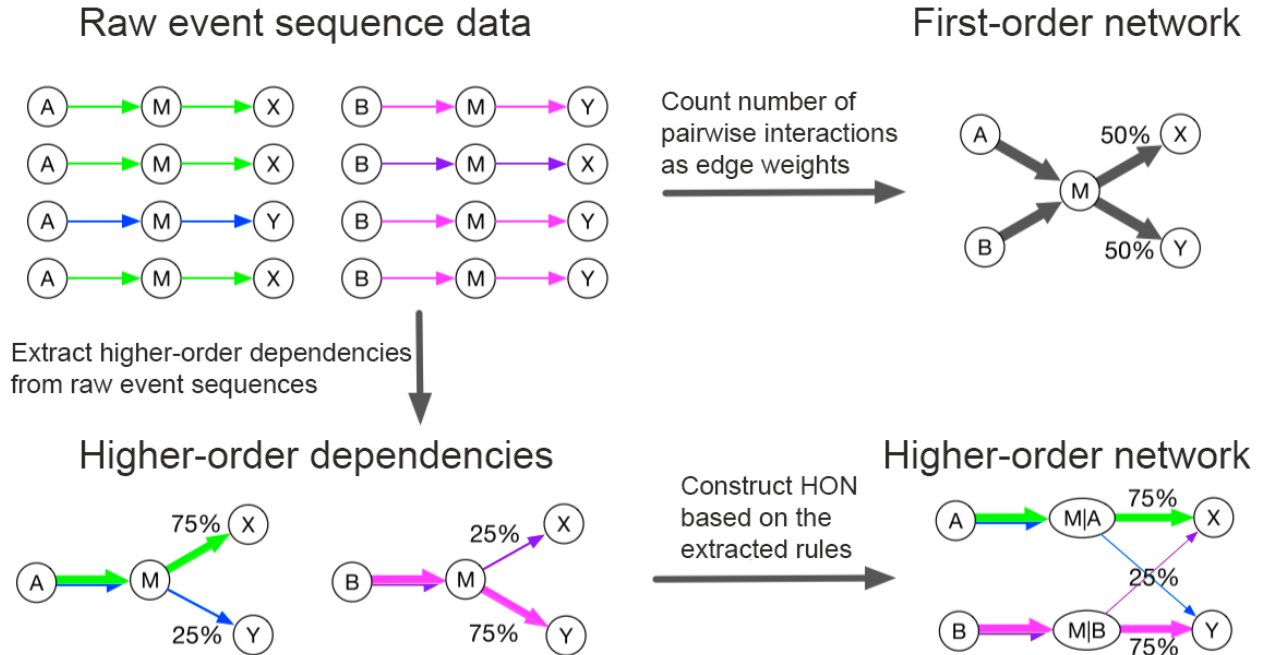


Figure 1.2: An overview of the principle behind the generation of a higher-order network. Figure from [8].

requires separate modifications and training for every graph analysis tool. Rather than a higher-order network solution, we would have a higher-order random walker, higher-order triangle counter, higher-order community detector [1] [9], and more [5].

2. Fixed-order Markov models [11]. In summary, these models optimistically assume sequential dependencies and force a higher-order representation of every interaction. They have demonstrated increased representative accuracy in a number of applications. The main problem with this approach is that the size of the resulting network grows exponentially, meaning analysis of the network will be much more costly. Given a data set in which the number of sequential dependencies is dense, this solution may make sense. However, dependencies in real-world data sets are sparse, so a fixed-order representation will incur severe overhead.
3. Higher-order networks (HON, the subject of this paper). A HON, given the example in Figure 1.2, identifies a dependency at node M, then from the dependency generates a set of sequential rules which correspond more accurately to the raw sequences. Finally, it builds a new network (or rewires the existing one) according to the extracted rules. Unlike fixed-order Markov models, a HON allows for rules of different orders to cohabitate in the network. The **order** of a rule is the count of node prefixes. For example, $M > X$ is a rule with order 1: X, the destination, has one prefix, M. $A > M > X$ is a rule with order 2: X, the destination, has 2 prefixes, A and M. $A > M > X$ is said to be a **higher order** rule with respect to $M > X$, because it shares a common destination but has one additional prefix. Likewise, $M > X$ is a **lower order** rule with respect to $A > M > X$. In a HON, orders can increase to any arbitrary threshold. The major goals of a HON are to embed sequential rules in the network structure such that 1) there is minimal overhead in terms of extra nodes and edges which do not provide useful information, and 2) existing network tools can be used without modification.

1.3 Some Realistic Data Sets

HON is concerned with sequential data that can be represented as a weighted digraph. The two data sets utilized in previous implementations are global shipping routes traversed during several months in 2012 [13], and synthetic clickstream data used for anomaly detection [12]. In this study I utilize the same base synthetic data set [14], and make several extension to it to further test scalability. These data sets are depicted in Figures 1.3 and 1.4. Unfortunately, the global shipping data is not freely available. Instead, I perform some tests using a set of records extracted from the New York City Taxi and Limousine Commission [10]. This data set, pictured in Figure 1.5, represents a series of locations between which taxis were called, and has some noteworthy characteristics. First, due to the nature by which I sequenced the data, the sequences may not be meaningful in terms of representing taxi routes. Second, this network is exceptionally dense, having 97% of all possible edges between nodes. An important item of future work will be to acquire more realistic and representative real-world data sets.





	Data Set	Number of Records	Record Length	Total Size	Unique Nodes	Average Density
	SyntheticFull	10,000	100	1,000,000	100	0.0404
	SyntheticLarge	100,000	100	10,000,000	1,000	0.0040
	SyntheticGiant	1,000,000	100	100,000,000	1,000	0.0040
	SyntheticBalrog	10,000,000	100	1,000,000,000	1,000	0.0040

Figure 1.3: Base synthetic data sets used to evaluate scalability.





	Base Data Set	Number of Records	Record Length	Total Size	Unique Nodes	Average Density
	SyntheticGiant	1,000,000	100	100,000,000	10,000 (x10)	0.0014
	SyntheticGiant	1,000,000	100	100,000,000	100,000 (x100)	0.0005
	SyntheticBalrog	10,000,000	10 (x0.1)	100,000,000	1,000	0.0040
	SyntheticLarge	100,000	1000 (x10)	100,000,000	1,000	0.0456

Figure 1.4: Extended synthetic data sets used to evaluate scalability. Modifications from the base synthetic data are shown in yellow.

