Causal Bayesian NetworkX

Michael D. Pacer[‡][*]

https://www.youtube.com/watch?v=qWAQgWOD_nA

Abstract—Probabilistic graphical models are useful tools for modeling systems governed by probabilistic structure. *Bayesian networks* are one class of probabilistic graphical model that have proven useful for characterizing both formal systems and for reasoning with those systems. Probabilistic dependencies in Bayesian networks are graphically expressed in terms of directed links from parents to their children. *Casual Bayesian networks* are a generalization of Bayesian networks that allow one to "intervene" and perform "graph surgery" — cutting nodes off from their parents. *Causal theories* are a formal framework for generating causal Bayesian networks.

This report provides a brief introduction to the formal tools needed to comprehend Bayesian networks, including probability theory and graph theory. Then, it describes Bayesian networks and causal Bayesian networks. It introduces some of the most basic functionality of the extensive NetworkX python package for working with complex graphs and networks [HSS08]. I introduce some utilities I have build on top of NetworkX including conditional graph enumeration and sampling from discrete valued Bayesian networks encoded in NetworkX graphs [Pac15]. I call this Causal Bayesian NetworkX, or CBNX. I conclude by introducing a formal framework for generating causal Bayesian networks called theory based causal induction [GT09], out of which these utilities emerged. I discuss the background motivations for frameworks of this sort, their use in computational cognitive science, and the use of computational cognitive science for the machine learning community at large.

Index Terms—probabilistic graphical models, causal theories, Bayesian networks, computational cognitive science, networkx

Introduction and Aims

My first goal in this paper is to provide enough of an introduction to some formal/mathematical tools such that those familiar with `python` and programming more generally will be able to appreciate both why and how one might implement causal Bayesian networks. Especially to exhibit *how*, I have developed parts of a toolkit that allows the creation of these models on top of the `NetworkX` python package:cite:*networkx*. Given the coincidence of the names, it seemed most apt to refer to this toolkit as `Causal Bayesian NetworkX` abbreviated as CBNX[1].

In order to understand the tool-set requires the basics of probabilistic graphical models, which requires understanding some graph theory and some probability theory. The first few pages are devoted to providing necessary background and illustrative cases for conveying that understanding.

* *Corresponding author: mpacer@berkeley.edu*
‡ *University of California at Berkeley*

1. Static code can be found at [Pac15], and the most recent version can be found at CBNX. CBNX is licensed with the BSD 3-clause license.

Notably, contrary to how Bayesian networks are commonly introduced, I say relatively little about inference from observed data. This is intentional, as is this discussion of it. Many of the most trenchant problems with Bayesian networks are found in critiques of their use to infer these networks from observed data. But, many of the aspects of Bayesian networks (especially causal Bayesian networks) that are most useful for thinking about problems of structure and probabilistic relations do not rely on inference from observed data. In fact, I think the immediate focus on inference has greatly hampered widespread understanding of the power and representative capacity of this class of models. Equally – if not more – importantly, I aim to discuss generalizations of Bayesian networks such as those that appear in [GT09], and inference in these cases requires a much longer treatment (if a comprehensive treatment can be provided at all). If you are dissatisfied with this approach and wish to read a more conventional introduction to (causal) Bayesian networks I suggest consulting [Pea00].

The current instantiation of the CBNX toolkit can be seen as consisting of two main parts: graph enumeration/filtering and the storage and use of probabilistic graphical models in a NetworkX compatible format [HSS08].

I focus first on establishing a means of building iterators over sets of directed graphs. I then apply operations to those sets. Beginning with the complete directed graph, we enumerate over the subgraphs of that complete graph and enforce graph theoretic conditions such as acyclicity over the entire graph, guarantees on paths between nodes that are known to be able to communicate with one another, or orphan-hood for individual nodes known to have no parents. We accomplish this by using closures that take graphs as their input along with any explicitly defined arguments needed to define the exact desired conditions.

I then shift focus to a case where there is a specific known directed acyclic graph that is imbued with a simple probabilistic semantics over its nodes and edges, also known as a Bayesian network. I demonstrate how to sample independent trials from these variables in a way consistent with these semantics. I discuss some of the challenges of encoding these semantics in dictionaries as afforded by NetworkX without resorting to `eval` statements.

I conclude by discussing Computational Cognitive Science as it relates to graphical models and machine learning in general. In particular, I will discuss a framework called **theory based causal induction** [GT09], or my preferred term: **causal theories**, which allows for defining problems of causal induction. The perspective expressed in this paper, the associated talk, and the CBNX toolkit developed out of this framework.

Graphical Models

Graphs are defined by a set of nodes ($X, |X| = N$) and a set of edges between those nodes ($E | e \in E \equiv e \in (X \times X)$).

Notes on notation

Nodes: In the examples in CBNX, nodes are given explicit labels individuating them such as $\{A, B, C, \ldots\}$ or {'rain','sprinkler','ground'}. Often, for the purposes of mathematical notation, it is better to index nodes with integers over a common variable label, e.g., using $\{X_1, X_2, \ldots\}$.[2]

Edges: Defined in this way, edges are all *directed* in the sense that an edge from X_1 to X_2 is not the same as the edge from X_2 to X_1, or $(X_1, X_2) \neq (X_2, X_1)$. An edge (X_1, X_2) will sometimes be written as $X_1 \rightarrow X_2$, and the relation may be described using language like "X_1 is the parent of X_2" or "X_2 is the child of X_1".

