Generating CP-Nets Uniformly at Random

Thomas E. Allen
University of Kentucky
Lexington, Kentucky, USA
teal223@g.uky.edu

Judy Goldsmith
University of Kentucky
Lexington, Kentucky, USA
goldsmit@cs.uky.edu

Hayden E. Justice
The Gatton Academy, WKU
Bowling Green, Kentucky, USA
hayden.justice259@topper.wku.edu

Nicholas Mattei
Data61 and UNSW
Sydney, Australia
nicholas.mattei@nicta.com.au

Kayla Raines
University of Kentucky
Lexington, Kentucky, USA
kayla.raines@live.com

Abstract

Conditional preference networks (CP-nets) are a commonly studied compact formalism for modeling preferences. To study the properties of CP-nets or the performance of CP-net algorithms on average, one needs to generate CP-nets in an equiprobable manner. We discuss common problems with naïve generation, including sampling bias, which invalidates the base assumptions of many statistical tests and can undermine the results of an experimental study. We provide a novel algorithm for provably generating acyclic CP-nets uniformly at random. Our method is computationally efficient and allows for multi-valued domains and arbitrary bounds on the indegree in the dependency graph.

1 Introduction

Modeling, capturing, and reasoning with preferences is a fundamental topic that spans artificial intelligence, including constraint programming (Rossi, Venable, and Walsh 2011), social choice (Chevaleyre et al. 2008), recommendation systems (Ricci et al. 2011), machine learning (Fürnkranz and Hüllrermeier 2010), multi-agent systems (Goldsmith and Junker 2009), and other fundamental areas. One of the most commonly studied preference models is the conditional preference network (CP-net) (Boutilier et al. 2004).

CP-nets are a factored, compact, and qualitative representation used to model, elicit, and reason about preferences. CP-nets have garnered considerable attention, particularly within the preference handling community (Domshlak et al. 2011). CP-nets have many potential and important applications—automated negotiation (Aydogan et al. 2013), interest-matching in social networks (Wicker and Doyle 2007), cybersecurity (Bistarelli, Fioravanti, and Peretti 2007), and as aggregation primitives for making group decisions (Lang and Xia 2009; Mattei et al. 2013; Xia, Conitzer, and Lang 2011), to name a few. One explanation for the popularity of CP-nets is their seemingly intuitive and visual representation of the language many of us use to describe what we want. For example, the CP-net below displays the statement, “If the weather is fair, I prefer to go cycling, but if it is raining, I’d rather play table tennis.”

Methods for generating random data have long been of interest to computer scientists—Alan Turing advocated for a random number generator in the 1951 Ferranti Mark I computer (Knuth 1997)—and continue to be an active topic of research. Random generation not only of numbers, but of combinatorial objects such as spanning trees and paths in directed graphs have been studied across both mathematics and computer science (Kulkarni 1990). To our knowledge, methods for generating complex preference models such as CP-nets in a uniform manner have not yet received attention.

There is considerable value in being able to generate CP-nets uniformly at random, including: enabling experimental analysis of CP-net reasoning algorithms, unbiased blackbox testing, effective Monte Carlo algorithms, analysis of all CP-nets to better understand their properties, and simulations for decision making or social choice experiments. Complementing theoretical results with empirical experiment, whether from real data or data generated according to a distribution, may provide a window into feasible algorithms that provide good results in practice; biased generation may heavily skew these results.

Experimental research in preference handling requires the use of real-world or simulated data. Real-world data are often messy, not openly available, notoriously difficult to collect reliably, hard to interpret, and nonexistent for CP-nets (Allen et al. 2015; Mattei and Walsh 2013). Principled methods exist to generate simulated data in social choice and preference handling using generative cultures (Berg 1985; Walsh 2011; Mattei, Forshee, and Goldsmith 2012). Such cultures have their drawbacks and limitations (Regenwetter et al. 2006; Popova, Regenwetter, and Mattei 2013), but provide a first step in experimentation for fields where data are hard to gather. While generative cultures over strict, linear
orders are well defined in social choice, there is not an anal-
log for preferences over more complex structures such as
CP-nets. To generalize any statistical cultures used in social
choice, we need to be able to generate samples uniformly at
random from a specified set of CP-nets.