1.4 BuildHON² - A Key Graph Kernel

BuildHON² does not fit the category of "graph kernel" in the same sense as random walking or triangle counting. Other kernels typically deal with a particular problem on graph that already


	Data Set	Number of Records	Record Length	Total Size	Unique Nodes	Average Density
	2018 NY Taxi Data	4,306,477	52	223,387,351	266	0.9757

Figure 1.5: New York City taxi data.

exists. BuildHON² is instead concerned with the construction of the graph - a step that necessarily precedes the application of other kernels.

Two prior versions of HON generators exist in the literature: BuildHON [13] and BuildHON+ [12]. Both include 2 major algorithmic components: 1) extracting rules from sequential data, and 2) rewiring the network according to the extracted rules. These are described in Section 1.5. The key innovation in BuildHON² is the SqTree, a tree structure for compactly storing the sequential data. This approach requires an overhaul of both the rule extraction and network rewiring steps. BuildHON²'s approach is detailed in section 1.9.





1.5 Prior and Related Work

Some general approaches to the problem of representing higher-order sequential dependencies are described in Section 1.2. Additional related work includes variable-length Markov chains [2], which, while conceptually very related to HONs, deal mostly with the stochastic process governing the flow of sequential data (i.e. rule extraction). This is an essential piece of HON generation, but stops short of achieving a HON's fundamental goal: accurate network representation.

The first HON generator, BuildHON, includes 2 major algorithmic components: rule extraction and network rewiring [13]. The second generator, BuildHON+, was a modification of the original BuildHON with significantly improved runtime. Its key changes were 1) a lazy version of rule extraction [12] and 2) an "indexing cache" to store and quickly reuse the location of sequences in the raw data. However, BuildHON+ still has time and space complexity $\theta(N(2R_1 + 3R_2 + \dots(i + 1)R_i))$. While not exponential, as the number of nodes N and rules i increases in very large and complex data sets, BuildHON+ will eventually become unusable. In such cases, researchers must make a difficult trade-off between performance and the superior accuracy of a HON representation. A key contributor to BuildHON+'s computational complexity is its reliance on revisiting the raw sequence data to generate higher-order rules. The indexing cache greatly speeds runtime but incurs additional memory overhead.

BuildHON²'s solution to this problem is the SqTree, a data structure inspired by the Frequent Pattern Tree, or FpTree. FpTrees are used in association rule mining to efficiently store transaction data in a tree structure. Association rules can then be mined directly from the tree without revisiting the raw data [6]. FpTrees have proven effective and scalable for a variety of big data problems. Using an FpTree consists of two algorithmic components: tree growth and rule extraction. However, as pictured in Figure 1.6, FpTrees cannot be directly used to generate a HON, because the sequential data with which HON is concerned includes constraints not considered in the FpTree algorithms.

Other solutions for sequential rule mining have been proposed. *MOWCATL* is solution that accounts for time lags, or gaps between related sequences [7]. HON is concerned with immediately sequential items, so *MOWCATL*'s additional constraints are unhelpful to the problem at hand. *RuleGrowth* [4], *ERMiner* [3], and *HUSRM* [15] are recent solutions that also focus on mining rules based on sequences. Perhaps these solutions could also be adapted to a solution for a HON

	Algorithmic Component	Helpful to HON?
	Growth*	
	Extraction**	

* No way to efficiently find rules of higher/lower order

** Extraction does not account for sequential ordering

Figure 1.6: A visualization of FpTree’s algorithmic components and their usefulness for generating a HON.

generator, but for this project I chose a different direction because they assume a different set of constraints on the data. HON is concerned with sequences which are 1) strictly-ordered and 2) single-state. For example, in the airport example from Figure 1.1, a passenger is present in exactly one location, arrived from no more than one immediately preceding location, and will travel to no more than one immediately subsequent location. These other solutions are generally concerned with more generalized sets of sequences. For example, *HUSRM* uses items purchased by customers as a motivating example: if a customer buys milk and bread, how likely is she to also purchase champagne? This example is neither strictly-ordered (a customer could purchase chocolate between the bread and champagne and the rule would still hold) or single-state (a customer can purchase multiple items together). While FpTrees pose similar problems when applied to HON generation, the idea was more easily adapted to the constraints of HON. Future work in synthesizing other solutions could prove fruitful, but such investigation is outside the scope of this paper.

1.6 A Sequential Algorithm

An extremely simplified version of BuildHON+ is shown in Algorithm 1. The detailed pseudocode is publicly available online [14].

Algorithm 1 BuildHON+ Rule Extraction (simplified from [14])

Input: A database of sequential data T

Output: A set of sequential dependencies R , which can easily be used to construct the HON.

```

1: for  $i$  in  $length(T)$  do                                     ▷ Count first order observations
2:   if  $(T_i, T_{i+1})$  not in  $R$  then
3:      $R.append(T_i, T_{i+1})$  as (source, destination)
4:   else
5:      $R[T_i, T_{i+1}] += 1$ 
6:  $EC := R$                                                      ▷ All first order observations are Extension Candidates
7: while  $EC$  do                                             ▷ Continue extracting rules until there are no more candidates
8:   for  $c$  in  $EC$  do
9:      $c_{ext} = get\_extension(c)$                                ▷ Requires revisiting raw sequences
10:    if  $KLD(c_{ext}, c) > THRESHOLD$  then                    ▷ KLD determines if extension is significant
11:       $R.replace(c, c_{ext})$ 
12:       $NEC.append(c_{ext})$                                    ▷ Next Extension Candidates
13:     $NEC := EC$ 

```

A key optimization not described in Algorithm 1 is the use of an "indexing cache" to store the position of items in the raw sequence data. This saves the algorithm from the tremendous overhead of repeated searches through the entire sequential data set. However, it incurs significant memory overhead, because there is an entry for every single set of rules. Additionally, the simple depiction of the algorithm does picture two important parameters. *max_order* provides a cutoff for rule extension, and *min_support* establishes a minimum frequency count for a sequence to be eligible for extension.

1.7 A Reference Sequential Implementation

I implemented a simple version of the Rule Extraction algorithm in C++ using only classes from the C Standard Library. I chose C++ because it offers potentially significant efficiency over the base Python implementation. The C++ 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*. As I will discuss in section 1.9, BuildHON² replaces all of them.

1.8 Sequential Scaling Results

The sequential C++ implementation was disappointing and did not offer significant performance improvements over the Python implementation. For this reason, I have chosen not to include the results of these experiments. Section 1.11 will compare BuildHON²'s performance with the Python version of BuildHON+ as a baseline.

1.9 An Enhanced Algorithm

BuildHON²'s key innovation is its use of an **SqTree** for efficiently storing and processing sequential data. In this section, I describe its algorithmic components: 1) tree growth, and 2) pruning (i.e. rule extraction). The tree growth algorithm is described in Algorithm 2 and depicted in Figure 1.7.