Directed paths: Paths are a useful way to understand sequences of edges and the structure of a graph. Informally, to say there is a path between X_i and X_j is to say that one can start at X_i and by traveling from parent to child along the edges leading out from the node that you are currently at, you can eventually reach X_j.

To define it recursively and more precisely, if the edge (X_i, X_j) is in the edge set or if the edges (X_i, X_k) and (X_k, X_j) are in the edge set there is a path from X_i to X_j. Otherwise, a graph has a path from node X_i to X_j if there is a subset of its set of edges such that the set contains edges (X_i, X_k) and (X_l, X_j) and there is a path from X_k to X_l.

Adjacency Matrix Perspective

For a fixed set of nodes X of size N, each graph is uniquely defined by its edge set, which can be seen as a binary $N \times N$ matrix, where each index (i, j) in the matrix is 1 if the graph contains an edge from $X_i \rightarrow X_j$, and 0 if it does not contain such an edge. We will refer to this matrix as $A(G)$.

This means that any values of 1 found on the diagonal of the adjacency matrix (i.e., where $X_i \rightarrow X_j, i = j$) indicate a self-loop on the respective node.

Undirected Graphs

We can still have a coherent view of *undirected* graphs, despite the fact that our primitive notion of an edge is that of a *directed* edge. If a graph is undirected, then if it has an edge from $X_i \rightarrow X_j$ then it has an edge from $X_j \rightarrow X_i$. Equivalently, this means that the adjacency matrix of the graph is symmetric, or $A(G) = A(G)^\top$. However from the viewpoint of the undirected graph, that means that it has only a single edge.

Directed Graphs

From the adjacency matrix perspective we've been considering, all graphs are technically directed, and undirected graphs are a special case where one (undirected) edge would be represented as two symmetric edges.

The number of directed graphs that can be obtained from a set of nodes of size n can be defined explicitly using the fact that they can be encoded as a unique $n \times n$ matrix:

$$R_n = 2^{n^2}$$

Directed Acyclic Graphs: A cycle in a directed graph can be understood as the existence of a path from a node to itself. This can be as simple as a self-loop (i.e., if there is an edge (X_i, X_i) for any node X_i).

Directed acyclic graphs(DAGs) are directed graphs that contain no cycles.

The number of DAGs that obtainable from a set of n noddes can be defined recursively as follows [MOR+04]:

$$R_n = \sum_{k=1}^{n} (-1)^{k+1} \binom{n}{k} 2^{k(n-k)} R_{n-k}$$

Note, because DAGs do not allow any cycles, this means that there can be no self-loops. As a result, every value on the diagonal of a DAG's adjacency matrix will be 0.

Probability Distributions: Conditional, Joint and Marginal

A random variable defined by a conditional probability distribution[3] has a distribution indexed by the realization of some other variable (which itself is often a random variable, especially in the context of Bayesian networks).

The probability mass function (pmf) for discrete random variable X with value x will be noted as $P(X = x)$. Often, when discussing the full set of potential values (and not just a single value), we leave out the $= x$ and just indicate $P(X)$.[4]

The conditional probability of X with value x given another variable Y with value y is $P(X = x | Y = y)$. Much like above, if we want to consider the probability of each possible event without specifying one, sometimes this will be written as $P(X | Y = y)$. If we are considering conditioning on any of the possible values of the known variable, we might use the notation $P(X|Y)$, but that is a slight abuse of the notation.

You *can* view $P(X|Y)$ as a function over the $X \times Y$ space. But do not interpret that as a probability function. Rather, this defines a probability function for X relative to each value of Y. Without conditioning on Y we have many potential probability functions for X. Equivalently, it denotes a *family* of probability functions on X indexed by the values $Y = y$.

The *joint probability* of X and Y is the probability that both X and Y occur in the event set in question. This is noted as $P(X, Y)$ or

2. Despite pythonic counting beginning with 0, I chose not to begin this series with 0 because when dealing with variables that might be used in statistical regressions, the 0 subscript will have a specific meaning that separates it from the rest of the notation. For example when expressing multivariate regression as $Y = \beta X + \varepsilon, \varepsilon \sim \mathcal{N}(0, \Sigma)$, β_0 refers to the parameter associated with a constant variable $x_0 = 1$ and X is normally defined as $x_1, x_2, x_3, \ldots$. This allows a simple additive constant to be estimated, which often is not of interest to statistical tests, acting as a scaling constant. This makes for a simpler notation than $Y = \beta_0 + \beta X + \varepsilon$, because that is equivalent to $Y = \beta X + \varepsilon$ if $x_0 = 1$. But, in other cases (e.g., [PG12]) 0 index will be used to indicate background sources for events in a system.

3. Rather than choose a particular interpretation of probability over event sets (e.g., Bayesian or frequentist), I will attempt to remain neutral, as those concerns are not central to the issues of graphs and simple sampling.