A key idea of our method is that the structure of a CP-
net is equivalent to a tuple of sets representing the parents
of nodes in the network. We show how to enumerate all
such dagcodes, as these tuples are known, and how to cal-
culate the number of CP-nets—the possible graphs and condi-
tional preference tables (CPTs)—that extend a partial dag-
code. The resulting novel recurrence allows us to generate
the graph and CPTs, node by node, such that all acyclic CP-

We first formalize definitions that we need to discuss CP-
nets and random generation, highlighting two problems—
bias and degeneracy—that result from commonly used naïve
generation methods. We show how to encode and count the
dependency graphs. We next show how to avoid degeneracy
in the CPTs and extend the recurrence to count all combina-
tions of CPTs that a partially specified CP-net can have. Fi-

3 Naïve Generation, Bias, and Degeneracy

If one wants to generate CP-nets without regard for the re-
sulting distribution, many simple random methods exist. For
example, initialize a CP-net with \( n \) nodes, no edges, and
empty CPTs; choose a random subset of pairs \((X_h, X_i)\),
h < i, inserting an edge from each \( X_h \) to \( X_i \); generate a
CPT for each \( X_i \) with \( d^{\mid Pa(X_i)\mid} \) rules, each a random per-
mutation of the \( d \) values of \( X_i \); and randomly permute the \( n \)
lables. We suspect that something along these lines is meant
when we read in papers, “We generated 1000 CP-nets at ran-
dom.” However, the resulting distribution is biased statisti-
cally, and this bias calls into question the validity of experi-
ments and the ensuing analysis of algorithms and methods.

To understand this bias, consider the following depen-
dency graphs and their associated CP-net count (for \( d = 2 \),

\[ E \rightarrow B \rightarrow A \rightarrow D \rightarrow C \]

32 CP-nets

\[ E' \rightarrow C' \]

1,033,504 CP-nets

Observe that for the chain-shaped graph on the left, there are
just two ways to choose each of the \( n \) CPTs such that they
are consistent with the dependency graph. The CPT of root
\( E \) could be \([e_1 \succ e_2]\) or \([e_2 \succ e_1]\). The other nodes, each
of which has only one parent, also have two possibilities for
their CPT; e.g., \( \text{CPT}(A) \) could be \([b_1 : a_1 \succ a_2, b_2 : a_2 \succ a_1]\)
or \([b_1 : a_2 \succ a_1, b_2 : a_1 \succ a_2]\). However, in the case
of the star-shaped graph on the right, \( \text{CPT}(A') \) has \( d^3 = 16 \)
rules, each with \( d! = 2 \) possible orderings. In all, over one
million CP-nets have the graph on the right, while only 32
have the graph on the left. Further observe that the ratio of
this imbalance increases with the domain size \( d \). Thus, if
the algorithm above in fact generated the two graphs with
equal likelihood, it would grossly oversample CP-nets with
the first graph, while correspondingly undersampling those
with the second.

However, the naïve algorithm does not even generate the
two DAGs with equal likelihood. Since there are \( 5! = 120 \)
ways to permute the labels of the first DAG, but only 5 ways
to permute those of the second, the star-shaped DAG on the
right would be generated 24 times as often as the chain-
shaped DAG on the left. Despite this, the CP-nets in the star-
shaped case would still be greatly undersampled.

A separate problem, degeneracy, can arise in assigning the
CPTs, regardless of how the DAGs are generated.
Example 3. Consider the following CP-net.

The edge from D to A indicates that the preference over the values of A depends on the value of D. However, in examining the CPT of A closely, one can observe that the preference over A does not in fact depend on D. It can thus be represented by the following, simpler CP-net.

Understanding when a CPT is degenerate is crucial to generating CP-nets uniformly at random.

