Causal Search in Practice

Student: J. Alexander D. Atkins
Supervisor: Vanessa Didelez
Level 3; 20 credit points

6 May 2011

Acknowledgement of Sources

For all ideas taken from other sources (books, articles, internet), the source of the ideas is mentioned in the main text and fully referenced at the end of the report.

All material which is quoted essentially word-for-word from other sources is given in quotation marks and referenced.

Pictures and diagrams copied from the internet or other sources are labelled with a reference to the web page,book, article etc.

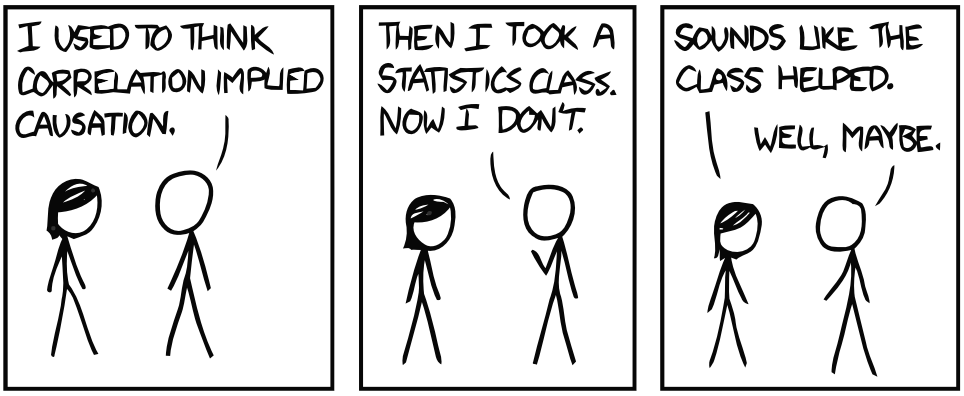
Signed:

Dated:

Contents

1 Introduction

One often hears the saying "association is not causation", and indeed, when only two qualities are observed, this is true. We are brought up to believe that, in order to make any meaningful conclusions about the influence of one quality on another in a population of objects, we must set up an experiment in which we control the quality that we suspect affects the other, in order to isolate it from influence in the other direction.



Unfortunately, sometimes it is not feasible, or not ethical, to perform an experiment. Suppose you want to know how factors such as unemployment, alcohol abuse, drug use, happiness, homelessness and criminalisation relate to each other in the population of a particular country. While one can observe these factors and measure statistical relations (by taking a survey) it is impractical to control them for the purposes of experiment. At the same time it is not obvious, even if statistical links are found, which way round the cause and effect might be. Yet ministers and other people in positions of responsibility must make decisions based on the best guesses available, for example when classifying a new drug.

Since we, as a society, are forced to make important decisions based on inferences about causal relationships, we are as well to study the matter in a formal and theoretical context, to see whether anything about the direction of influence can be inferred from observational data alone, in the absence of background knowledge (which might tell us, for example, that a subject's age cannot be influenced by his lifestyle).

You may be under the impression that such an endeavour is futile. In section 6 of this report I hope to convince you otherwise. Section 3 lays out the language that we will need for this discussion, and sections 4 and 5 make explicit the assumptions that we require about the connection between causal influence and statistical properties. In section 8 I present a method for drawing inferences about causation from observational data (under certain assumptions

about the data) and in section 9 I investigate what happens when some of these assumptions are relaxed, and explore improvements to the basic method in order to overcome the problems that arise. In section 10 I actually try out the method on simulated data, and compare one or two modifications of the algorithm against each other. In section 11 I present my findings, and in section 12 I point to areas in which this study might be, and is being, taken further.

None of this is original, except for the two-pass method in section 9 and the specific details of my implementation, presented in Appendix A and described in sections 9 and 10.

2 What is Causality?

Philosophical discussion as to the existence and nature of causality is ongoing. For a short, readable argument against causality, see Hume (1740). For an unusual and brief argument defending causality by reference to Einstein's special relativity theory, see Salmon (1977). For the purposes of this study, the existence of cause and effect will be taken as read, and as for its nature, we will just have to fall back on familiar intuition. In defence of intuition, then, Spirtes, Glymour, and Scheines (2000) present the following examples:

> "The skeptic about causality pushes the brake pedal to make his car slow, flips a switch to make a lamp glow, puts his money in the bank to collect interest."

I feel that these examples are not sufficient to show that we all have an intuitive belief in causality, since the 'skeptic' here knows that the two events pushing the brake pedal and the car slowing down are associated, he knows that he wants one to happen, and he knows that he only has control over the other, so a sufficient rule of thumb would be: If you want an event **E** to happen, either bring it about directly (if you can control it) or, failing that, enact any other event with a positive association, to increase **E**'s probability. However, I note that the student who frequently orders pizza and wants some tonight, though she may telephone the pizza company and ask for one, her intuition will not allow her merely to put plates in the oven or to tell her housemates that it will arrive shortly, even though she has control over these events and they are positively associated with the event of the pizza arriving. Indeed, they even occur before the arrival of the pizza, so expanding the rule to take into account temporal information still does not reflect the extent of our intuition.

3 Causal Graphs

Causal networks are conventionally depicted by graphs. A node in the graph represents an event and a directed edge represents a *direct* causal relationship

where the tail (parent) is the cause and the head (child) is the effect. 'Direct' here just means that the cause influences the effect directly, and not via an intermediary.

Examples of causal graphs can be seen below. Figure 1(a) shows a possible model for the example given in the previous section. The same notation extends easily to random variables instead of events, so Figure 1(c) gives an example where the nodes represent random variables, two of which are continuous, and one of which is boolean.

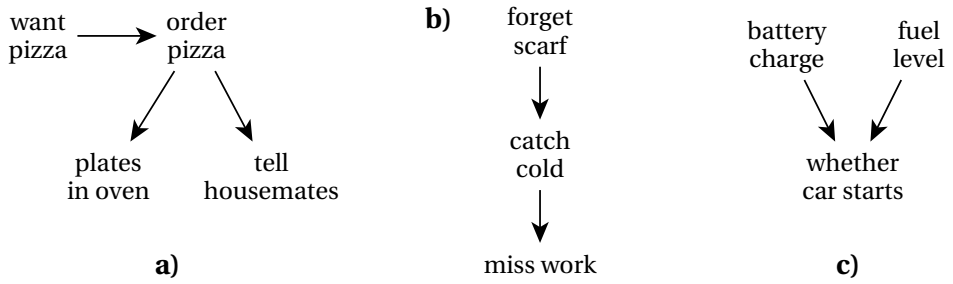


Figure 1: Examples of causal graphs

4 Intervention Formula

But what is it that makes these graphs *causal*? Clearly the nodes must represent events or random variables, but what criterion must the arrangement of edges satisfy in order to be considered correct? And what does 'direct cause' really mean?

A graph is said to be causal if it satisfies the intervention formula. There is more than one way of introducing this, but I think the best for our purposes is given by (Pearl, 2009, section 3.2.1). He writes:

> "In its general form, a functional causal model consists of a set of equations of the form
>
> $$x_i = f_i(pa_i, u_i), \quad i = 1, \ldots, n, \tag{1}$$
>
> where pa_i (connoting *parents*) stands for the set of variables that directly determine the value of X_i and where the U_i represent errors (or 'disturbances') due to ommitted factors."

The causal models that we are looking at in this study will have a further requirement that the random variables U_i be independent of one another. But before we proceed, I had better define statistical independence.