4. If one is dealing with continuous quantities rather than discrete quantities one will have to use a probability density function (pdf) which does not have as straightforward an interpretation as a probability mass function. This difficult stems from the fact that (under most cases) the probability of any particular event occurring is "measure zero", or "almost surely" impossible. Without getting into measure theory and the foundation of calculus and continuity we can simply note that it is not that any individual event has non-zero probability, but that sets of events have non-zero probability.As a result, continuous random variables are more easily understood in terms a cumulative density function (cdf), which states not how likely any individual event is, but how likely it is that the event in question is less than a value x. The notation usually given for a cdf of this sort is $F(X \leq x) = \int_{-\infty}^{x} f(u)du$, where $f(u)$ is the associated probability density function.

$P(X \cap Y)$ (using the set theoretic intersection operation). Similar to $P(X|Y)$, you *can* view $P(X,Y)$ as a function over the space defined by $X \times Y$. However, $P(X,Y)$ is a probability function in the sense that the sum of $P(X=x, Y=y)$ over all the possible events in the space defined by $(x,y) \in X \times Y$ equals 1.

The *marginal probability* of X is just $P(X)$. The term "marginalization" refers to the notion of summing over values of Y in their joint probability. When probabilities were recorded in probability tables, the sum would be recorded in the *margins*. Formally, this can be stated as $P(X) = \sum_{y \in Y} P(X,Y)$.

Relating conditional and joint probabilities

Conditional probabilities are related to joint probabilities using the following form:

$$P(X|Y=y) = \frac{P(X,Y=y)}{P(Y=y)} = \frac{P(X,Y=y)}{\sum_{x \in X} P(X=x, Y=y)}$$

Equivalently:

$$P(X,Y=y) = P(X|Y=y)P(X)$$

Bayes' Theorem

Bayes' Theorem can be seen as a result of how to relate conditional and joint probabilities. Or more importantly, how to compute the probability of a variable once you know something about some other variable.

Namely, if we want to know $P(X|Y)$ we can transform it into $\frac{P(X,Y)}{\sum_{x \in X} P(X=x,Y)}$, but then can also transform joint probabilities ($P(X,Y)$) into statements about conditional and marginal probabilities ($P(X|Y)P(X)$). This leaves us with

$$P(X|Y) = \frac{P(Y|X)P(X)}{\sum_{x \in X} P(Y|X=x)P(X=x)}$$

Probabilistic Independence

To say that two variables are independent of each other means that knowing/conditioning on the realization of one variable is irrelevant to the distribution of the other variable. This is equivalent to saying that the joint probability is equal to the multiplication of the probabilities of the two events.

If two variables are conditionally independent, that means that conditional on some set of variables, condition

Example: Marginal Independence $\neq$ Conditional Independence

Consider the following example:

$$\begin{aligned} X &\sim \text{Bernoulli}_{\{0,1\}}(.5),\ Y \sim \text{Bernoulli}_{\{0,1\}}(.5) \\ Z &= X \oplus Y, \oplus \equiv \text{XOR} \end{aligned}$$

Note that, $X \perp\!\!\!\perp Y$ but $X \not\perp\!\!\!\perp Y|Z$.

Bayesian Networks

Bayesian networks are a class of graphical models that have particular probabilistic semantics attached to their nodes and edges. This makes them probabilistic graphical models.

In Bayesian networks when a variable is conditioned on the total set of its parents and children, it is conditionally independent of any other variables in the graph. This is known as the "Markov blanket" of that node.[5]

Common assumptions in Bayesian networks

While there are extensions to these models, a number of assumptions commonly hold.

Fixed node set: The network is considered to be comprehensive in the sense that there is a fixed set of n known nodes. This rules out the possibility of hidden/latent variables as being part of the network. From this perspective inducing hidden nodes requires postulating a new graph that is potentially unrelated to the previous graph.

Trial-based events, complete activation and DAG-hood: Within a trial, all events are presumed to occur simultaneously.There is no notion of temporal asynchrony, where one node/variable takes on a value before its children take on a value (even if in reality – i.e., outside the model – that variable is known to occur before its child). Additionally, the probabilistic semantics will be defined over the entirety of the graph which means that one cannot sample a proper subset of the nodes of a graph without marginalizing out and incorporating information from the ignored nodes into the subset in question.

This property also explains why Bayesian networks need to be acyclic. Most of the time when we consider causal cycles in the world the cycle relies on a temporal delay between the causes and their effects to take place. If the cause and its effect is simultaneous, it becomes difficult (if not nonsensical) to determine which is the cause and which is the effect — they seem instead to be mutually definitional. But, as noted above, when sampling in Bayesian networks simultaneity is presumed for *all* of the nodes.

Independence in Bayes Nets

One of the standard ways of describing the relation between the semantics (probability values) and syntax (graphical structure) of Bayesian networks is how graph encodes particular conditional independence assumptions between the nodes of the graph. Indeed, in some cases Bayesian networks merely play the role of a convenient representation for conditional and marginal independence relationships between different variables.