In Figure 1.7, SqTreeGrowth is given a *max_order* of 3 and begins at the first "1" in the first transaction. 1 is added as a child of root (denoted root>1). 2 is then added as a child of

Algorithm 2 SqTreeGrowth**Input:** A database of sequential data T , an integer max_order **Output:** An SpTree SpT with height max_order

```

1:  $SpT.root := null$ 
2: for  $S$  in  $T$  do                                     ▷ For sequence in the database
3:   for  $n$  in  $S$  do                                       ▷ For each node in the sequence
4:     parent :=  $SpT.root$                                      ▷ First parent is root
5:     for  $i := 0$  to  $max\_order$  do
6:       if parent.has_child( $n \gg i$ ) then                 ▷  $n \gg i$  means  $n$  offset by  $i$  positions in  $S$ 
7:         parent.increment_count( $n \gg i$ )
8:       else
9:         parent := add_child(parent,  $n \gg i$ )             ▷ Adds another layer of depth

```

root>1 (denoted root>1>2), and 3 added as a child of root>1>2 (denoted root>1>2>3). Because max_order is 3, the maximum length of a sequence embedded in the tree is 3. The next item is the first "2" in the first transaction. 2 is added as a child to root, 3 as a child of 2, and 1 as a child of 3. The process repeats for the first "3." When the algorithm reaches the second "1" in the first transaction, it finds root>1 already exists, so it does not add another child node. Instead, it increments the branch weight of root>1 to track the pattern's frequency count. That 1 is also followed by a 2, and root>1>2 exists, so its branch weight is also incremented. This process continues until the entire set of transactions has been processed.

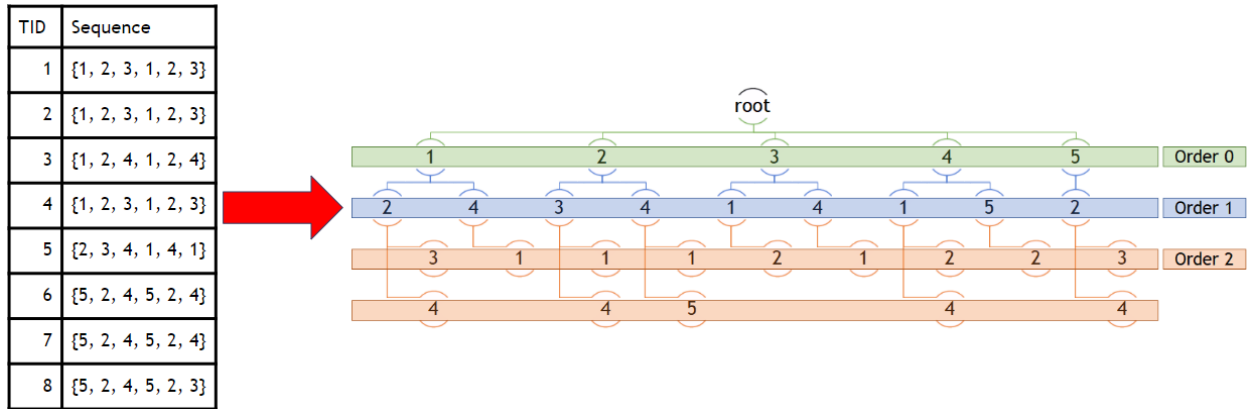


Figure 1.7: The input and output of SqTree's growth algorithm. The input is a series of strictly-ordered sequences in which each item represents a single state. The output is a tree structure, within which are embedded all sequences up to the parameter max_order . Each node represents a rule that is a candidate for extension, and each level of the tree represents all rules of a given order. The branch weights, header table, and cousin links are not pictured.

The pruning part of the algorithm relies on two additional set of links generated during tree growth: the header table and cousin links. The 2D header table stores a reference to the first appearance of each unique node in each level of the tree. This is heavily utilized in the pruning process to quickly locate related rules of a higher or lower order. For example, in determining if rule 1>2>3 has a dependency, we can use the header table to locate sequence 2>3 in $O(1)$ time and compare the confidence of each rule. The cousin links connect all nodes in a given order that share the same node label. These are used during pruning to efficiently iterate over every node in

each level of the tree and to quickly find all rules that terminate at the same node.

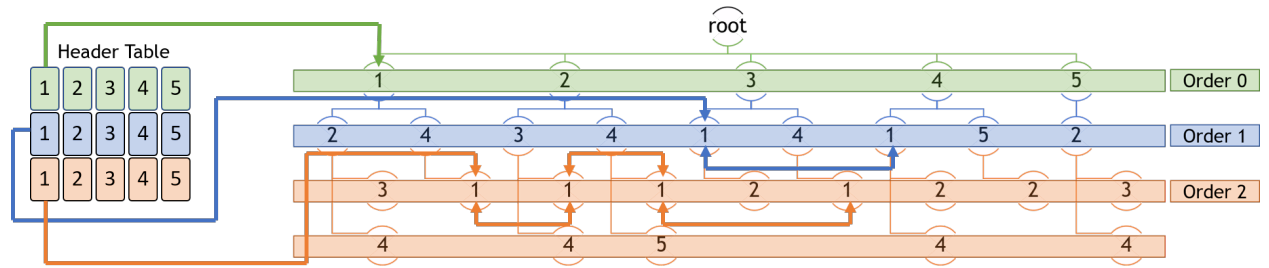


Figure 1.8: The final SqTree, including the 2D header table and a subset of cousin links. The header table is used in the pruning process to quickly locate rules of a given order and destination. The cousin links are used to quickly determine the confidence of a given sequence. Branch weights are not pictured.

The pruning algorithm is currently implemented as shown in Algorithm 2. This method of pruning has the advantage of being done "in place" and thus avoiding the memory cost of building a separate rule tree. However, the algorithm has not yet been perfected, and currently over-prunes parts of the tree. Once the pruning algorithm completes, the SqTree that remains is equivalent to the list of rules that can be used to construct the HON. A simple procedure can then be implemented to construct a HON from the pruned SqTree.