Definition. Two continuous random variables X and Y are independent if and only if their joint probability density is the product of their individual densities, i.e.

$$X \perp\!\!\!\perp Y \iff f_{X,Y}(x,y) = f_X(x) \cdot f_Y(y) \; \forall x, y$$

Already the quotation above places a restriction on the joint probability density function of the variables in the system. Every variable X_i must be determinable by a function that takes only the parents of X_i and some error term as input, and the error is – crucially – independent of all other error terms, and thus of all other variables $X_j, j \neq i$. We will then find that the random variables X_i must satisfy various relationships of conditional independence between each other, but before we can do that, we require another definition.

Definition. For continuous random variables X and Y with probability density functions f_X and f_Y, and joint probability density function $f_{X,Y}$, the probability density of X conditional on Y (aka X given Y) is given by:

$$f_{X|Y}(x) \doteq \frac{f_{X,Y}(x,y)}{f_Y(y)}$$

Combining the above two definitions, two variables X and Y are independent conditional on a third Z, written $X \perp\!\!\!\perp Y \mid Z$, iff:

$$f_{X,Y|Z}(x,y) = f_{X|Z}(x) \cdot f_{Y|Z}(y) \equiv \frac{f_{X,Z}(x,z) \cdot f_{Y,Z}(y,z)}{f_Z(z)^2}$$

The above two definitions and lemma also apply when X, Y and Z are sets of random variables. You just have to read the values x, y and so on as vectors and imagine that the probability density functions are multidimensional, so that, e.g. $f_X : \mathbb{R}^n \mapsto [0,1]$, where n is the number of variables in the set X.

We find, then, that in this 'functional causal model', the variable X_i must be, conditional on the parents pa_i of X_i, independent of all other variables, i.e. $X_i \perp\!\!\!\perp X_j \mid pa_i \; \forall \; X_j \notin pa_i$ with $j \neq i$.

So X_i can be determined (give or take a random error, which is independent of all other observed variables) solely by its parents. But this may not be the only way that X_i can be determined. Regard Figure 1(b). While a friend of mine who notices that I have caught a cold may infer that I am likely to take time off work as a result, might he or she not infer something about my forgetfulness also? Indeed, you can see from the equations above that conditional independence is symmetric (i.e. $X \perp\!\!\!\perp Y \mid Z \iff Y \perp\!\!\!\perp X \mid Z$) so that if the number of hours of work I miss is independent of the number of days I have forgotten my scarf, conditional on the time I am ill, the converse must also be true.

We require a stronger criterion. In order to encapsulate the nature of causal (as opposed to statistical) relationships, you will not be surprised to hear that this will be defined in terms of what would happen to the joint probability

density function of the whole system, were we able (hypothetically) to set up a controlled experiment and force one or more variables to take particular values.

I quote Pearl (2009) again:

> **"Definition.** Given two disjoint sets of variables, X and Y, the causal effect of X on Y, denoted either as $P(y \mid \hat{x})$ or as $P(y \mid \text{do}(x))$, is a function from X to the space of probability distributions on Y. $P(y \mid \hat{x})$ gives the probability of $Y = y$ induced by deleting from the model of (1) all equations corresponding to variables in X and substituting $X = x$ in the remaining equations."

The graphical equivalent of deleting equations as described above would be to remove all edges with X at the head of the arrow.

From all of this section so far, we derive what Pearl (2009) calls a *truncated factorisation formula*. The following describes the behaviour of the system when an intervention is carried out to force a particular variable X_i to take the value x_i':

$$\text{“}P(x_1, \ldots, x_n \mid \hat{x_i}') = \begin{cases} \prod_{j \neq i} P(x_j \mid pa_j) & \text{if } x_i = x_i', \\ 0 & \text{if } x_i \neq x_i'. \text{”} \end{cases} \tag{2}$$

The P on the left there represents the joint probability density function over all variables in the system, including the one we are controlling, and the right-hand-side encapsulates everything we have said about conditional independencies already, the only difference being that now the term $P(x_i \mid pa_i)$ has been removed from the product, so that the influence of pa_i on x_i is no longer relevant.

This provides the real distinction between *observing* that a variable takes a particular value (e.g. that I have caught a disease) and *forcing* a variable to take a particular value (e.g. administering to me a vaccine or antibiotics). If you try to find (using equation (2) above) the marginal probability of my having forgotten to wear a scarf given that you either gave me a disease or somehow prevented me from protracting one, you will be left (after integrating over the other variables, in this case the amount of work I miss) with $P(\text{forget scarf} \mid pa_{\text{forget scarf}})$, which in this case is just the marginal probability $P(\text{forget scarf})$, since the node 'forget scarf' has no parents. Certainly this is not the same as the probability distribution on my work attendance, given that you force me to have, or not to have, a disease, and neither is it the same as the probability distribution on my forgetfulness conditional on your observation that I have, or do not have, a disease.

Almost immediately Pearl generalises (2) to a form suitable for describing the effect of controlling several variables at once:

$$\text{“}P(x_1,\ldots,x_n \mid \hat{s}) = \begin{cases} \prod_{i|X_i \notin S} P(x_i \mid pa_i) & \text{for } x_1,\ldots,x_n \text{ consistent with } s, \\ 0 & \text{otherwise.”} \end{cases} \tag{3}$$

5 Three Axioms

With that theory behind us, we can begin to nail down some concrete assertions about causal graphs. We take these as axioms, so the justification for them will be solely intuitive.

5.1 Causal Markov Condition

This assumption states that all the independencies and conditional independencies implied by the graph are borne out by the joint probability densities. Specifically, it states that each variable is independent of all its non-descendants given its parents, i.e. $X \perp\!\!\!\perp Y \mid pa_X \ \forall\ Y \notin \text{descendants}(X)$.

This relies on the requirement that I mentioned in passing on page 6, when I said that the random errors were to be independent of one another. If this requirement is violated, you introduce ‘latent variables’ – unobserved variables which influence more than one of the observed variables. For now, I will assume that all graphical models are causally sufficient (i.e. that there are no latent variables) but we will return to this issue (briefly) in section 9.

5.2 Causal Minimality Condition

This simply states that you can’t remove any edge without breaking the Markov condition. A graph G with probability distribution P satisfies the minimality condition iff every proper subset of G over the same set of vertices, with the same probability distribution, does not satisfy the Markov condition.

5.3 Faithfulness

The faithfulness condition is subtly different from the minimality condition. It says that every independence or conditional independence relation in the probability distribution is entailed by the graph, or to put it another way, that no independence or conditional independence relation not implied by the Markov condition is true of the distribution.

Note that this is not the same as the minimality condition: it is possible for a causal model to satisfy the Markov and minimality conditions but not satisfy the faithfulness condition. Consider, for example, the graph with vertices $\{A, B, C\}$ and edges $A \to C$, $A \to B$ and $B \to C$, together with a distribution satisfying

the intervention formula (3) from the previous section. It is possible for the influence of A on C via B to cancel out exactly with the direct influence due to the edge $A \to C$. In this case we would find that $A \perp\!\!\!\perp C$, even though this is not implied by the Markov condition. Were we to remove, say, the edge $A \to C$ (but keep the distribution the same) the Markov condition would then imply that $A \perp\!\!\!\perp C \mid B$, but this would not be the case. Likewise if we were to remove $A \to B$ or $B \to C$, the Markov condition would then imply that $A \perp\!\!\!\perp B$ or $B \perp\!\!\!\perp C \mid A$ (respectively) and this would not hold.

Thus this graph and distribution satisfy the Markov and minimality conditions but not the faithfulness condition. On the other hand, the Markov and faithfulness conditions together *do* imply minimality.