It is the perspective of the graphs as *merely* representing the independence relationships and the focus on inference that leads to the focus on equivalence classes of Bayes nets. The set of graphs $\{A \rightarrow B \rightarrow C,\ A \leftarrow B \rightarrow C,\ \text{and}\ A \leftarrow B \leftarrow C\}$ represent the same conditional independence relationships, and thus cannot be distinguished on the basis of observational evidence alone. This also leads to the emphasis on finding V-structures or common-cause structures where (at least) two arrows are directed into the same child with no direct link between those parents(e.g., $A \rightarrow B \leftarrow C$). V-structures are observationally distinguishable because any reversing the direction of any of the arrows will alter the conditional independence relations that are guaranteed by the graphical structure.[6]

Though accurate, this eschews important aspects of the semantics distinguishing arrows with different directions when you consider the kinds of values variables take on.

Directional semantics between different types of nodes: The conditional distributions of child nodes are usually defined with parameter functions that take as arguments their parents' realizations for that trial. Bayes nets often are used to exclusively

5. The word "Markov" refers to Andrei Markov and appears as a prefix to many other terms. It most often indicates that some kind of independence property holds. For example, a Markov chain is a sequence (chain) of variables in which each variable depends only on the value of the immediately preceding and postceding variables in the chain. Properties like this make computation easier.

6. A more thorough analysis of this relation between graph structures and implied conditional independence relations invokes the discussion of *d-separation*. However, d-separation (despite claims that "[t]he intuition behind [it] is simple") is a more subtle concept than it at first appears as it involves both which nodes are observed and the underlying structure.

represent discrete (usually, binary) nodes the distribution is usually defined as an arbitrary probability distribution associated with the label of it's parent's realization.

If we allow (for example) positive continuous valued nodes to exist in relation to discrete nodes the kind of distributions available to describe relations between these nodes changes depending upon the direction of the arrow. A continuous node taking on positive real values mapping to an arbitrarily labeled binary node taking on values $\{a,b\}$ will require a function that maps from $\mathbb{R} \to [0,1]$, where it maps to the probability that the child node takes on (for instance) the value a[7].However, if the relationship goes the other direction, one would need to have a function that maps from $\{a,b\} \to \mathbb{R}$. For example, this might be a Gaussian distributions for a and b $((\mu_a,\sigma_a),(\mu_b,\sigma_b))$. Regardless of the particular distributions, the key is that the functional form of the distributions are radically different.

Sampling and semantics in Bayes Nets

The procedure we will use to sample from Bayesian networks uses an *active sample set*. This is the set of nodes for which we have well-defined distributions at the time of sampling.

There will always be at least one node in a Bayesian network that has no parents. We will call these nodes *orphans*. To sample a trial from the Bayesian network we begin with the orphans. Because orphans have no parents – in order for the Bayes net to be well-defined – each orphan will have a well-defined probability distribution available for direct sampling. The set of orphans is our first active sample set.

After sampling from all of the orphans, we will take the union of the sets of children of the orphans, and at least one of these nodes will have values sampled for all of its parents. We take the set of orphans whose entire parent-set has sampled values, and sample from the conditional distributions defined relative to their parents' sampled values and make this the *active sample set*.

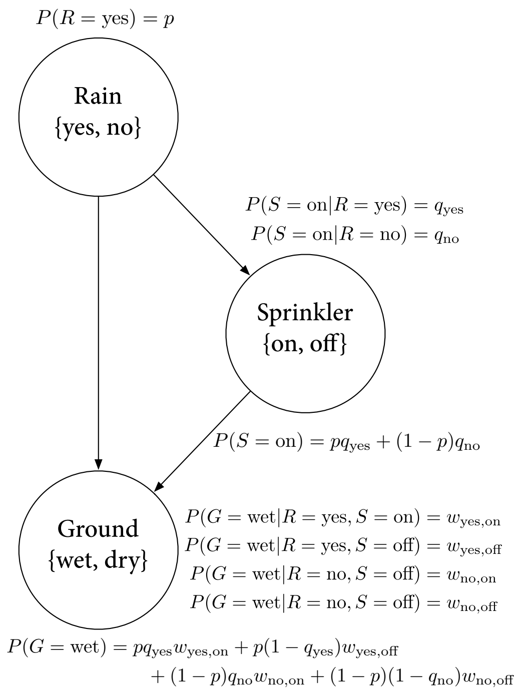
After sampling the active sample set, we will either have new variables whose distributions are well-defined or will have sampled all of the variables in the graph for that trial.

Example: Rain, Sprinkler & Ground

In the sprinkler Bayesian network in Figure 1[8], there three discrete nodes that represent whether it *Rains* (yes or no), whether the *Sprinkler* is on (on or off) and whether the *Ground* is wet (wet or dry). The edges encode the fact that the rain listens to no one, that the rain can alter the probability of whether the sprinkler is on, and the rain and the sprinkler together determine how likely it is that the ground is wet.

Causal Bayesian Networks

Causal Bayesian networks are Bayesian networks that are given an interventional operation allowing for "graph surgery" by cutting nodes off from their parents[9]. Interventions are cases where a causal force is able to exogenously set the values of individual nodes, rendering intervened on nodes independent of their parents.