4 Encoding and Counting Graphs

To facilitate unbiased generation, we model the dependency graphs of CP-nets as dagcodes (Steinsky 2003), inspired by Prüfer codes for labeled trees (Kreher and Stinson 1999). We first treat the dagcode as an abstraction and then show how it relates to the dependency graph.

Definition 4. A dagcode $A = \langle A_1, \ldots, A_{n-1} \rangle$ is a tuple of $n-1$ subsets $A_j \subseteq \{1, \ldots, n\}$ that satisfy the cardinality constraint $|\bigcup_{k \leq j} A_k| \leq j$ for all $j$, $1 \leq j < n$.

Observe from Def. 4 that tuples $\langle \{1\}, \{1,3\} \rangle$ and $\langle \{3\}, \emptyset \rangle$ are valid dagcodes ($n = 3$), but $\langle \{1,2\}, \emptyset \rangle$ and $\langle \emptyset, \{1,2,3\} \rangle$ are not, since each violates the cardinality constraint. Steinsky (2003) proved that dagcodes correspond one-to-one with DAGs and described efficient algorithms for converting dagcodes to DAGs (see Alg. 1) and vice versa.

Applied to CP-nets, each subset $A_j \subseteq \{1, \ldots, n\}$ in the dagcode corresponds to the parents of some node $X_i$ in the dependency graph: i.e., $h \in A_j \iff X_h \in Pa(X_i)$. Note that the root node with the smallest label is implicit; informally, it is helpful to consider every dagcode as having an implicit element $A_0 = \emptyset$. The order in which the remaining $n-1$ parent sets $Pa(X_i)$ occur in the dagcode depends on the order of the child node $X_i$ with respect to other nodes in the graph and the relative size of node label $i$, as follows:

1. If $X_i$ is an ancestor of $X_h$ in the DAG, the encoded parent set $Pa(X_h)$ is ordered before $Pa(X_i)$ in the dagcode.
2. If $h < i$ and $X_h$ is neither an ancestor nor a descendant of $X_i$, then $Pa(X_h)$ is ordered before $Pa(X_i)$.

Example 5. The dagcode $\langle \{1\}, \{1,3\} \rangle$ corresponds to a DAG with $n = 3$ nodes depicted below.

The sets $\{1\}$ and $\{1,3\}$ indicate that one node has parent $X_1$ and another has parents $X_1$ and $X_3$; the third (implicit) node is a root. The mapping from parent sets to their children can be recovered from DAGCODE-TO-DAG (Alg. 1) (Steinsky 2003, adapted) working right to left as follows: $A_2 = \{1,3\}$ corresponds to the parents of $X_2$ since $2$ is the largest unassigned label in $\{1\} \cup \{1,3\}$; $A_1 = \{1\}$ corresponds to the parents of $X_3$ since $3$ is the largest unassigned label not in $\{1\}$. The remaining root node is $X_1$.

Observe that a DAG has bounded indegree $c$ iff $|A_j| \leq c$ for all $A_j$ in the corresponding dagcode: every node $X_i$ in the DAG corresponds to the parent set of an element $A_j$ in the dagcode, with the exception of a root with indegree 0.

Our generation method depends on counting the number of extensions to a partially specified graph. Consider a partial encoding $A_{<j} = \langle \{1\}, \{2\}, \ldots \rangle$ of a graph with $n = 4$ nodes and bound $c = 1$ on indegree. Here the $\emptyset$ could be any subset of $\{1,2,3,4\}$ of cardinality 0 or 1 such that the resulting dagcode is valid, viz., $\emptyset, \{1\}, \{2\}, \{3\}, \{4\}$.

Generalizing, let $A_{<j} = \langle A_1, \ldots, A_{j-1}, \ldots \rangle$ be a partial dagcode for which only elements $A_1$ through $A_{j-1}$ have been specified, such that for all $k$, $1 \leq k < j$, $A_k \subseteq V$, $V = \{1, \ldots, n\}$, $|\bigcup_{k \leq j} A_k| \leq k$, and $|A_j| \leq c$.