6 V-Structures & Markov Equivalence

What we want now is some way of making distinctions between different causal structures from the data alone – in other words when the number of vertices and the conditional independence relations are known, but nothing more. Before I embarked on this study I had not heard of such a thing, but happily much progress has been made in this area in the last thirty years or so, most notably by contributors to the annual *Conference on Uncertainty in Artificial Intelligence*.

Suppose we look at some data relating to Figure 1(b), and another set of data corresponding to Figure 1(c), but suppose also that we have not been told what variables we are looking at, or we have no notion of their meaning – they are just labelled A, B, C, say (and not 'fuel level' etc).

Hitherto I have been blurring the distinction between continuous random variables, and discrete random variables or events. This is because the exact same principles apply to all three – all the equations you have seen so far, and all the definitions of Markov conditions and so on. The only differences are in a) the notation, and b) the underlying tests for conditional independence, which we will look at briefly in the next section (for the continuous case, at least). For simplicity's sake, however, I am trying to stick to the language of continuous random variables.

Well, then. You have two sets of data generated by causal networks such as those in figures 1(b) and (c), and the question is: are they distinguishable? For starters, it is very likely that none of the marginal distributions for any of the variables in one set will match any of those in the other set, as they will have different scales, units, means and so on. But suppose that we have no notion of scale or unit, but that we *are* told that one dataset came from a system of the form $A \to B \to C$ (such as that of Figure 1(b)) and that the other came from a system of the form $A \to B \leftarrow C$ (such as that of Figure 1(c)), and we are challenged with the task of determining which is which.

First perform all possible tests for independence or conditional independence. We will find, if the Markov and faithfulness conditions are satisfied, the

following relations, for both datasets:

- either $A \perp\!\!\!\perp C$ or $A \perp\!\!\!\perp C \mid B$, i.e. $\exists$ a set $S \subseteq V \setminus \{A, C\}$ s.t. $A \perp\!\!\!\perp C \mid S$
- $A \not\perp\!\!\!\perp B$ and $A \not\perp\!\!\!\perp B \mid C$, i.e. $\nexists$ a set $S \subseteq V \setminus \{A, B\}$ s.t. $A \perp\!\!\!\perp B \mid S$
- $B \not\perp\!\!\!\perp C$ and $B \not\perp\!\!\!\perp C \mid A$, i.e. $\nexists$ a set $S \subseteq V \setminus \{B, C\}$ s.t. $B \perp\!\!\!\perp C \mid S$

. . . where V is the set of all vertices, i.e. $\{A, B, C\}$.

At first sight this may not seem helpful, as we have not so far separated the two datasets. We have, however, discovered enough to deduce that the corresponding graphs must both constitute a line of the form $A — B — C$. This observation helps me to point out the difference, which is that in the dataset corresponding to the graph $A \to B \to C$ we will find that $A \perp\!\!\!\perp C \mid B$ but $A \not\perp\!\!\!\perp C$, and in the dataset corresponding to $A \to B \leftarrow C$ we will see that $A \perp\!\!\!\perp C$ but $A \not\perp\!\!\!\perp C \mid B$.

The existence or absence of these conditional independence relations follows from the Markov and faithfulness conditions, but they are also supported by our intuition. Consider Figure 1(b). Clearly my work attendance will be associated with my proficiency at immunising myself from disease, but once you know the full state of my health, neither my attendance nor my attention to attire will tell you anything about the other, which is sufficient intuitive grounds for concluding that the variables representing these will be independent.

Consider now Figure 1(c). This one is less obvious. I got this example from Spirtes et al. (2000) (who derived it from a much more complicated one given by Pearl) so I may as well quote them directly:

> "Whether or not your car starts depends on whether or not the battery is charged and also on whether or not there is fuel in the tank, but these conditions are independent of one another. Suppose you find that your car won't start, and you hold in that case that there is some probability that the fuel tank is empty and some probability that the battery is dead. Suppose next you find that the battery is not dead. Doesn't the probability that the fuel tank is empty change when that information is added?"

I present another example of exactly the same structure. Suppose a given university accepts students for matriculation if they are either academically proficient or good at rowing (or both). Assume that academic standing and rowing prowess are independent in the general population of the country. In that case we would expect a slight negative correlation between these two abilities in the population of students at said university. Neither skill is influenced by the other, nor are they influenced by the student's success at entry, or whether she got into the college of her choice. They are two causes with a common effect, namely, the decision of the university on whether or not to accept the student. We would expect a student's mental and physical talents to be independent of

each other, but (once we have thought about it for a few seconds) dependent conditional on whether or not the student was accepted for a place.

$P(B=1)$	0.5
$P(A=1 \mid B=1)$	0.16
$P(A=1 \mid B=0)$	0.64
$P(C=1 \mid B=1)$	0.9
$P(C=1 \mid B=0)$	0.6
$A=0, B=0, C=0$	0.072
$A=0, B=0, C=1$	0.108
$A=0, B=1, C=0$	0.042
$A=0, B=1, C=1$	0.378
$A=1, B=0, C=0$	0.128
$A=1, B=0, C=1$	0.192
$A=1, B=1, C=0$	0.008
$A=1, B=1, C=1$	0.072

(a) $A \leftarrow B \rightarrow C$

$P(A=1)$	0.4
$P(B=1 \mid A=1)$	0.2
$P(B=1 \mid A=0)$	0.7
$P(C=1 \mid B=1)$	0.9
$P(C=1 \mid B=0)$	0.6
$A=0, B=0, C=0$	0.072
$A=0, B=0, C=1$	0.108
$A=0, B=1, C=0$	0.042
$A=0, B=1, C=1$	0.378
$A=1, B=0, C=0$	0.128
$A=1, B=0, C=1$	0.192
$A=1, B=1, C=0$	0.008
$A=1, B=1, C=1$	0.072

(b) $A \rightarrow B \rightarrow C$

$P(C=1)$	0.75
$P(B=1 \mid C=1)$	0.6
$P(B=1 \mid C=0)$	0.2
$P(A=1 \mid B=1)$	0.16
$P(A=1 \mid B=0)$	0.64
$A=0, B=0, C=0$	0.072
$A=0, B=0, C=1$	0.108
$A=0, B=1, C=0$	0.042
$A=0, B=1, C=1$	0.378
$A=1, B=0, C=0$	0.128
$A=1, B=0, C=1$	0.192
$A=1, B=1, C=0$	0.008
$A=1, B=1, C=1$	0.072

(c) $A \leftarrow B \leftarrow C$

$P(A=1)$	0.4
$P(C=1)$	0.75
$P(B=1 \mid A=0, C=0)$	0.368
$P(B=1 \mid A=0, C=1)$	0.778
$P(B=1 \mid A=1, C=0)$	0.059
$P(B=1 \mid A=1, C=1)$	0.273
$A=0, B=0, C=0$	0.095
$A=0, B=0, C=1$	0.100
$A=0, B=1, C=0$	0.055
$A=0, B=1, C=1$	0.350
$A=1, B=0, C=0$	0.094
$A=1, B=0, C=1$	0.218
$A=1, B=1, C=0$	0.006
$A=1, B=1, C=1$	0.082

(d) $A \rightarrow B \leftarrow C$

Table 1: Markov Equivalence

As an exercise, I decided to attempt to find compatible distributions between all the graphs on three nodes with edges in series as A — B — C, with the four possible combinations of edge direction. I started with the graph $A \leftarrow B \rightarrow C$, and generated the distribution using the description of 'functional causal models' on page 6 as a guide. For simplicity's sake, I let the nodes represent boolean random variables. I chose arbitrary distributions for P_B, $P_{A|B}$ and $P_{C|B}$ as shown in the top part of table 1a. I then computed the joint probabilities

according to the latter definition on page 7, to get the bottom part of table 1a. The way I constructed tables 1b, c and d was similar, except that the values in the top part were read off the joint probability distribution of table 1a. I then again computed the joint probability distribution, assuming that the description of a functional causal model (from page 6) would still hold, even under a different causal structure.