Algorithm 3 SqTreePruning

Input: An SpTree SpT with height max_order and header table ht

Output: A pruned SpTree SpT

- 1: **for** $i := 2$ to max_order **do** ▷ Start at order 2 (3rd level of the tree)
 - 2: **for** n in $ht[i]$ **do**
 - 3: $h_ord_rule := n$
 - 4: **while** h_ord_rule **do** ▷ Check every higher order candidate
 - 5: $l_ord_rule := find_l_rule(h_ord_rule)$
 - 6: **if** $KLD(h_ord_rule, l_ord_rule) > THRESHOLD$
 - 7: $SpT.prune(l_ord_rule)$ ▷ If there is a dependency, prune the lower rule
 - 8: **else**
 - 9: $SpT.prune(h_ord_rule)$ ▷ Otherwise, prune the higher rule
-

The complexity of SqTree's algorithmic components is depicted in Figure 1.9. The single largest factor is the density of the sequential data. This density is equivalent to the density, or the ratio of edges to possible edges, of the first-order representation of the data. This is because each unique first-order (pairwise) sequence exists in the tree at a frequency proportional to the number of prefix combinations in the raw data. More unique first-order sequences means more duplication, especially at higher orders within the tree. The SqTree is thus extremely efficient when the network is sparse, and extremely inefficient when it is dense. Additionally, it should be noted that building the tree bears a far heavier cost with respect to time than the pruning.

	Algorithm	Time	Space
	Growing	$O(k * m)$	$O(n ^ m)$ - dense $O(n * m)$ - sparse
	Pruning	$O(n ^ m)$ - dense $O(n * m)$ - sparse	--

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

Figure 1.9: Complexity analysis of SqTree’s algorithmic components.

1.10 A Reference Enhanced Implementation

BuildHON² is implemented in C++. The implementation contains about 480 lines of code. 300 lines are for the SqTree class, 100 for the SqNode class, and the remaining 80 for the driver, header files, and other definitions. I chose C++ for its efficiency and the availability of pointers, which are well-suited to implementing tree structures. The implementation relies heavily on features introduced in C++, such as Standard Template Library’s unordered map (hash table) and iterator implementations. The driver code is shown in Code Segment 1.2.

Code Segment 1.2: SqTreeDriver.cpp

```

1  int main(int argc, char** argv) {
2      string inf_path = (argc > 1) ? DEF_INF_PATH : DEF_INF_PATH;
3      int num_seq_prefixes = argc > 2 ? stoi(argv[2]) : DEF_NUM_SEQ_PREFIXES;
4      int max_order = argc >= 3 ? stoi(argv[3]) : DEF_MAX_ORDER;
5      bool verbose = argc >= 4 ? stoi(argv[4]) : DEF_VERBOSE;
6
7      SqTree sqt = SqTree(inf_path, num_seq_prefixes, max_order, verbose);
8      sqt.prune();
9      sqt.dump_rules();
10 }
```

The driver function is quite simple. The implementation allows a user to provide up to 4 runtime parameters: location of the input data, number of prefixes in each sequence (used to filter out the sequence ID), max order, and an option for verbose logging. After setting the runtime parameters, the driver initializes, prunes, and dumps the SqTree.

1.11 Enhanced Scaling Results

To test the scalability of the SqTree, I ran a series of experiments. Each experiment was conducted as a series of 10 runs on a worker machine in the Center for Research and Computing at the University of Notre Dame. Each run was carried out by a Lenovo NeXtScale nx360 M5 server

BuildHON²

using dual 12 core Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz Haswell processors and 64 GB RAM. These resources are time-shared, which may impact overall performance.

The entire experimental series is depicted in Figure 1.10. The first two rows are the baseline comparison between the Python implementation of BuildHON+ and the C++ implementation of BuildHON². Both use the same sets of synthetic data described in Figure 1.3. The third row tests the performance of BuildHON² against the modified synthetic data described in Figure 1.4.





	Application	Data Collection	Runtime Parameters	Iterations
	BuildHON ² (CHON)	Base Synthetic	max_order = 5	40 (10 runs * 4 data sets)
	BuildHON+ (Python)	Base Synthetic	max_order = 99	40 (10 runs * 4 data sets)
	BuildHON ² (CHON)	Extended Synthetic	max_order = 5	40 (10 runs * 4 data sets)
	BuildHON ² (CHON)	SyntheticGiant only	max_order = [3, 4, 5, 6, 7, 8, 9]	70 (10 runs * 7 parameters)

Figure 1.10: The list of experiments performed to test the performance and scalability of BuildHON².

The baseline comparison of BuildHON+ and BuildHON²(CHON) is shown in Figure 1.11. BuildHON² shows superior scaling in terms of wallclock runtime. It also demonstrates excellent memory efficiency, using a peak 200MB of virtual memory when processing a data set of 1 billion items. BuildHON+, when tasked with the set of 1 billion items, consistently exceeded available memory and was killed by the cluster manager after an average wallclock time of 30 minutes.

The second set of experiments using synthetic data, shown in Figure 1.12, involved varying the number of unique nodes in the raw data while keeping the total size and length of each sequential record fixed. The variance between scaling from 10k to 100k unique nodes is likely due to the difference in density of the synthetic data. In these scenarios, BuildHON² scales linearly and in line with the theoretical complexity.

The third set of experiments using synthetic data, shown in Figure 1.13, involved varying the length of each sequential record while keeping the total size and number of unique nodes fixed. The difference in results between data sets is much more likely attributable to the density of the network than the sequence length itself.

The fourth and final set of experiments using synthetic data, shown in Figure 1.14, involved varying the max order while using the same set of data. These results scale in linear fashion and exactly according to the theoretical complexity when we include the density factor.

The final experiment was performed using the NY Taxi data set [10]. This experiment revealed a key problem in this implementation of BuildHON²: after a certain point in execution, the processing speed slows dramatically. The first set of runs, with *max_order* = 3, completed quickly. However, all other *max_order* values experienced the slowdown pictured in Figure 1.15. I believe this problem is implementation-specific, and not inherent in the SqTree algorithm. The suspects are 1) hashing collisions and 2) excessive memory reallocation, both related to C++11's unordered maps. The total memory usage at this point in execution was still low, so thrashing is an unlikely cause. If this problem was inherent in the SqTree algorithm, I would expect the trend to scale upward much more slowly, rather than take a sharp, almost-90-degree turn upward.