Algorithm 2 generates all extensions to $A_{<j}$ by recursively combining $A_{<j}$ with each $A_j$ such that the resulting partial dagcode $A_{<j+1}$ satisfies the constraints on cardinality and indegree. To generate all DAGs with $n$ nodes and bound $c$ on indegree, we call ALL-DAGS($n$, $c$, 1, 0, $\emptyset$, $A_{<1}$).

Theorem 6. ALL-DAGS generates each DAG exactly once.

Proof. (Sketch.) Since dagcodes are in one-to-one correspondence with DAGs (Steinsky 2003, Cor. 1), it suffices to show that each dagcode is generated exactly once. For this we will use the recursion invariant: Each time Line 1 is reached, $A_{<j}$ is valid; that is, for all $k$, $1 \leq k < j$, $|\bigcup_{k \leq j} A_k| \leq k$ and $|A_k| \leq c$. We will show that under this assumption, Alg. 2 generates each $A_j$ such that the invariant holds for $A_{<j+1}$. Base case: Observe that the invariant holds trivially for the empty dagcode $A_{<1} = \langle \emptyset, \ldots \emptyset \rangle$.

Inductive hypothesis: Assume the invariant holds for $A_{<j}$, $1 \leq j < n$. Let $U = \bigcup_{k \leq j} A_k$ and $q = |U|$. Observe that the invariant will also hold for $A_{<j+1}$ as long as we choose $A_j \subseteq V$ such that $|U \cup A_j| \leq j$ and $|A_j| \leq c$. We can select each element of $A_j$ either from $U$ or $\overline{U}$. Let $A_j = S \cup T$, where $S \subseteq U$ and $T \subseteq \overline{U}$. Let $s = |S|$ and $t = |T|$; hence, $0 \leq s \leq q$ and $0 \leq t \leq n - q$. Observe that $|U \cup A_j| \leq j$ iff $q + t \leq j$, and $|A_j| \leq c$ iff $s + t \leq c$. Line 3 iterates over all $(s, t)$ that satisfy these conditions. Lines 4–5, then, iterate over all $A_j = S \cup T$ such that the invariant holds for $A_{<j+1}$. Thus each $A_j$ is generated such that $A_{<j+1}$ is valid. Furthermore, since no pair $(s, t)$ is ever repeated in the outer loop and $S \cap T = \emptyset$, no subset $A_j = S \cup T$ is ever repeated.

Termination: Since $j$ increments with each descent, recursion bottoms out at $j = n$, and a DAG corresponding to fully specified dagcode $A = A_{<n}$ is output. After all valid combinations $\langle A_1, \ldots, A_{n-1} \rangle$ are output, Alg. 2 terminates. □
number of DAGs that is more easily extended to CP-nets than those of Robinson (1973) and Steinsky (2003).

Let \( a_{n,c} \) denote the number of DAGs (resp. dagcodes) with \( n \) nodes and bound \( c \) on indegree. Let \( a_{n,c}(j,q) \), where \( q = |\bigcup_{k<j} A_k| \) denote the number of extensions to a partial dagcode \( A_{<j} \). That is, \( a_{n,c}(j,q) \) is the number of ways to choose the remaining elements \( A_j, \ldots, A_{n-1} \) such that the cardinality and indegree constraints are satisfied.

**Theorem 7.** \( a_{n,c} = a_{n,c}(1,0) \). For \( j = n \), \( a_{n,c}(j,q) = 1 \); for all \( j \), \( 0 < j < n \), \( a_{n,c}(j,q) = \sum_{s \geq 0, t \geq 0, 0 \leq q \leq c} \binom{q}{s} \binom{n-q}{t} a_{n,c}(j+1,q+t) \). (1)

**Proof.** (Sketch) \( \text{Base case} (j = n) \): One DAG is generated at Line 2; hence, \( a_{n,c}(n,q) = 1 \) for all \( q \). \( \text{Inductive hypothesis:} \) Assume \( a_{n,c}(j',q') \) gives the correct count for \( j' > j \) and all \( q' \). We will show that the resulting count for \( a_{n,c}(j,q) \) is also correct. Observe that, whatever the size of set \( U \subset V \), the loop at Line 4 iterates over the \( j \) ways to choose \( s \) elements from \( U \). Similarly, the loop at Line 5 iterates over the \( \binom{n-q}{t} \) ways to choose \( t \) elements from \( U \).