As you can see, the models $A \leftarrow B \rightarrow C$, $A \rightarrow B \rightarrow C$ and $A \leftarrow B \leftarrow C$ were able to give rise to the same probability distribution, but the model $A \rightarrow B \leftarrow C$ was not. This is intended to serve as an informal demonstration that, sometimes, some information about the underlying causal structure can be deduced from the data alone. For example, supposing we were shown the probability distribution outlined in the last eight rows of table 1a. We could then conclude that it is not compatible with the interpretation $A \rightarrow B \leftarrow C$ (the model behind table 1d) as no choice of parameters in that model will give rise to the same distribution.

We would not, however, be able to distinguish between the other three interpretations, as the corresponding graphs all entail the same set of conditional independence relations. What we have just seen is an example of a concept formalised by Verma and Pearl (1990): Markov equivalence.

Two graphs are said to be *Markov equivalent* if they imply the same set of conditional independencies as a result of the causal Markov condition (see subsection 5.1). Verma & Pearl showed that this is an equivalence relation, and they also found that it has something to do with V-structures.

Definition. A V-structure is a subset $\{A, B, C\}$ of a causal graph G, with edges $A \rightarrow B$ and $B \leftarrow C$, but where A and C are not adjacent in G.

The following result from Verma and Pearl (1990) is absolutely crucial for the rest of this study:

Theorem. *Two directed acyclic graphs are Markov equivalent if and only if they share the same skeleton (undirected graph) and have the same V-structures.*

The significance of this is not so much that it gives us a nice and easy way, by graphical inspection, to tell if two causal graphs are Markov equivalent, but that it gives us an idea of what to work towards when designing an algorithm to recover causal information from observational data. This is the holy grail of theorems (in this context). If we can come up with an algorithm that, by performing the right conditional independence tests, can identify the skeleton of the graph (where all the edges are) and all the V-structures, we will have narrowed the possible causal explanations down to one Markov equivalence class, and we'll know that that's as much as we can do.

Now hold that thought. I just want to say a few things about conditional independence tests first.

7 Conditional Independence

For the most part we will treat the statistical tests as a 'black box': in order to do this we must understand the properties of the tests, but need not know the internal details. The test to be used depends on the hypothesised distribution of the variables, but does not affect the theory which is the subject of this report. The algorithm we are about to look at does not care how one makes conclusions about conditional independence relations, but only that one find an answer.

It does, however, care how accurate these answers are. And they are not accurate. When we come to generate simulated data in section 10, these data will be normally distributed and linearly associated. (Remember those functions from page 6? Each function will be a linear combination of its inputs, and the U_i will be normally distributed.) When it comes to testing the data for conditional independencies, I use existing functions from the well-known statistics package R to do this, rather than implementing the tests myself.

The test is a form of *null hypothesis test*. It works by assuming the independence is true (that's the null hypothesis) and computing the probability of getting data 'at least as associated as' the observed data under that hypothesis. One is left with a probability, and if the probability is very small, one rejects the null hypothesis. In my case, I reject the null hypothesis (and conclude that the independence does *not* hold) if this probability (referred to in my code as a 'p-value') is less than 0.05.

But this will happen 5% of the time that the null hypothesis is true, resulting in a *type I error*: concluding that the null hypothesis is false when it is not (a false positive). With only four variables in the dataset, the number of conditional independence tests that can be run is ${}^4\mathsf{C}_2 \times (4-2)^2 = 24$, so even on such a small graph the expected number of type I errors is uncomfortably high, and with five or six variables it becomes a near certainty.

A false negative (type II error) occurs when the p-value found happens by chance to be unusually high, and the null hypothesis is not rejected when it should have been. It is impossible to work out the probability of a type II error unless you already know the probability of the null hypothesis being true or false beforehand, which in our application is related to knowing the likely sparseness or density of the graph.

8 Method

The Algorithm

The basic details of the algorithm that is the focus of this report were first introduced by Verma and Pearl (1990), but being unable to find a copy of that text, I quote from Pearl (2009):

"***IC Algorithm*** **(Inductive Causation)**

Input: $\hat{P}$, a stable distribution on a set V of variables.
Output: a pattern $H(\hat{P})$ compatible with $\hat{P}$.

1. For each pair of variables a and b in V, search for a set S_{ab} such that $(a \perp\!\!\!\perp b \mid S_{ab})$ holds in $\hat{P}$ – in other words, a and b should be independent in $\hat{P}$, conditioned on S_{ab}. Construct an undirected graph G such that vertices a and b are connected with an edge if and only if no set S_{ab} can be found.
2. For each pair of nonadjacent variables a and b with a common neighbor c, check if $c \in S_{ab}$.
   - If it is, then continue.
   - If it is not, then add arrowheads pointing at c (i.e. $a \rightarrow c \leftarrow b$).
3. In the partially directed graph that results, orient as many of the undirected edges as possible subject to two conditions: (i) Any alternative orientation would yield a new v-structure; or (ii) Any alternative orientation would yield a directed cycle."

In the same paper, the authors proved the theorem that I showed on page 13, the upshot of which is that phase 2 above is as clever as is possible; there are no other combinations of statistical conditions that can lead us to conclude anything else about the causal structure. But they didn't present the full details of phase 3 until two years later (Verma and Pearl, 1992). In that time, Spirtes and Glymour (1991) had found an efficient implementation of phase 1 that narrows the search for S_{ab} considerably. This has the result of a) making the algorithm a lot faster than it would be otherwise, and b) significantly reducing the number of errors of the sort that I described in section 7.

The full algorithm is therefore:

Input: A probability distribution on n variables

Phase 1:

- Initialise G to the complete, undirected graph on n vertices. Associate each vertex with one of the variables in the distribution.
- For $j = 0$ to $n - 2$:
  - For each pair of vertices (A, B) (or equivalently, variables):

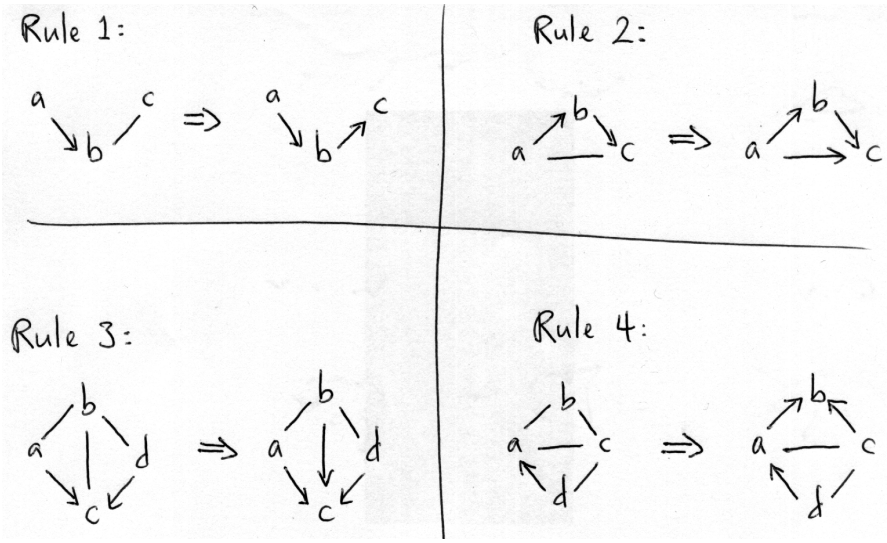
    Search for a set $S_{AB} \subseteq (\text{neighbours}(A) \cup \text{neighbours}(B)) \setminus \{A, B\}$ with $|S_{AB}| = j$ such that $A \perp\!\!\!\perp B \mid S_{AB}$ in the distribution. (By convention $A \perp\!\!\!\perp B \mid \varnothing \iff A \perp\!\!\!\perp B$.) If such a set can be found, record it and delete the edge A — B from G.