1.12 Conclusion

A higher-order network is an extremely useful tool for representing and analyzing sequential data as a network. Traditional, or first-order, network representations cannot preserve important dependencies that often exist in real-world sequential data. Higher-order networks tackle the problem of sequential dependencies by seeking to embed those dependencies in the structure of the network itself, so that existing network applications can be utilized without modification. However, this process is not free, and BuildHON+, the state of the art HON generator, does not scale well for big data. This paper introduced BuildHON², which, while still in its infancy, has already demonstrated superior performance to BuildHON+ in terms of runtime and memory usage. In experiments performed with the synthetic data with which BuildHON+ was originally developed, BuildHON² performed over an order of magnitude faster with up to several orders of magnitude less memory utilization. This is due to an innovative data structure, the **SqTree**, which builds a tree from sequential patterns in a process somewhat similar to an FpTree. My future work with BuildHON² will likely include the following:

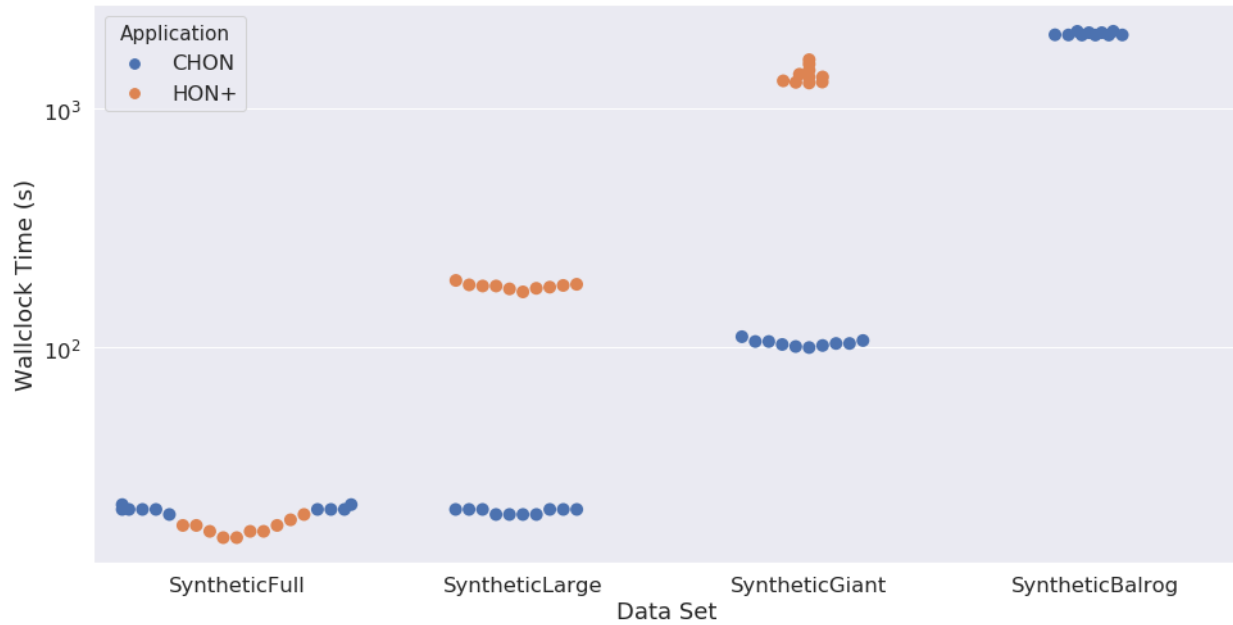
- Diagnose the runtime slowdown discovered in the NY Taxi data experiment. I hypothesize this is specific to the C++11 implementation, not the BuildHON² algorithm itself. If the slowdown is caused by C++'s unordered maps, the fix could be as simple as implementing a different hashing function (to avoid collisions) or intelligently reserving memory at the beginning of program execution.
- Improve the accuracy of the pruning algorithm. My current implementation of the pruning algorithm over-prunes the tree. This is essential to application's utility, but should not significantly affect performance or scaling.
- Vary the density of test data. The density of sequential data has the most significant effect on BuildHON²'s theoretical complexity. While some of the synthetic data sets used in the experiments had different densities as a consequence of modification to other attributes, this was not a sufficient test of BuildHON²'s ability to scale to dense data.
- Parallelize the algorithmic components of the SqTree. A divide-and-conquer approach could enable an SqTree to process trillions of sequential records with efficiency, and perhaps set the stage for the HON to be analyzed by distributed graph engines with even greater ease.

1.13 Response to Reviews

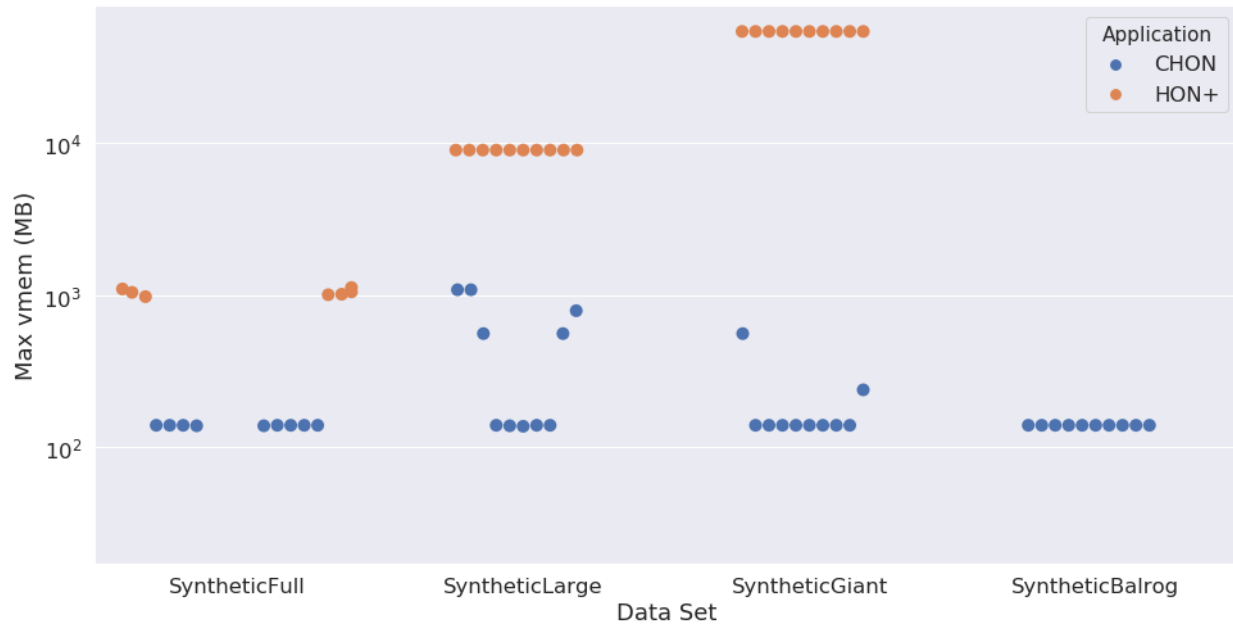
Many of the issues with my first and second drafts were naturally addressed by a reworking of the algorithm and data sets for this final version.

The most common feedback was that some of the terminology and motivation was not clear. I clarified several definitions, including Markov chains, higher-order dependencies, and removed the references to random walkers. I clarified the writing in the "As Graph" section and included an additional example of a first-order network in the introduction.