Note that the number of DAGs generated in the body of the outermost loop depends on \( s \) and \( t \), which differ on each iteration. Thus, for all \( (s,t) \) as defined in Line 3, we take the sum of the DAGs generated in the loop body, obtaining the result given in Eq. 1. Finally, observe that all dagcodes parameterized by \( n, c \) extend the fully unspecified dagcode \( A_{<j} = \{ a_1, \ldots, a \} \), for which \( j = 1 \) and \( q = 0 \). Thus, \( a_{n,c} = a_{n,c}(1,0) \).

### 5 Counting and Generating the CPTs

We can generalize the notion of a degenerate CPT introduced in Section 3 with the help of a bijection with discrete multi-valued functions. We model each CPT \( X_i \) as a function \( F_j : \{0, \ldots, d - 1\}^m \rightarrow \{0, \ldots, d! - 1\} \). The inputs correspond to the values of the \( m \) parents of \( X_i \). The outputs correspond to one of the \( d! \) orderings on the domain of \( X_i \).

### Algorithm 1: Generate a DAG from its dagcode

```
DAGCODE-TO-DAG( A )
Input:  dagcode A = (A_1, \ldots, A_{n-1})
Output: corresponding DAG G
1: n ← length(A) + 1
2: Q ← \{1, \ldots, n\}
3: for j ← n - 1 downto 1 do
4:   i ← max(Q \bigcup_{k=1}^{j} A_k)
5:   for all h ∈ A_i do
6:     Q ← Q \{i\}
7:     insert edge to X_i from its parent X_h
8:   Q ← Q
9: output DAG G
```

### Algorithm 2: Generate all DAGs that extend dagcode \( A_{<j} \)

```
ALL-DAGS( n, c, j, q, U, A_{<j} )
Input:  n number of nodes
c bound on indegree
j index of current element A_j
q current value of |U|
U current value of A_1 \cup \cdots \cup A_{j-1}
A_{<j} partial dagcode
1: if j = n then
2:   DAGCODE-TO-DAG(A_{<n}); return
3: for all s, t ≥ 0, s ≤ q, s + t ≤ c, q + t ≤ j do
4:   for all S ⊆ U, |S| = s do
5:     for all T ⊆ U, |T| = t do
6:       A_j ← S ∪ T; include A_j with A_{<j} to form A_{<j}
7:       ALL-DAGS(n, c, j+1, q+t, U ∪ A_j, A_{<j})
```

Observe that if variables are binary \( (d = 2) \), \( F_j \) is a Boolean function. In that case the values \( x_i^h \) and \( x_j^h \) of each parent \( X_h \) can map to 0 and 1 respectively. The two possible linear orders \( x_i^1 > x_i^2 \) and \( x_j^1 > x_j^2 \) can correspond to the outputs 1 and 0. For example, we can model the degenerate CPT of node \( A \) from Example 3 with the following truth table.

<table>
<thead>
<tr>
<th>CPT(A)</th>
<th>In1</th>
<th>In2</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_1d_1 : a_2 &gt; a_1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c_2d_1 : a_2 &gt; a_1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>c_2d_2 : a_1 &gt; a_2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Formally, we say \( F_j(u) \) is vacuous in variable \( u_k \) iff its output never depends on \( u_k \); i.e., for all \( u \in \{0, \ldots, d-1\}^m \),

\[
F_j(u_1, \ldots, u_{k-1}, 0, u_{k+1}, \ldots, u_m) = F_j(u_1, \ldots, u_{k-1}, 1, u_{k+1}, \ldots, u_m) = \cdots = F_j(u_1, \ldots, u_{k-1}, d-1, u_{k+1}, \ldots, u_m).
\]

Function \( F_j \) is degenerate if it is vacuous in a variable; otherwise, it is non-degenerate. By extension we say a CPT is degenerate (resp. vacuous in a parent variable) if function \( F_j \) to which it maps is degenerate (resp. vacuous in an input).