Phase 2:

- For each subgraph A — B — C with A not adjacent to C in G:
  - If $B \notin S_{AC}$, orient the edges as $A \rightarrow B \leftarrow C$.

Phase 3:

- Repeatedly apply these four rules until no more edges can be oriented:



Output: The partially directed graph G

Rule 1 of phase 3 applies because if $C \rightarrow B$ then we ought to have detected that in phase 2. Rule 2 applies because we don't allow cycles. Rule 3 applies because at least one of $B \rightarrow A$ and $B \rightarrow D$ must be the case, because otherwise we would have a V-structure $A \rightarrow B \leftarrow D$, and so we must have $B \rightarrow C$ in order to avoid a cycle. In rule 4, $A \rightarrow B$ is already given by rule 1. Then $B \rightarrow C \rightarrow D$ produces a cycle and $B \rightarrow C \leftarrow D$ produces a new V-structure, so we must have $B \leftarrow C$.

Meek (1995) showed that phase 3 is 'complete', that is, that repeated application of these rules will eventually lead to the Markov equivalence class of graphs with the skeleton found in step 1 that have the V-structures detected in step 2.

My Implementation

My implementation of this can be found in Appendix A.2. I have written it in Python, but when I come to require a function for testing for conditional independence, I call functions from the R packages `ggm` and `stats` via the Python package *RPy2*.

Once you have got past the first fourteen lines of `import` instructions, the code really begins with the function `causal_search()` on line 111. This has one argument that must be supplied, `table`, a table of data containing as many columns as there are variables and as many rows as the number of samples from the population. `threshold` is the p-value cutoff to be used in phase 1. The next two parameters control which version of the algorithm is to be used, as I implemented two slight modifications that I will describe in the next section. The basic PC algorithm as described above can be selected by keeping the default values of `None` and `False`. `verbose=True` instructs the computer to output more information about the decisions it is making as it goes along.

Lines 114–118 initialise some variables and data structures. Crucially, `Gu` (standing for 'undirected G') is initialised to the complete graph. Then the covariance matrix `S` is computed and the function `phase_1()` is called, which takes us to line 17.

Lines 19–20 and 43–48 are to do with parallel processing, so ignore those. Line 21 is the 'for $j = 0$ to $n - 2$' from phase 1. (`range(k)` is Python-speak for the sequence $0, 1, \ldots k - 1$.) Then, for each pair of variables `(a, b)`, the function `test_edge(a, b)` is called, which looks for separating sets of length j for (a, b). Depending on the version of the algorithm in effect, it either stops searching as soon as it finds one (line 99) or it tries all possible sets (of length j) to find the best one.

Again, lines 56, 88–89, 95–96, 102–103 and 107–108 are to do with parallel processing. Due to the fact that it is running in as many separate processes as you have CPUs on your machine, the function `test_edge()` is unable to delete edges from the graph or update the table of separating sets, so it returns the necessary information (lines 58, 65 and 105) and updating the graph and so on is done by lines 29–33 (in the 'master' process).

Then j is incremented, and as soon as $j > n - 2$, phase 1 finishes, and the graph is converted to a datatype capable of recording direction (line 134) in preparation for phase 2 (line 137).

As the DiGraph class provided by the Python package NetworkX was not designed to store the various different kinds of edge direction that my algorithm has to work with, I have had to use it in a slightly non-standard way. A completely undirected edge is represented by directed edges pointing both ways. But then phase 2 sometimes results in contradictions, where it believes an edge to be pointing one way and simultaneously believes it to be pointing the other way (i.e. contradictory V-structures were detected implying both directions). This has to be distinguished from a truly undirected edge when I get to stage 3, so I indicate it by keeping both edges but by attaching an arrowhead to each one. From phase 3 onwards, a directed edge is indicated by the existence of only one (directed) edge between the two nodes, which by this point will have its 'arrow' flag set. During phase 2, no edges are deleted but some are marked with arrowheads whenever a V-structure is detected. I then remove

the extraneous edges after phase 2, in lines 155–160.

To assist with all this, I defined some extra functions and constants in the file edgetypes.py, which is given in Appendix A.3. The `orient()` function there is used in phase 3 but not during phase 2 (cf. lines 152–3 of causes.py).

The code for phase 3 (lines 177-240) of causes.py should be fairly self-explanatory, except that I have to adapt rules 2, 3 and 4 to cope with the contradictions just mentioned. This was done by examining the explanations for those rules. Wherever the reasoning refers to a V-structure not having been detected (as in rule 1) I require that the relevant edge be truly undirected. But where the rules apply just because we don't yet know which way an edge is directed (as in rule 2) I also allow it to be directed both ways. This can be seen by comparing line 186 (for rule 1) with line 197 (for rule 2).

9 Refinements

• mention the difference between IC and PC

• The methods I have implemented are:

1. PC algorithm

2. the 'two-pass' PC method that I showed you last time (runs PC twice, first with a high threshold of about 0.3 or 0.4, then with the final threshold, i.e. about 0.5. Reduces chaotic sensitivity to the ordering of the variables)

3. version that records some sort of confidence along with the separating sets. this could be used to make guesses when contradictions are found, or just to output a confidence for each edge in the final graph. I might cite Kalisch, Mächler, Colombo, Maathuis, and Bühlmann at some point

4. combination of the above two

• The above four I will describe in detail, explaining how they work, pointing out the lines of code from appendix A.2 that encode them, etc

• At some point, I modified the code to run in parallel on multiple processors, so that it would be faster. This required a slight change in the algorithm, meaning that removal of edges is delayed until j is incremented, so which nodes are neighbours of other nodes is delayed until we increase the search size of the separating sets. The main upshot of this is that the algorithm is no longer sensitive to the ordering of the variables

• mention that there exists a method to detect latent variables. cite the relevant paper

10 Simulation

• discussion of parameters in distribution generation that affect: a) faithfulness and b) reliability of conditional independence tests. Explanation that my simulation tells you very little about how reliable such an approach would be in the

real world due to the somewhat arbitrary way I have chosen the parameters that affect (b).

• explain the source code that runs the simulation, but not too much. this is in appendix A.1

For each version of the algorithm in section 9, I have tested the following:

• the following graphs under one random distribution. (Apart from the first one, some edges are oriented randomly, if the test will work either way.)

• A -> B <- C

• a graph designed to test rule 3

• a graph designed to test rule 4. sometimes it (correctly) activates rule 3 as well.

• a graph that looks a bit like a dumbbell, designed to test both rules 3 and 4. due to the complexity of the graph, sometimes it activates the rules on some combination of nodes other than the ones I had deliberately set up rule 3 and rule 4 situations on. this is OK

• I tested one of each of the above, under one random distribution. I did this three times and then picked the most interesting results. All the results were typical. These are in appendix B.1.