Previous reviewers indicated the BuildHON+ algorithm was unclear, so I removed the detailed version and replaced it with a simplified version. This is not the focus of my enhanced implementation anyway, so it was a sensible change for the final paper.



(a) Values on the y-axis measure wallclock execution time, in log scale, for each experiment.



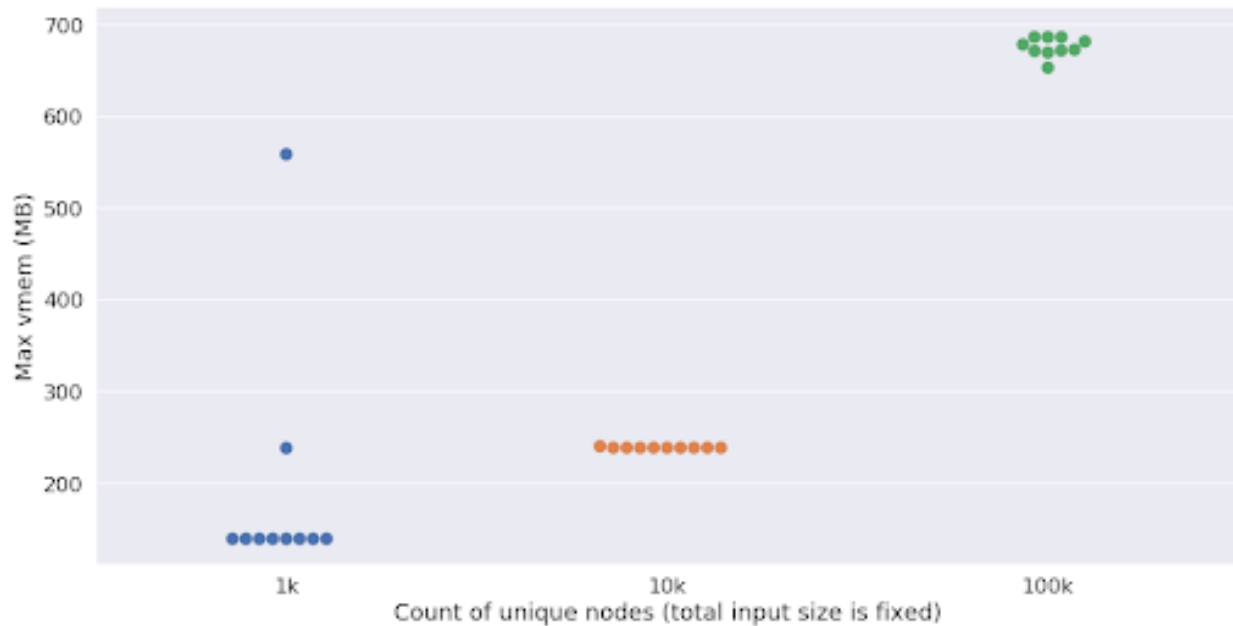
(b) Values on the y-axis measure peak virtual memory usage, in log scale, for each experiment.

Figure 1.11: A baseline comparison of the Python implementation of BuildHON+ and the C++ implementation of BuildHON² (CHON). Each point represents one run of the application on a CRC worker. The categories on the x-axis are synthetic data sets. BuildHON+'s process was killed in every iteration of the SyntheticBalrog data set.

BuildHON²

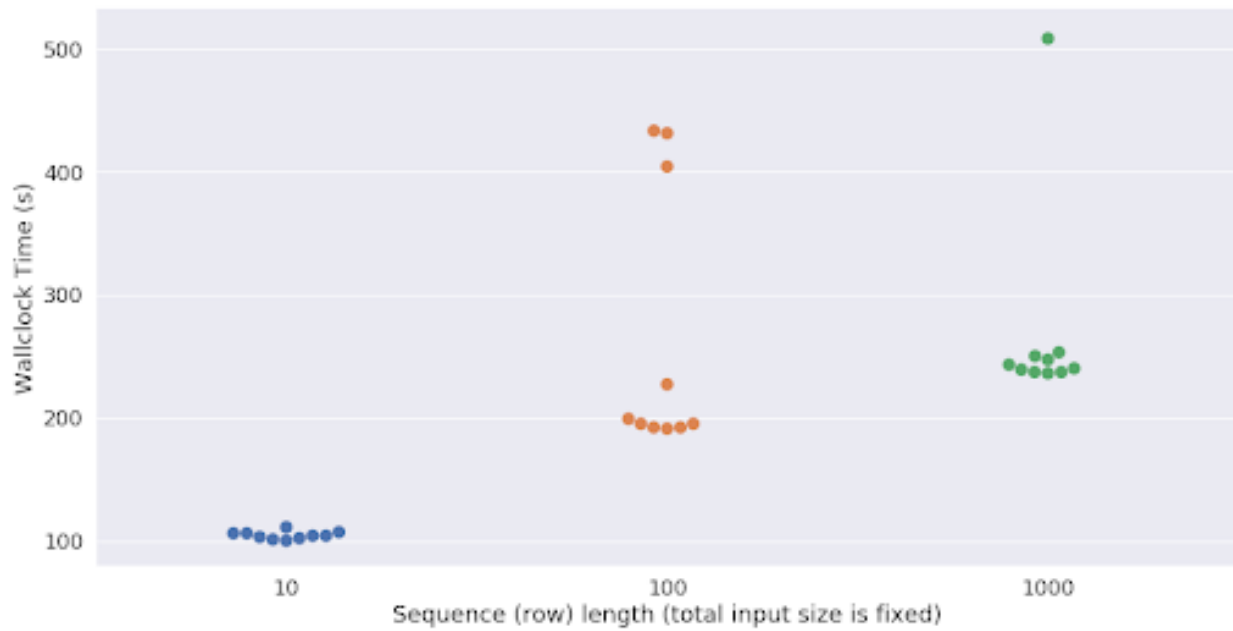


(a) Values on the y-axis measure wallclock execution time, in normal scale, for each experiment.