$P(R = \text{yes}) = p$
Rain {yes, no}
$P(S = \text{on}|R = \text{yes}) = q_{\text{yes}}$
$P(S = \text{on}|R = \text{no}) = q_{\text{no}}$
Sprinkler {on, off}
$P(S = \text{on}) = pq_{\text{yes}} + (1-p)q_{\text{no}}$
Ground {wet, dry}
$P(G = \text{wet}|R = \text{yes}, S = \text{on}) = w_{\text{yes,on}}$
$P(G = \text{wet}|R = \text{yes}, S = \text{off}) = w_{\text{yes,off}}$
$P(G = \text{wet}|R = \text{no}, S = \text{off}) = w_{\text{no,on}}$
$P(G = \text{wet}|R = \text{no}, S = \text{off}) = w_{\text{no,off}}$
$P(G = \text{wet}) = pq_{\text{yes}}w_{\text{yes,on}} + p(1-q_{\text{yes}})w_{\text{yes,off}} + (1-p)q_{\text{no}}w_{\text{no,on}} + (1-p)(1-q_{\text{no}})w_{\text{no,off}}$

Fig. 1: *An Bayesian network describing the sprinkler example. Including both conditional and marginal distributions.*

NetworkX [HSS08]

NetworkX is a package for using and analyzing graphs and complex networks. It stores different kinds of graphs as variations on a "dict of dicts of dicts" structure. For example, directed graphs are stored as two dict-of-dicts-of-dicts structures[10].

Basic NetworkX operations

NetworkX is usually imported using the `nx` abbreviation, and you input nodes and edges as lists of tuples, which can be assigned dictionaries as their last argument, which stores the dictionary as the nodes' or edges' data.

```python
import networkx as nx

G = nx.DiGraph() # init directed graph
G.add_edges_from(edge_list) # input edges
G.add_nodes_from(node_list) # input nodes
edge_list = G.edges(data=True) # output edges
node_list = G.nodes(data=True) # output nodes
```

7. If the function maps directly to one of the labeled binary values this can be represented as having probability 1 of mapping to either *a* or *b*.

8. This is an ill-specified Bayesian network, because while I have specified the states and their relations, I left open the potential interpretation of the parameters and how they relate to one another. I did so because it shows both the limits and strengths of what is encoded knowing only the structure, computing both conditional and marginal distributions for all variables.

9. This is technically a more general definition than that given in [Pea00] as in that case there is a specific semantic flavor given to interventions as they affect the probabilistic semantics of the variables within the network. This is related to his notion of a `do`-operator which deterministically sets a node to a particular value. Because here we are considering a version of intervention that affects the *structure* of a set of graphs rather than an intervention's results on a specific parameterized graph, this greater specificity is unnecessary.

10. It can also represent multi-graphs (graphs where multiple versions of "the same" edge from the adjacency matrix perspective can exist and will (usually) carry different semantics). We will not be using the multigraph feature of NetworkX, as multigraphs are not traditionally used in the context of Bayesian networks.

CBNX: Graphs

Here we will look at some of the basic operations described in the *ipython notebook* [JOP+] found at CBNX. For space and formatting reasons this code may differ slightly from that either in the variable names or comments, for the original version of these code snippets see graph-builder-code.

Other packages

In addition to networkX, we need to import numpy [VDWCV11], scipy [JOP+], and functions from itertools.

```
import numpy as np
import scipy
from itertools import chain, combinations, tee
```

Beginning with a max-graph

Starting with the max graph for a set of nodes (i.e., the graph with N^2 edges), we build an iterator that returns graphs by successively removing subsets of edges. Because we start with the max graph, this procedure will visit all possible subgraphs. One challenge that arises when visiting *all* possible subgraphs is the sheer magnitude of that search space (2^{N^2}).

```
def completeDiGraph(nodes):
    G = nx.DiGraph()
    G.add_nodes_from(nodes)
    edgelist = list(combinations(nodes,2))
    edgelist.extend([(y,x) for x,y in edgelist)
    edgelist.extend([(x,x) for x in nodes])
    G.add_edges_from(edgelist)
    return G
```

Preëmptive Filters

The graph explosion problem is helped by determining which individual edges are known to always be present and which ones are known to never be present. In this way we can reduce the size of the edgeset over which we will be iterating.

Filters can be applied by using `filter_Graph()`, which takes a graph and a filter_set as its arguments and returns a graph. A filter_set is a set of functions that take each take (at least) a graph as an argument and return a graph with a reduced edgeset according to the semantics of the filter.

```
def filter_Graph(G,filter_set):
    graph = G.copy()
    for f in filter_set:
        graph = f(graph)
    return graph
```

Example filter: remove self-loops

By default the graph completed by `completeDiGraph()` will have self-loops, often we will not want this (e.g., DAGs cannot contain self-loops).

```
def extract_remove_self_loops_filter():
    def remove_self_loops_filter(G):
        g2 = G.copy()
        g2.remove_edges_from(g2.selfloop_edges())
        return g2
    return remove_self_loops_filter
```

Conditions

The enumeration portion of this approach is defined in this `conditionalSubgraphs` function.[#]_ This allows you to pass in a graph from which you will want to sample subgraphs that meet the conditions that you also pass in.

```
def conditionalSubgraphs(G,condition_list):
    for edges in powerset(G.edges()):
        G_test = G.copy()
        G_test.remove_edges_from(edges)
        if all([c(G_test) for c in condition_list]):
            yield G_test
```