• For $n = 4$ to 23, I tested three random connected graphs on n vertices, under three random distributions. I then had the computer aggregate all the results. I then did all that three times. It took about 24 hours, even parallelised to run on four processors simultaneously. These results are shown raw in appendix B.2 and graphically on pages 20–21.

• For each test, I recorded a) a normalised measure of edge existence accuracy, b) a normalised measure of edge orientation accuracy and c) the time it took. For the four specific graphs, I have also included (in appendix B.1) the computer's output on what decisions it was making, and what its reasoning was. Due to the parallel processing, some of these decisions appear in random order (but this does not affect the outcome)

11 Analysis

• All I'm doing here is introducing Figure 2, Figure 3 and Figure 4, and detailing my inevitable conclusions.

12 Summary & Outlook

• V. says this section should be written on the last day, when it is too late to change anything else

• mention algorithm to detect latent variables where causal sufficiency is violated, if not already mentioned

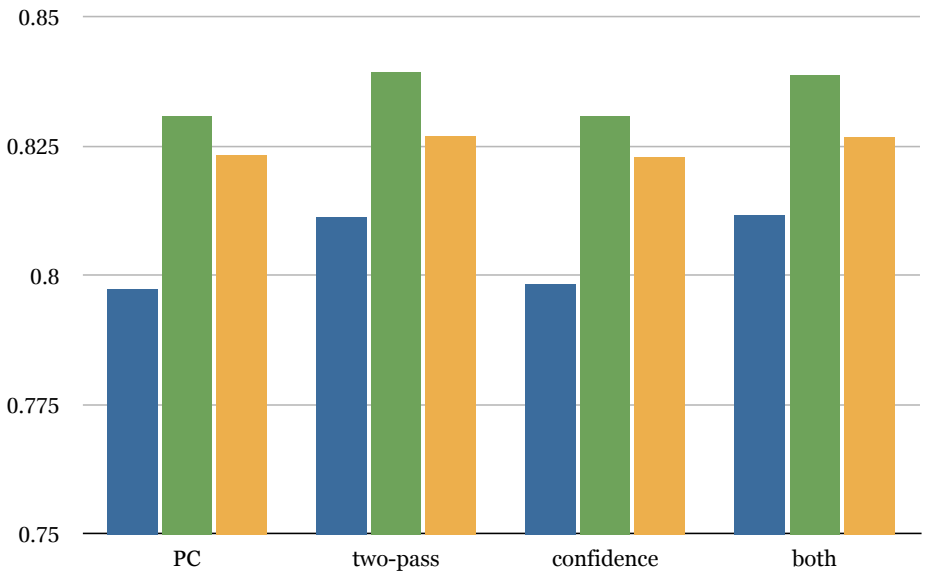


Figure 2: Edge Existence Accuracy

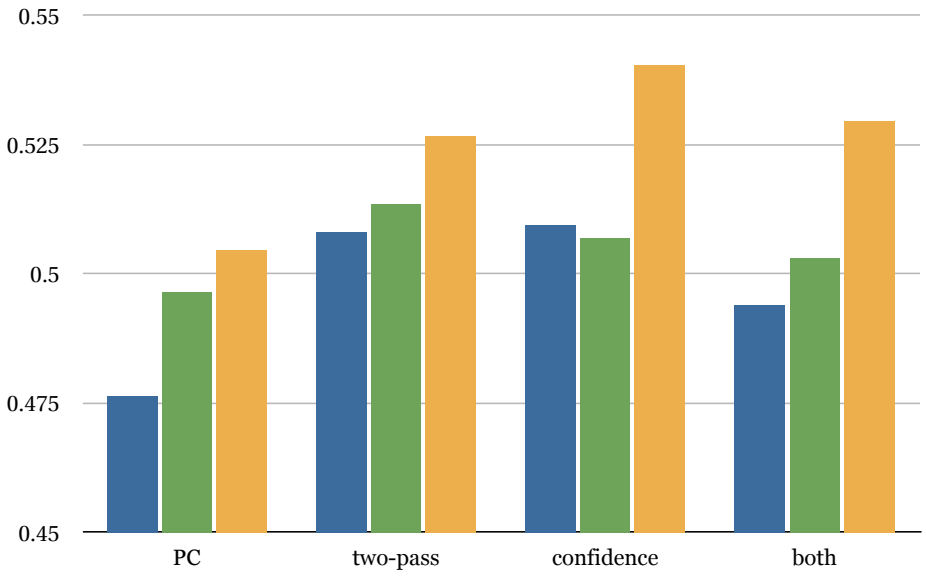


Figure 3: Edge Orientation Accuracy

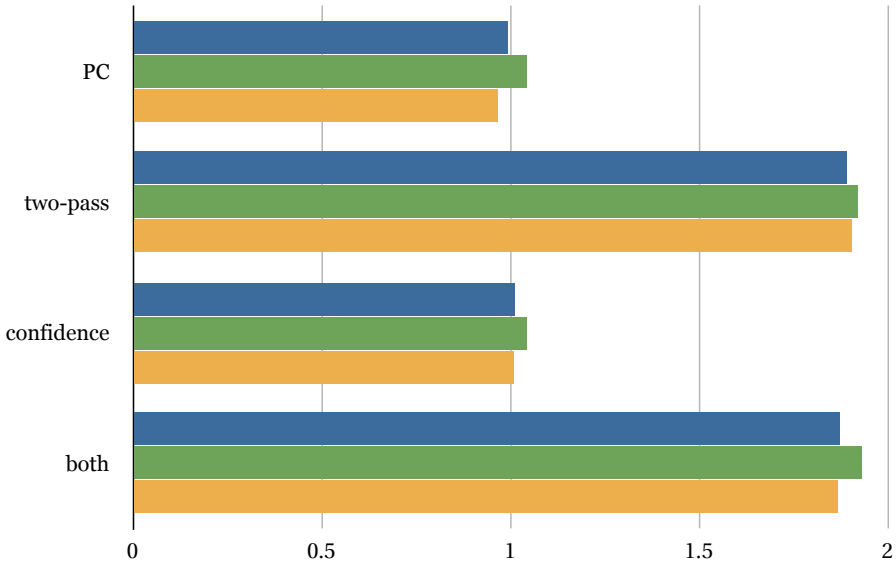


Figure 4: Relative Time Taken

• maybe mention that even two-pass test will sometimes find separating sets that are not taken from the neighbours of the nodes they separate

• explain probability of bugs in code

13 Null Section

• this section doesn't exist anymore. I might write an appendix explaining how to run my software on your own computer.

Required software:

- Python 2.7 (no earlier or later versions will work, but 2.7 is installed as the default on nearly every up-to-date OS except Windows)
- NumPy (installed by default certainly on Mac OS X and probably on many Linux distributions and other distributions of Python): `http://pypi.python.org/pypi/numpy/1.6.0` for the Windows version
- NetworkX: `http://networkx.lanl.gov/`
- R: `http://www.stats.bris.ac.uk/R/`
- RPy2: `http://rpy.sourceforge.net/rpy2.html`
- The R packages 'ggm' and 'stats', installable by typing, e.g. `install.packages("stats")` at the R prompt