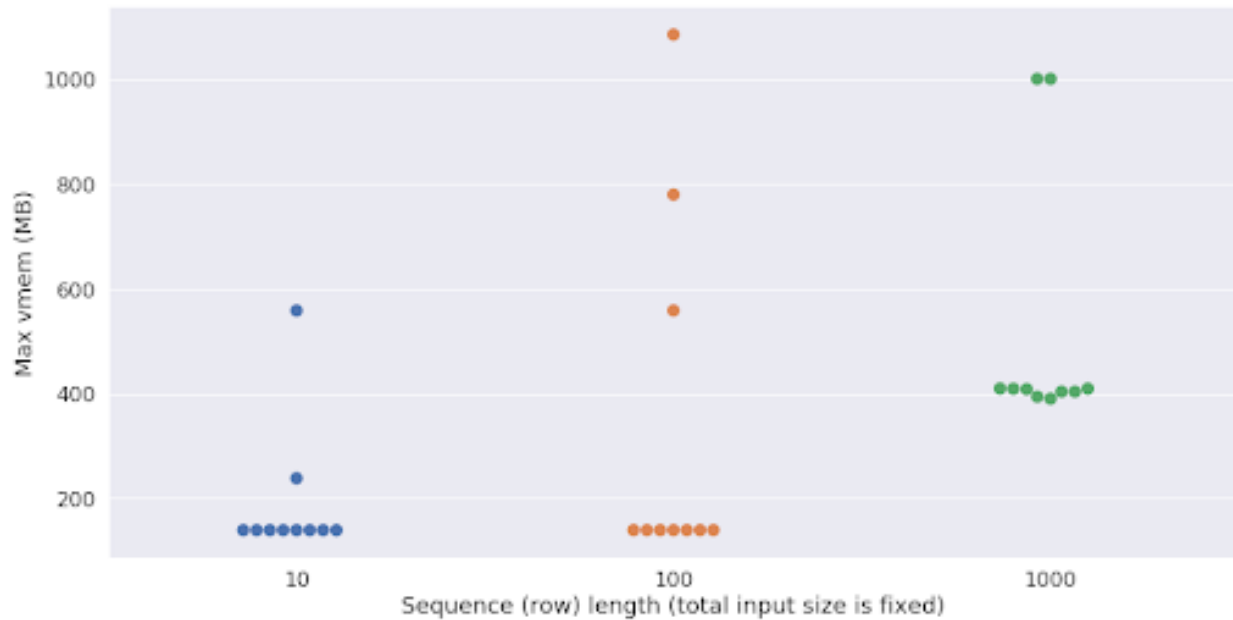


(b) Values on the y-axis measure peak virtual memory usage, in normal scale, for each experiment.

Figure 1.12: Results of using BuildHON² (CHON) to process data sets with different numbers of unique nodes. Each point represents one run of the application on a CRC worker. The categories on the x-axis are synthetic data sets.

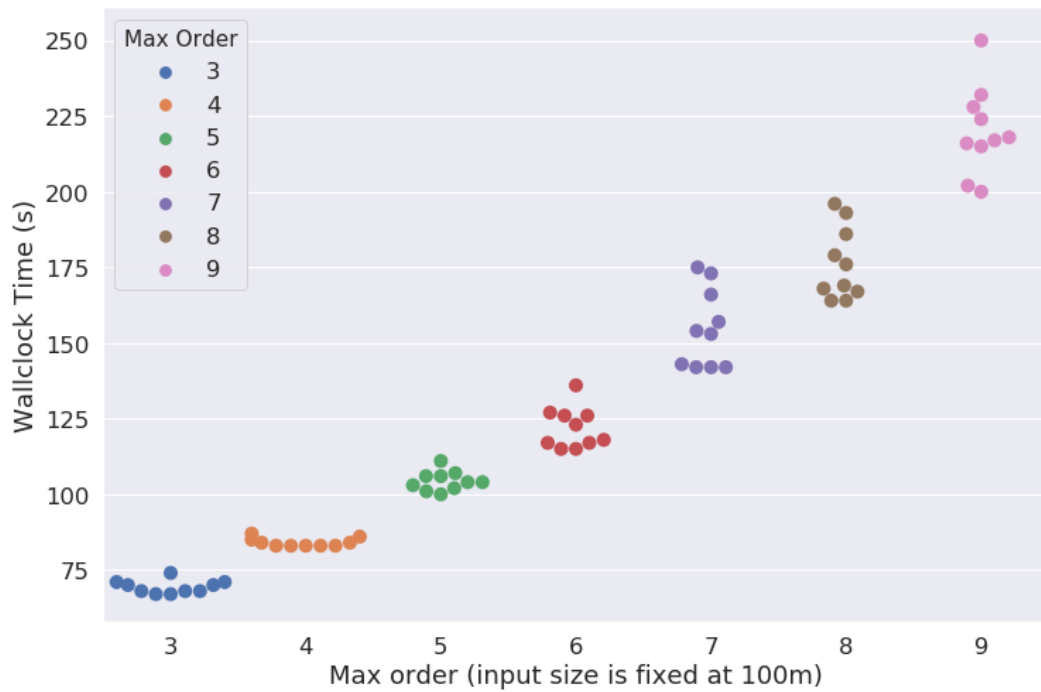


(a) Values on the y-axis measure wallclock execution time, in normal scale, for each experiment.

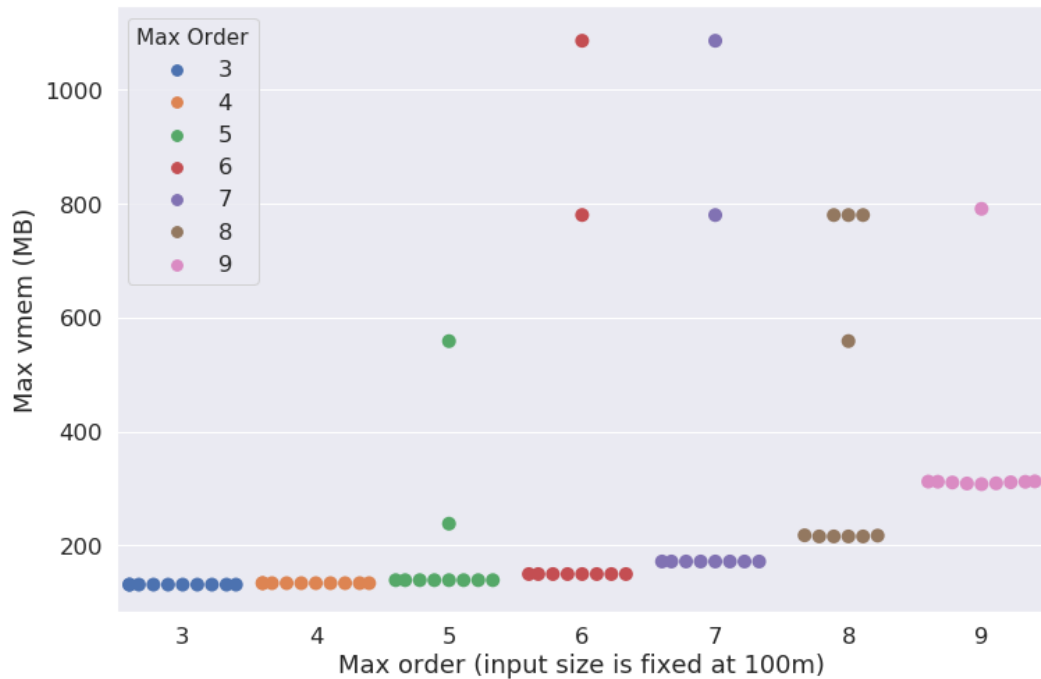


(b) Values on the y-axis measure peak virtual memory usage, in normal scale, for each experiment.