Let \( \phi_d(m) \) be the total number of possible CPTs for a node with \( m \) parents, and let \( \psi_d(m) \) be the number of those that are non-degenerate. First consider binary domains \((d = 2)\). Since CPTs and Boolean functions are in one-to-one correspondence, \( \phi_2(m) \) is equivalent to the number of Boolean functions of \( m \) inputs, and \( \psi_2(m) \) is equivalent to the number of non-degenerate Boolean functions.

Hu (1968, §2, §10) (cf. Harrison 1965, O’Connor 1997) proved that for Boolean functions \( \phi_2(m) = 2^{2^m} \), \( \psi_2(m) = \sum_{k=0}^{m} (-1)^{m-k} \binom{m}{k} 2^k \), and \( \lim_{m \rightarrow \infty} \psi_2(m)/\phi_2(m) = 1 \).

We now generalize these results to domains of size \( d > 1 \).

### Theorem 8. \( \phi_d(m) = d! d^m \)

**Proof.** Each rule of CPT \( X_i \) specifies one of \( d! \) linear orders of Dom(\( X_i \)). The number of rules is \(|\text{Asst}(\text{Pa}(X_i))| = d^m \), where \( m = |\text{Pa}(X_i)| \). Since each rule can be assigned independently, \( \phi_d(m) = d! d^m \).
Theorem 9. $\psi_d(m) = \sum_{k=0}^{m} (-1)^{m-k} \binom{m}{k} d^k$.

Theorem 10. $\lim_{m \to \infty} \psi_d(m)/\phi_d(m) = 1$.

(The proofs of Thms. 9 and 10, omitted here, follow those of Hu (1968) [§2, §10] for Boolean functions.)

Theorem 11. Determining whether a CPT (resp. its corresponding function $F_j$) is degenerate can be conducted in time polynomial in the size of the CPT (resp. domain of $F_j$).

Proof. (Sketch.) Recall that a CPT of $X_i \in V$ is degenerate if it is vacuous in any parent $X_h$. It is vacuous in $X_h$ if CPT($X_i \mid X_h = x_k^h$) = CPT($X_i \mid x_h^h$) for all $x_k^h$, $x_h^h \in \text{Dom}(X_h)$. Note that we can check the CPT of $X_i$ for degeneracy using an algorithm with 4 nested loops: 1. Iterate over each $X_h \in \text{Pa}(X_i)$ to check whether the CPT is vacuous in $X_h$. 2. For each $X_h$, iterate over the $d$ values of $x_k^h \in \text{Dom}(X_h)$. 3. For each $x_k^h \in \text{Dom}(X_h)$, iterate over the $\|\text{Asst}(\text{Pa}(X_i))\|$ $d^{\|\text{Pa}(X_i)\|}$ rules to determine whether CPT($X_i \mid X_h = x_k^h$) = CPT($X_i \mid x_k^h$) in each case. 4. For each rule $\text{Asst}(X_i) \ni X_h$, $u \in \text{Asst}(\text{Pa}(X_i) \setminus \{X_h\})$, iterate over the $d$ values that specify the linear order $\succ^i$ on $X_i$. Let $m$ denote the number of parents of $X_i$. Note that the input is $d^{m+1}$, since the CPT has $d^m$ rules of length $d$. The nested loops require $O(md^{m+2})$ time. Thus, we can check for degeneracy in time polynomial in the size of the input.

Moreover, since CPTs of $d$-ary nodes with $m$ parents correspond one-to-one with functions $F_j : \{0, \ldots, d - 1\}^m \to \{0, \ldots, d! - 1\}$, the proof applies also to the latter.