A Python Source Code

A.1 study.py

```
#! /usr/bin/env python

from copy import deepcopy
from random import *
from itertools import *
from time import clock
import numpy as np
import networkx as nx
from edgetypes import *
from causes import causal_search


def blank():
  return {
    'pscl': {'existence': 0.0, 'orientation': 0.0, 'time': 0.0},
    'twop': {'existence': 0.0, 'orientation': 0.0, 'time': 0.0},
    'conf': {'existence': 0.0, 'orientation': 0.0, 'time': 0.0},
    'both': {'existence': 0.0, 'orientation': 0.0, 'time': 0.0}
  }


def few_tests(graphs_per_test=1, distns_per_g=3, verbosity=2):
  accu_one = blank()
  print '-' * 62
  print "\nreally simple graph:"
  Gtrue = nx.empty_graph(3, create_using=nx.DiGraph())
  nodes = range(3)
  shuffle(nodes)
  [a, b, c] = nodes
  Gtrue.add_edges_from([(a, b), (c, b)])
  print 'Gtrue.edges() =', Gtrue.edges()
  test_one_graph(Gtrue, accu_one, distns_per_g, 250, verbosity)
  print_accuracy(accu_one, 1)

  accu_two = blank()
  for k in range(graphs_per_test):
    print '-' * 62
    print "\nrule 3 test, graph", str(k+1) + ':'
    Gtrue = nx.empty_graph(4, create_using=nx.DiGraph())
    nodes = range(4)
    shuffle(nodes)
    [a, b, c, d] = nodes
    Gtrue.add_edges_from([(a, d), (b, d), (c, d)])
    Gtrue.add_edge(*choice([(a, b), (b, a)]))
    Gtrue.add_edge(*choice([(b, c), (c, b)]))
```

```
      print '(a, b, c, d) =', (a, b, c, d)
      print 'Gtrue.edges() =', Gtrue.edges()
      test_one_graph(Gtrue, accu_two, distns_per_g, 10000, verbosity)
    print_accuracy(accu_two, graphs_per_test)

    accu_three = blank()
    for k in range(graphs_per_test):
      print '-' * 62
      print "\nrule 4 test, graph", str(k+1) + ':'
      Gtrue = nx.empty_graph(5, create_using=nx.DiGraph())
      nodes = range(5)
      shuffle(nodes)
      [a, b, c, d, e] = nodes
      Gtrue.add_edges_from([(a, b), (c, b), (d, a)])
      Gtrue.add_edges_from([(c, a), (c, d)])
      Gtrue.add_edge(e, a)  # required in order to detect (d, a)
      # prevents V-structure e -> b <- c:
      Gtrue.add_edge(*choice([(c, e), (e, c)]))
      # a -> b will still be detected by rule 1 due to d -> a
      print '(a, b, c, d, e) =', (a, b, c, d, e)
      print 'Gtrue.edges() =', Gtrue.edges()
      assert nx.is_directed_acyclic_graph(Gtrue)
      test_one_graph(Gtrue, accu_three, distns_per_g, 40000, verbosity)
    print_accuracy(accu_three, graphs_per_test)

    accu_four = blank()
    for k in range(graphs_per_test):
      print '-' * 62
      print "\ncombined test, graph", str(k+1) + ':'
      Gtrue = nx.empty_graph(8, create_using=nx.DiGraph())
      nodes = range(8)
      shuffle(nodes)
      [a3, b3, c3, d3, a4, b4, c4, d4] = nodes
      # start with the rule 3 graph
      Gtrue.add_edges_from([(a3, d3), (b3, d3), (c3, d3)])
      Gtrue.add_edge(*choice([(a3, b3), (b3, a3)]))
      Gtrue.add_edge(*choice([(b3, c3), (c3, b3)]))
      # add the rule 4 graph
      Gtrue.add_edges_from([(a4, b4), (c4, b4), (d4, a4)])
      Gtrue.add_edges_from([(c4, a4), (c4, d4)])
      # adding these two edges between the two subgraphs cannot produce a cycle:
      u = choice([a3, b3, c3, d3])
      Gtrue.add_edges_from([(u, a4), choice([(c4, u), (u, c4)])])
      print '(a3, b3, c3, d3, a4, b4, c4, d4) =', \
            (a3, b3, c3, d3, a4, b4, c4, d4)
      print 'Gtrue.edges() =', Gtrue.edges()
      assert nx.is_directed_acyclic_graph(Gtrue)
      test_one_graph(Gtrue, accu_four, distns_per_g, 90000, verbosity)
    print_accuracy(accu_four, graphs_per_test)
```

```
95
    def many_tests(min_n=6, max_n=15, graphs_per_n=3, distns_per_g=3, verbosity=1):
      number_of_tests = graphs_per_n * len(range(min_n, max_n+1))
      accuracy = blank()
100   for n in range(min_n, max_n+1):
        for j in range(graphs_per_n):
          print '-' * 62
          if verbosity >= 1:
            print
105       print "n =", n, "\tgraph", str(j+1) + ':'
          Gtrue = random_dag(n)
          print 'Gtrue.edges() =', Gtrue.edges()
          test_one_graph(Gtrue, accuracy, distns_per_g, None, verbosity)
      print_accuracy(accuracy, number_of_tests)
110

    def print_accuracy(accuracy, number_of_tests):
      print '-' * 62
      print '#' * 14, 'Overall Accuracy:', '#' * 14
115   for (key, message) in [
        ('pscl', 'basic PC algorithm:'),
        ('twop', 'two-pass PC algorithm:'),
        ('conf', 'using confidence levels to make best guesses:'),
        ('both', 'using two passes and confidence levels:')
120   ]:
        print
        print message
        print 'mean edge existence accuracy   =',
        print accuracy[key][ 'existence' ] / number_of_tests
125     print 'mean edge orientation accuracy =',
        print accuracy[key]['orientation'] / number_of_tests
        print 'total time taken (seconds)     =',
        print accuracy[key]['time']


130
    def test_one_graph(Gtrue, accuracy, batches=1, rows=None, verbosity=1):
      n = Gtrue.number_of_nodes()
      num_edges = Gtrue.number_of_edges()
      if rows is None:
135     rows = int(n**(5.0/3.0) * 100)
      if verbosity >= 2:
        verbose = True
      else:
        verbose = False
140
      for batch in range(1,batches+1):
        if verbosity >= 1:
          print
```

```
        print "n =", n, "\trows =", rows, "\tbatch", str(batch)+':'
145     random_dist(Gtrue)
        if verbosity >= 1:
          print 'Gtrue.nodes(data=True) =', Gtrue.nodes(data=True)
          print 'Gtrue.edges(data=True) =', Gtrue.edges(data=True)
          print
150     data = sample(Gtrue, rows)

        for (test_fn, key) in (
          (test_pc  , 'pscl'),
          (test_twop, 'twop'),
155       (test_conf, 'conf'),
          (test_both, 'both')
        ):  # Now test_fn is one of the four functions test_pc, etc
          # and accuracy[key] is one of accuracy['pscl'], etc

160       accuracy[key]['time'] = accuracy[key]['time'] - clock()
          Guess = test_fn(data, verbosity, verbose)
          accuracy[key]['time'] = accuracy[key]['time'] + clock()
          if verbosity >= 1:
            print 'Guess.edges() =', Guess.edges()
165
          # count edge existence errors
          gtrue_edges = set(Gtrue.to_undirected().edges())
          guess_edges = set(Guess.to_undirected().edges())
          exist_errs = len(gtrue_edges ^ guess_edges)
170       if verbosity >= 1:
            print 'edge existence accuracy   =', \
                  str(num_edges - exist_errs) + '/' + str(num_edges),
          if num_edges > 0:
            exist_accu = float(num_edges - exist_errs) / float(num_edges)
175         if verbosity >= 1:
              print "\t=", exist_accu,
            accuracy[key]['existence'] = \
              accuracy[key]['existence'] + exist_accu / batches
          else:
180         accuracy[key]['existence'] = \
              accuracy[key]['existence'] + 1.0 / batches
          if verbosity >= 1:
            print

185       # determine edge orientation accuracy
          direction_good = 0
          direct_max = num_edges
          for (u, v) in Gtrue.edges_iter():
            direction = edge_type(Guess, u, v)
190         if direction == EdgeT.forward:
              direction_good = direction_good + 1
            elif direction == EdgeT.back:
```

```
            direction_good = direction_good - 1
          elif direction == EdgeT.none:
            direct_max = direct_max - 1
        if verbosity >= 1:
          print 'edge orientation accuracy =', \
                str(direction_good) + '/' + str(direct_max),
        if direct_max > 0:
          direct_accu = float(direction_good) / float(direct_max)
          if verbosity >= 1:
            print "\t=", direct_accu,
          accuracy[key]['orientation'] = \
            accuracy[key]['orientation'] + direct_accu / batches
        else:
          accuracy[key]['orientation'] = \
            accuracy[key]['orientation'] + 1.0 / batches
        if verbosity >= 1:
          print "\n"

    return accuracy


  def test_pc(data, verbosity=2, verbose=True):
    if verbosity >= 1:
      print 'basic PC algorithm:'
    return causal_search(data, 0.05, None, False, verbose)

  def test_twop(data, verbosity=2, verbose=True):
    if verbosity >= 1:
      print 'two-pass PC algorithm:'
    return causal_search(data, 0.05, 1.0/3.0, False, verbose)

  def test_conf(data, verbosity=2, verbose=True):
    if verbosity >= 1:
      print 'using confidence levels to make best guesses:'
    (guess, meta) = causal_search(data, 0.05, None, True, verbose)
    return guess

  def test_both(data, verbosity=2, verbose=True):
    if verbosity >= 1:
      print 'using two passes and confidence levels:'
    (guess, meta) = causal_search(data, 0.05, 1.0/3.0, True, verbose)
    return guess


  def random_dag(n, verbose=False):
    if n < 1:
      raise(ValueError('n = ' + str(n)))

    G = nx.empty_graph(n, create_using=nx.DiGraph())
```

```
    rota = range(n)
    shuffle(rota)
    if verbose:
      print 'rota =', rota

    if n == 1:
      return G

    for k in range(n-1):
      (i, j) = (randint(0, n-2), randint(1, n-1))
      if j > i:
        G.add_edge(rota[i], rota[j])

    for (u, v) in combinations(rota, 2):
      if not G.has_edge(u, v) and random() < 2.0 / ((n-1)*(n-1)):
        G.add_edge(u, v)

    if verbose:
      print 'G.edges() =', G.edges(), '+ [',

    while not nx.is_weakly_connected(G):
      (i, j) = (randint(0, n-2), randint(1, n-1))
      if j > i and not G.has_edge(rota[i], rota[j]):
        if verbose:
          print str((rota[i], rota[j])) + ',',
        G.add_edge(rota[i], rota[j])

    if verbose:
      print ']'

    assert nx.is_directed_acyclic_graph(G)
    return G


  def random_dist(G, nonnegative=False):
    if not nx.is_directed_acyclic_graph(G):
      raise(ValueError('G has cycles: ' + str(nx.simple_cycles(G))))

    for i in G.nodes_iter():
      G.node[i]['mu'] = choice((1, -1)) * lognormvariate(1, 0.2)
      G.node[i]['sigma'] = lognormvariate(0, 0.2)

    for (u, v) in G.edges_iter():
      w = lognormvariate(0, 0.2)
      if not nonnegative and random() < 0.5:
        w = -w
      G[u][v]['weight'] = w
```

```
    return


def sample(G, rows, verbose=False):
  if not nx.is_directed_acyclic_graph(G):
    raise(ValueError('G has cycles: ' + str(nx.simple_cycles(G))))

  for v in G:  # let v be an arbitrary node in G
    if ('mu' not in G.node[v] and 'sigma' not in G.node[v]):
      raise(TypeError('G has no distribution. Call random_dist(G) first.'))
    break

  # rota specifies when the data are generated, in order from causes to effects
  rota = list()
  for (v, d) in G.in_degree_iter():
    if d == 0:
      rota.append(v)

  waiting = list()  # waiting to be added to rota; children of recent
  current = list()  # currently being added to rota; snapshot of waiting
  recent = rota     # recently added to rota; try to add children now

  while len(rota) < len(G):
    for parent in recent:
      for child in G.successors_iter(parent):
        if child not in rota and child not in waiting:
          waiting.append(child)
    recent = []
    current = deepcopy(waiting)  # we can't modify waiting
    for node in current:         # while stepping through it
      if all(parent in rota for parent in G.predecessors_iter(node)):
        rota.append(node)
        recent.append(node)
        waiting.remove(node)

  if verbose:
    print 'rota =', rota
    if rows > 10000:
      print 'generating data...',

  # preprocessing done -- time to generate some data
  table = np.zeros((rows, len(G)))

  for row in range(rows):
    for v in rota:
      (mu, sigma) = (G.node[v]['mu'], G.node[v]['sigma'])
      for p in G.predecessors_iter(v):
        mu = mu + G.edge[p][v]['weight'] * table[row][p]
      table[row][v] = gauss(mu, sigma)
```

```
340
      if verbose and rows > 10000:
        print 'done.'

      return table
345

    if __name__ == '__main__':
      print 'few_tests(graphs_per_test=1, distns_per_g=1, verbosity=2):'
      few_tests(1, 1, 2)
350   print "\n" + ('=' * 62) + "\n"
      print 'many_tests(min_n=4, max_n=23, graphs_per_n=3,', \
            'distns_per_g=3, verbosity=0):'
      many_tests(4, 23, 3, 3, 0)
```