Example condition: requiring complete paths

This condition holds only if a graph has paths from the first node to the second node for each 2-tuple in the node-pair list.

```
def create_path_complete_condition(n_p):
    def path_complete_condition(G):
        return all([nx.has_path(G,x,y) for x,y in n_p])
    return path_complete_condition
```

Non-destructive conditional subgraph generators

Because `conditionalSubgraph` produces an iterator, applying a condition after that initial set is generated, requires splitting it into two copies of the iterator. This involves the `tee` function from the `itertools` core package.

```
def new_conditional_graph_set(graph_set,cond_list):
    graph_set_newer, graph_set_test = tee(graph_set,2)
    def gen():
        for G in graph_set_test:
            G_test = G.copy()
            if all([c(G_test) for c in condition_list]):
                yield G_test
    return graph_set_newer, gen()
```

Filters versus Conditions: which to use: The structural differences between filters and conditions highlight how they are to be used. Filters are intended to apply a graph to reduce its edge set in place; as such they return a graph. Conditions return truth values — they are applied to graph set reducing the size of that graph set.

CBNX: Representing probabilistic relations and sampling

We discuss an algorithm for sampling from Bayesian networks above (sampling). But, most of the difficult parts of encoding a sampling procedure prove (in this case) to do with the algorithm. Rather, the most pressing difficulties arise from attempting to store the relevant information within the NetworkX data dictionaries, so that a self-contained graphical object can be imported and exported. There is a general problem of a lack of standard storage format for Bayesian networks (and probabilistic graphical models in general). This is just one flavor of that problem.

A CBNX implementation for sprinkler graph

Below I will illustrate how to use NetworkX [HSS08] and node-associated attributes to define and sample from a parameterized version of the sprinkler Bayesian network represented in abstract, graphical form in Figure 1. for space reasons comments and formatting were reduced, if you wish to see the original code it can be found at sampling-code.

11. Note that powerset will need to be built (see CBNX for details).

Sampling infrastructure

```
def sample_from_graph(G,f_dict=None,k = 1):
    if f_dict == None:
        f_dict = {"choice": np.random.choice}
    n_dict = G.nodes(data = True)
    n_ids = np.array(G.nodes())
    n_states = [(n[0],n[1]["state_space"])
        for n in n_dict]
    orphans = [n for n in n_dict
        if n[1]["parents"]==[]]
    s_values = np.empty([len(n_states),k],dtype='U20')
    s_nodes = []
    for n in orphans:
        samp_f = str_to_f(n[1]["sample_function"],
            f_dict)
        s_states = n[1]["state_space"]
        s_dist = n[1]["dist"]
        s_idx = G.nodes().index(n[0])
        s_values[s_idx,:]  = samp_f(s_states,
            size=[1,k],p=s_dist)
        s_nodes.append(n[0])
    while set(s_nodes) < set(G.nodes()):
        nodes_to_sample = has_full_parents(G,s_nodes)
        for n in nodes_to_sample:
            par_indices = [(par,G.nodes().index(par))
                for par in G.node[n]["parents"]]
            par_vals = [(par[0],s_values[par[1],:])
                for par in par_indices]
            samp_index = G.nodes().index(n)
            s_values[samp_index,:] = cond_samp(G,n,
                par_vals,f_dict,k)
            s_nodes.append(n)
    return s_values

def has_full_parents(G,s_n):
    check_n = [x for x in G.nodes() if x not in s_n]
    nodes_to_be_sampled = []
    for n in G.nodes(data = True):
        if (n[0] in check_n) & (n[1]["parents"]<=s_n):
            nodes_to_be_sampled.append(n[0])
    if len(nodes_to_be_sampled)==0:
        raise RuntimeError("A node must be sampled")
    return nodes_to_be_sampled

def nodeset_query(G,n_set,n_atr=[]):
    if len(n_atr)==0:
        return [n for n in G.nodes(data = True)
            if n[0] in n_set]
    else:
        return_val = []
        for n in G.nodes(data=True):
            if n[0] in node_set:
                return_val.append((n[0],
                {attr:n[1][attr] for attr in n_atr}))
        return return_val

def cond_samp(G,n,par_vals,f_dict, k = 1):
    try: n in G
    except KeyError:
        print("{} is not in graph".format(n))
    output = np.empty(k,dtype="U20")
    for i in np.arange(k):
        val_list = []
        for p in par_vals:
            val_list.append(tuple([p[0],p[1][i]]))
        samp_dist = G.node[n]["dist"][tuple(val_list)]
        samp_f = str_to_f(
            G.node[n]["sample_function"],f_dict)
        samp_states = G.node[n]["state_space"]
        temp_output = samp_f(samp_states,
            size=1,p=samp_dist)
        output[i] = temp_output[0]
    return output

def str_to_f(f_name, f_dict=None):
    if f_dict == None:
        f_dict = {"choice": np.random.choice}
    try: f_dict[f_name]
    except KeyError:
        print("{} is not defined.".format(f_name))
    return f_dict[f_name]
```