Figure 1.13: Results of using BuildHON² (CHON) to process data sets with different sequence lengths. Each point represents one run of the application on a CRC worker. The categories on the x-axis are synthetic data sets.



(a) Values on the y-axis measure wallclock execution time, in normal scale, for each experiment.



(b) Values on the y-axis measure peak virtual memory usage, in normal scale, for each experiment.

Figure 1.14: Results of using BuildHON² (CHON) to process data sets using a different value for the parameter *max_order*. Each point represents one run of the application on a CRC worker. The categories on the x-axis are the results for a given order. All experiments are run using the base **SyntheticGiant** data set.

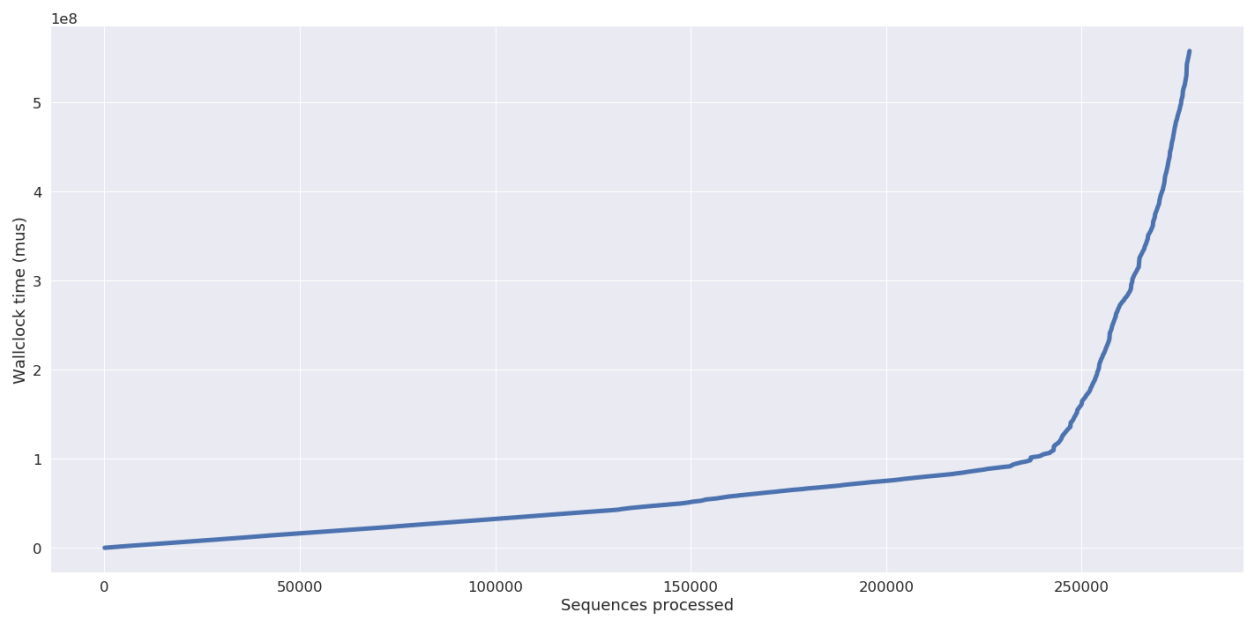


Figure 1.15: A trendline showing the progress made by BuildHON² over time while processing the NY Taxi data. A sharp increase in sequences processed per microsecond appears toward the right side of the graph.

Bibliography

- [1] Austin R Benson, David F Gleich, and Jure Leskovec. Higher-order organization of complex networks. *Science*, 353(6295):163–166, 2016.
- [2] Peter Bühlmann, Abraham J Wyner, et al. Variable length markov chains. *The Annals of Statistics*, 27(2):480–513, 1999.
- [3] Philippe Fournier-Viger, Ted Gueniche, Souleymane Zida, and Vincent S Tseng. Erminer: sequential rule mining using equivalence classes. In *International Symposium on Intelligent Data Analysis*, pages 108–119. Springer, 2014.
- [4] Philippe Fournier-Viger, Roger Nkambou, and Vincent Shin-Mu Tseng. Rulegrowth: mining sequential rules common to several sequences by pattern-growth. In *Proceedings of the 2011 ACM symposium on applied computing*, pages 956–961. ACM, 2011.
- [5] David F Gleich, Lek-Heng Lim, and Yongyang Yu. Multilinear pagerank. *SIAM Journal on Matrix Analysis and Applications*, 36(4):1507–1541, 2015.
- [6] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *ACM sigmod record*, volume 29, pages 1–12. ACM, 2000.
- [7] Sherri K Harms and Jitender S Deogun. Sequential association rule mining with time lags. *Journal of Intelligent Information Systems*, 22(1):7–22, 2004.
- [8] iCeNSA. Higher Order Networks. <http://www.higherordernetwork.com/>, 2018 (accessed Sept. 30, 2018).
- [9] Christine Klymko, David Gleich, and Tamara G Kolda. Using triangles to improve community detection in directed networks. *arXiv preprint arXiv:1404.5874*, 2014.
- [10] The City of New York. TLC Trip Record Data. <http://www.nyc.gov/html/tlc>, 2018 (accessed Dec. 3, 2018).
- [11] Martin Rosvall, Alcides V Esquivel, Andrea Lancichinetti, Jevin D West, and Renaud Lambiotte. Memory in network flows and its effects on spreading dynamics and community detection. *Nature communications*, 5:4630, 2014.
- [12] 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.
- [13] Jian Xu, Thanuka L Wickramaratne, and Nitesh V Chawla. Representing higher-order dependencies in networks. *Science advances*, 2(5):e1600028, 2016.

- [14] Jian Xu, Thanuka L. Wickramaratne, and Nitesh V. Chawla. Higher Order Networks Repository, 2018 (accessed Sept. 30, 2018).
- [15] Souleymane Zida, Philippe Fournier-Viger, Cheng-Wei Wu, Jerry Chun-Wei Lin, and Vincent S Tseng. Efficient mining of high-utility sequential rules. In *International Workshop on Machine Learning and Data Mining in Pattern Recognition*, pages 157–171. Springer, 2015.