We can leverage these results to generate non-degenerate CPTs efficiently and uniformly. For tiny values of $d$ and $m$, we can choose uniformly from a modest-sized table of non-degenerate functions (e.g., $\psi_2(4) = 64594$). For larger values, we use rejection sampling, generating a random permutation $\succ^i$ for each assignment to parents and repeating this process in the unlikely event (e.g., $< 0.0001$ for $m > 4$ and very rapidly converging to 0 as $m$ increases) that the result is degenerate. With probability $\psi_d(m)/\phi_d(m)$, asymptotic to 1, we will obtain a non-degenerate CPT on a given attempt.

6 Generating CPTs

We can now extend ALL-DAGS (Alg. 2) to generate ALL-CPT-NETS (Alg. 4). CPTs with the same dependency graph differ if any rule of a CPT differs. To generate all combinations of CPTs, we need only introduce a new innermost loop iterating over the possibilities. Since the dagcode is partial, we do not yet have all of the information we need to construct the CPT; we know the parents, but not the child to which they belong. However, we do have enough information to iterate over the corresponding functions $F_j$, since we know the number of parents (of $A_j = s + t$) and the size ($d$) of every domain, so we do that instead. Each $F_j$ is included in a tuple $F = (F_0, \ldots, F_{n-1})$ that we call a cp-code. (We use $F_{<j}$ and $F_{\leq j}$, analogous to $A_{<j}$ and $A_{\leq j}$, for a partial cp-code.) Since a root node is implicit in the dagcode, $F$ contains an additional element $F_0$ corresponding to that node’s CPT, and we invoke Alg. 2 with $j = 0$ instead of 1. When $j = n$, the encoding is complete: $A$ and $F$ fully and uniquely characterize a CPT-$N$. We call Alg. 3 (cf. Alg. 1) to decode it—the DAG from $A$, the CPTs from $F$.

Theorems 6 and 7 can similarly be extended to CP-nets. Let $a_{n,c,d}$ denote the number of CP-nets with $n$ nodes, bound $c$ on indegree, and domains of size $d$. Let $a_{n,c,d}(j,q) = \sum_{q \geq 0, s \geq 0, s+t \leq c} \binom{n-q}{s} \binom{q-q}{t} \psi_d(s+t) a_{n,c,d}(j+1, q+t)$. (2)
Algorithm 5: Compute tables for uniform CP-net generation

The loop at Line 8 executes \( \psi_d(s+t) \) times. The proofs of Thms. 12–13 are otherwise congruent to those of Thms. 6–7.

Generating all CP-nets is feasible only for small \( n, c, \) and \( d \). To generate larger random instances, we propose an efficient method that relies on Eq. 2. Algorithm 6 generates a dagcode one \( A_j \) at a time, such that all CP-nets (as opposed to DAGs) are equally likely. To satisfy the cardinality constraint, we keep track of node labels \( U = \bigcup_{k<j} A_k \) that already occur in \( A_{<j} \), choosing \( A_j \) from \( U \) and the other \( t \) from \( U^c \), subject to constraints on cardinality and indegree. We also choose a non-degenerate function \( F_j \) for the CPT (see Sec. 5). To avoid bias, we choose \( (s, t) \) such that all extensions to \( A_{<j} \) are equally likely, using a table precomputed by Alg. 5. Finally, we call Alg. 3 to output \( N \).

**Theorem 14.** Algorithm 6 generates each CP-net \( N \) with uniform probability \( P(N) = 1/|a_{n,c,d}|. \)

**Proof.** (Sketch.) Line 1 randomly selects one of the \( \psi_d(0) = d! \) possibilities for the CPT of the root node implicit in \( A \); thus, \( P(F_0) = 1/d! \). Each \( A_j, F_j, 0 < j < n, \) is then generated, conditioned on \( U_j = \bigcup_{k<j} A_k \) and \( q_j = |A_j| \). Line 4 chooses \( s \) and \( t \) with probability

\[
\left( \begin{array}{c} n \\vline \\ \ \ \ \ v_d(s+t) \end{array} \right) \frac{a_{n,c,d}(j+1,q_j+t)}{a_{n,c,d}(j,q_j)}.
\]