Sampling from the sprinkler Bayes net with CBNX

The following encodes the sprinkler network from Figure 1 with parameters $p = .2, q_{\text{yes}} = .01, q_{\text{no}} = .4, w_{\text{yes,on}} = .99, w_{\text{yes,off}} = .8, w_{\text{no,on}} = .9 \text{and} w_{\text{no,off}=0}$. This distribution is meant to accord with our intuitions that rain and sprinklers increase the probability of the ground being wet, and that we are less likely to use the sprinkler when it has rained.

```
node_prop_list = [("rain",{
    "state_space":("yes","no"),
    "sample_function": "choice",
    "parents":[],
    "dist":[.2,.8]}),
    ("sprinkler",{
    "state_space":("on","off"),
    "sample_function": "choice",
    "parents":["rain"],
    "dist":{(("rain","yes"),):[.01,.99],
            (("rain","no"),):[.4,.6]}}),
    ("grass_wet",{
    "state_space":("wet","dry"),
    "sample_function": "choice",
    "parents":["rain","sprinkler"],
    "dist":{
        (("rain","yes"),("sprinkler","on")):[.99,.01],
        (("rain","yes"),("sprinkler","off")):[.8,.2],
        (("rain","no"),("sprinkler","on")):[.9,.1],
        (("rain","no"),("sprinkler","off")):[0,1]}})]


edge_list = [("sprinkler","grass_wet"),
    ("rain","sprinkler"),
    ("rain","grass_wet")]

G = nx.DiGraph()
G.clear()
G.add_edges_from(edge_list)
G.add_nodes_from(node_prop_list)
test = sample_from_graph(G,k=10)
```

Causal Theories and Computational Cognitive Science

Theory based causal induction is a formal framework arising out of the tradition in computational cognitive science to approach problems of human cognition with rational, computational-level analyses [GT09]. Causal theories form generative models for defining classes of parameterized probabilistic graphical models. They rely on defining a set of classes of entities (ontology), potential relationships between those classes of entities and particular entities (plausible relations), and particular parameterizations of how those relations manifest in observable data (or in how other relations eventually ground out into observable data). This allows Griffiths and Tenenbaum to subsume the prediction of a wide array of human causal inductive, learning and reasoning behavior using this framework for generating graphical models and doing inference over the structures they generate.

Rational analysis

Rational analysis is a technique that frees us from some of the problems inherent in mechanistic modeling in cognition. We specify the goals of the cognitive system, the environment in which it exists and minimal constraints on the computations available to the agent. We translate this into mathematically precise accounts of "mechanism-free casting[s] of psychological [theories]" for optimal behavior. These formal models provide empirical predictions that can be evaluated by studying human cognitive behavior under different observable environmental conditions [And90][11]. If the model disagrees with the empirical data, we iterate — reëvaluating each component of the theory until we match a wide variety[12] of empirical data.

Computational-Level Analysis of Human Cognition

A computational-level analysis [Mar82] is one in which we model a system in terms of its functional role(s) and how they would be optimally solved. This is distinguished from algorithmic-level analysis by not caring how this goal achievement state is implemented in terms of the formal structure of the underlying system and from mechanistic-level analysis by not caring about the physical structure of how these systems are implemented (which may vary widely while still meeting the structure of the algorithmic-level which itself accomplishes the goals of the computational level).

A classic example [Mar82] of the three-levels of analysis are different ways of studying flying with the example of bird-flight. The mechanistic-level analysis would be to study feathers, cells and so on to understand the component subparts of individual birds. The algorithmic-level analysis would look at how these subparts fit together to form an active whole that is capable of flying often by flapping its wings in a particular way. The computational-level analysis would be a theory of aerodynamics with specific accounts for the way forces interact to produce flight through the particular motions of flying observed in the birds.

Causal theories: ontology, plausible relations, functional form

The causal theory framework generalizes specifying Bayesian network in the same way first-order logic generalizes specifying propositions in propositional logic. A causal theory requires elements necessary to populate nodes, those nodes with properties, and relations between the nodes, stating which of those relations are plausible (and how plausible), and a specific, precise formulation for how those relations manifest in terms of a probabilistic semantics. In the terms of [GT09]'s theory-based causal induction, this requires specifying an ontology, plausible relations over those ontologies, and functional forms for parameterizing those relations.

Ontology: This specifies the full space of potential kinds of entities, properties and relations that exist. This is the basis around which everything else will be defined. It is straightforward populate nodes with features using the data dictionary in NetworkX.

Plausible Relations: This specifies which of the total set of relations allowed by the ontology are plausible and how plausible. If you do not dramatically restrict the sets of relations you consider, there will be an explosion of possibilities. People, even young children, have many expectations about what sorts of things can can feasibly be causally related to one another. This sometimes has been interpreted as the plausible existence of a mechanism linking cause and effect. For example, we know that in most situations a fan is more likely than a tuning fork to blow out a candle.

Functional form:

> Even in the most basic cases of causal induction we draw on expectations as to whether the effects of one variable on another are positive or negative, whether multiple causes interact or are independent, and what type of events (binary, continuous, or rates) are relevant to evaluating causal relationships.
>
> —[GT09]

Of course, this allows for uncertainty about these functional forms and indeed, quite different judgments can be warranted depending on treats the underlying relation and structure of the data (e.g., continuous vs. binary data [PG11]).

Generalizations to other kinds of logical/graphical conditions

The causal theory framework is richer than the set of examples developed in [GT09]. It can express conditions of graphical connectivity, context-sensitive functional forms, substructures of constrained plausible relations, among others.

