Supplementary materials

Algorithm. We present the parameter-free and scalable `BuildHON+` algorithm for constructing HON. The rule extraction step is given in Algorithm 1, and the network wiring step remains the same as that of HON in [2]. While `BuildHON+` algorithm is parameter-free, we provide *MaxOrder* and *MinSupport* as optional parameters. We also provide the optional *ThresholdMultiplier* parameter (the default value 1 is consistent with the HON algorithm), for users to control how aggressive the algorithm prevents higher-order dependencies from being generated. Setting the parameter larger than 1 results in less higher-order dependencies, smaller than 1 for more higher-order dependencies.

Algorithm 1 `BuildHON+` 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 2 *(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 ∅

function ExtendObservation(Source)
    if length(Source) > 1 then
        if not Source[1 :] in ExtC or ExtC[Source] = ∅ then
            ExtendObservation(Source[1 :])
    order = length(Source)
    define ExtC as nested counter
    for Tindex, index in StartingPoints[Source] do
        if index − 1 ≤ 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 = ∅ 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 = ∑ 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] = ∅
                    if not NewOrder in SourceToExtSource[curr] then
                        SourceToExtSource[curr][NewOrder] = ∅
                    SourceToExtSource[curr][NewOrder].add(source)
```

MinSupport	1	2	3	4	5
Precision	0.667	0.5	0.333	0.167	0.167
Recall	1	0.5	0.5	0.5	0.5

Table 2 Precision and recall values for the anomaly detection task using BuildHON with different values of *MinSupport*.

Effect of *MinSupport*. In this section, we discuss the effect of *MinSupport* parameter (in `BuildHON`) on the anomaly detection performance. We want to answer the following question: Can we find an optimal value for *MinSupport* using parameter sweeping for `BuildHON`? Here we show that any values higher than 1 result in lower performance for the anomaly detection using `BuildHON`. As a result, all the experiments in the manuscript use *MinSupport*=1 for `BuildHON`.

We evaluate the anomaly detection performance on the real-world and synthetic data with different values of *MinSupport* for `BuildHON`. The results for the real-world data are shown in Table 2. We notice that for this data, *MinSupport*=1 was the best value, as a higher value led to a decrease in recall and precision both. The total running time for building the HONs (with different values of *MinSupport*) was not significant (order of a few seconds with parallel processing), as the size of the networks is small.

However, the computational time for building the HONs for the synthetic data was significantly higher. The total running time for the parameter sweeping process was 4.372 hours (using parallel processing). On the other hand, the anomaly detection performance was the same for the different *MinSupport* values we tried (1 - 5). This is expected since the synthetic data does not have any noisy patterns.

Note that, *MinSupport* is a global threshold that applies to all paths regardless of order or support. This threshold should have the ability to vary by path/order/support. For example, we may want to preserve paths that their distribution is very different from their lower-order variant, despite having smaller support. Parameter sweeping for *MinSupport* does not necessarily help to preserve significant rules while pruning noise, because it might prune lower orders too aggressively or prune higher-order too mildly. It can also be too aggressive for significantly different paths, or too mild for similar paths.