Then, given \( s, t, \) and \( U \), Lines 5–10 choose \( S, T, \) and \( F_j \) with probability

\[
\frac{1}{\left( \begin{array}{c} n \\vline \\ \ \ \ v_d(s+t) \end{array} \right)}.
\]

Multiplying Eq. 3 and 4 and simplifying gives us the probability of generating \( A_j \) and \( F_j \) given \( U_j \) in Lines 4–10:

\[
P(A_j, F_j|U_j) = \frac{a_{n,c,d}(j+1,q_j+t)}{a_{n,c,d}(j,q_j)} = \frac{a_{n,c,d}(j+1,q_{j+1})}{a_{n,c,d}(j,q_j)}.
\]

Algorithm 6: Generate a CP-net uniformly at random

since \( q_j + t = q_{j+1} \) for \( j = 1 \) to \( n-1 \) (Line 7).

Since \( A \) and \( F \) uniquely characterize a CP-net, \( P(N) = P(A, F) \). Altogether, iterating through all values of \( j \) in the for loop at Line 3, the probability of generating \( N \) is:

\[
P(N) = P(F_0)P(A_1,F_1|U_1)P(A_2,F_2|U_2) \cdots P(A_{n-1},F_{n-1}|U_{n-1})
\]

\[
= 1 \frac{a_{n,c,d}(2,q_2)}{d!} \frac{a_{n,c,d}(3,q_3)}{a_{n,c,d}(2,q_2)} \cdots \frac{a_{n,c,d}(n,q_n)}{a_{n,c,d}(n-1,q_{n-1})}.
\]

One can use Eq. 2 to verify that \( a_{n,c,d}(0,0) = d!a_{n,c,d}(1,0) \); also, \( q_1 = |\bigcup_{k<j} A_k| = 0 \). We can thus rewrite the first term as \( P(F_0) = 1/d! = a_{n,c,d}(1,q_1)/a_{n,c,d}(0,0) \). Further observe that the numerator of the last term is \( a_{n,c,d}(n,q_n) = 1 \). All terms except the first then cancel out, leaving us with

\[
P(N) = \frac{1}{a_{n,c,d}(0,0)} = \frac{1}{a_{n,c,d}}
\]

which proves our case. \( \square \)

**Theorem 15.** Algorithm 5 runs in time and space polynomial in the number of nodes \( n \).

**Proof.** (Sketch.) Observe that the nested loops are bounded by \( n \). We compute \( a_{n,c,d}(j,q) \) with the help of a table. We need only perform this computation once for each \( j \) and \( q \), and the ranges of \( j \) and \( q \) are similarly bounded by \( n \). \( \square \)

Algorithm 6 is also efficient. Random subset sampling and proportional (i.e., weighted) sampling can be performed efficiently (Bringmann and Panagiotou 2012; Knuth 1997, 3.4.2). The efficiency of rejection sampling (the inner loop) is discussed in Section 5.

**7 Conclusion**

We have presented an efficient and provably uniformly random method for generating CP-nets. The method allows for bounds on indegree and multi-valued domains. The recurrence of Theorem 13 can also be adapted to generate CP-nets.
from other distributions. For example, to generate the DAGs without weighting these by the number of CPT combinations, one can simply remove the $\psi(s + t)$ factor. Similarly, it is possible to generate tree-shaped CP-nets by changing the condition $s + t \leq c$ in Line 4 of Alg. 5 to $s + t = 1$.

We have implemented our method in C++ using the GnuMP library (Granlund et al. 2014), allowing generation of thousands of CP-nets per second. Our code is available at http://cs.uky.edu/~goldsmit/papers/GeneratingCPNetCode.html.

Acknowledgements
We thank Dr. Miroslaw Truszczyński for his suggestions and also the anonymous reviewers for their feedback.

Data61 (formerly known as NICTA) is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References