In [GT09], plausible relations are described in terms of sufficient conditions, implicitly suggesting that most relations are not plausible. However, we can also make necessary statements about the kinds of relations that *must* be there. And one can see this as selecting a subset of all the possible graphs implementable by the set of nodes defined by the ontology. It is for this purpose that I first arrived at the node enumeration.

One goal for CBNX is to enable causal theory programming. The utilities in `networkX`, plus the enumerating, filtering and conditioning functions in CBNX, ease implementing higher-order graphical conditions (e.g., a directed path necessarily existing between two nodes) than in the original notation described in [GT09]. These ideas were expressible in the original mathematical framework, but would have required a good deal more notational infrastructure to represent. CBNX not only provides a notation, but a programming infrastructure for expressing and using these kinds of conditions.

Uses in modeling human cognition

Using this framework, Griffiths and Tenenbaum were able to provide comprehensive coverage for a number of human psychology experiments. This allows them to model people's inferences in causal induction and learning regarding different functional forms, at different points in development, with different amounts of data, with and without interventions, and in continuous time and space (to name only a few of the different conditions covered).

They successfully modeled human behavior using this framework by treating people as optimal solvers of this computational problem[13] (at least as defined by their framework). Furthermore, by examining different but related experiments, they were able to demonstrate the different ways in which specific kinds of prior knowledge are called upon differentially to inform human causal induction resulting in quite different inferences on a rational statistical basis.

Cognition as Benchmark, Compass, and Map

People have always been able to make judgments that are beyond machine learning's state-of-the-art. In domains like object recognition, we are generally confident in people's judgments as

12. As Anderson notes, it is often the mathematization that proves to be the most difficult aspect of this procedure [And90].

13. Optimality in these cases is taken to mean on average approximating the posterior distribution of some inference problem defined by the authors in each case.

veridical, and – as such – they have been used as a benchmark against which to test and train machine learning systems. The eventual goal is that the system reaches a Turing point — the point at which machine performance and human performance are indistinguishable.

But that is not the only way human behavior can guide machine learning. In domains like causal induction, people's judgments cannot form a benchmark in the traditional sense because we cannot trust people to be "correct". Nonetheless, people *do* make these judgments and, more importantly, these judgments exhibit systematic patterns. This systematicity allows the judgments output by cognition to be modeled using formal, computational frameworks. Further, if we formally characterize both the inputs to *and* outputs from cognition, we can define judgments as optimal according to some model. Formal models of individual cognitive processes can then act as a compass for machine learning, providing a direction for how problems and some solutions can be computed.

Formal frameworks for generating models (e.g., causal theories) can be even more powerful. Data can often be interpreted in multiple ways, with each way requiring a model to generate solutions. Holding the data constant, different goals merit different kinds of solutions. Frameworks that generate models, optimality criteria and solutions not only provide a direction for machine learning, but lay out *sets* of possible directions. Generalized methods that use one system for solving many kinds of problems provide the ability to relate these different directions to each other. Formalizing the inputs, processes and outputs of human cognition produces a map of where machine learning could go, even if it never goes to any particular destination. From this, navigators with more details about the particular terrain can find newer and better routes.

Acknowledgements

Thank you to Jess Hamrick for aiding in the design of the underlying code, Katy Huff and Stéfan van der Walt for aiding in getting the bibliography working and helping me navigate github and the submission and review processes, Seb Benthall and Ankur Ankan for helping reviews, and Elizabeth Seiver for comments and support throughout the writing process.

REFERENCES

[And90] J. R. Anderson. *The adaptive character of thought*. Erlbaum, Hillsdale, NJ, 1990.

[GT09] T. L. Griffiths and J. B. Tenenbaum. Theory-based causal induction. *Psychological review*, 116(4), 2009.

[HSS08] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA, August 2008.

[JOP$^+$] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python. http://www.scipy.org/, 2001–. Online (accessed 2015-07-04).

[Mar82] D. Marr. *Vision*. W. H. Freeman, San Francisco, CA, 1982.

[MOR$^+$04] Brendan D McKay, Frederique E Oggier, Gordon F Royle, NJA Sloane, Ian M Wanless, and Herbert S Wilf. Acyclic digraphs and eigenvalues of (0, 1)-matrices. *Journal of Integer Sequences*, 7:3, 2004.

[Pac15] M.D. Pacer. Causal-Bayesian-NetworkX. http://dx.doi.org/10.6084/m9.figshare.1471763, 2015. Online (accessed July 2, 2015). URL: http://dx.doi.org/10.6084/m9.figshare.1471763, doi:10.6084/m9.figshare.1471763.

[Pea00] J. Pearl. *Causality: Models, reasoning and inference*. Cambridge University Press, Cambridge, UK, 2000.

[PG11] M.D. Pacer and T.L. Griffiths. A rational model of causal induction with continuous causes. In *Advances in Neural Information Processing Systems*, volume 24, Cambridge, MA, 2011. MIT Press.

[PG12] M.D. Pacer and T.L. Griffiths. Elements of a rational framework for continuous-time causal induction. In *Proc. of the 34th Conf. of the CogSci Society*, 2012.

[VDWCV11] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.