A.2 causes.py

```
from itertools import *
import sys
from multiprocessing import Pool, Lock
import numpy as np
import networkx as nx
from edgetypes import *
import rpy2.robjects as robjects
import rpy2.robjects.numpy2ri
from rpy2.robjects.vectors import IntVector
from rpy2.robjects.packages import importr
r = robjects.r
ggm = importr("ggm")          # R: library(ggm)
stats = importr("stats")      # R: library(stats)
lock = Lock()


def phase_1(Gu, table, meta, args, final):
  (rows, n) = table.shape
  pool = Pool()
  try:
    for j in range(n-1):

      # multiple CPU stuff
      result = pool.imap_unordered(test_edge, (
        ((a, b), (Gu, table, meta[a][b], final, rows, n, j), args)
        for (a, b) in combinations(Gu.nodes(), 2)
      ), 4)

      for (a, b, trigger, pvalue, sepset) in result:
        if pvalue > meta[a][b]['pvalue'] and (final or trigger):
          meta[a][b]['pvalue'] = pvalue
          meta[a][b]['sepset'] = sepset
        if trigger and Gu.has_edge(a, b):
          Gu.remove_edge(a, b)

      # the above is somewhat equivalent to:
      # for (a, b) in combinations(Gu.nodes(), 2):
      #   test_edge(( (a, b), (Gu, table, meta[a][b], final, rows, n, j),
      #               (S, threshold, confidence, verbose) ))
      # except replace the line 'trigger = True' in test_edge()
      # with if Gu.has_edge(a, b): 'Gu.remove_edge(a, b)

  finally:
    try:
      lock.release()
    except Exception:
      pass
```

```
    pool.terminate()

50
def test_edge((
  (a, b),
  (Gu, table, meta_ab, final, rows, n, j),
  (S, threshold, confidence, verbose)
)):
  try:
    if (not Gu.has_edge(a, b)) and (j > len(meta_ab['sepset'])):
      return (a, b, False, 0, tuple())

60  a_nb = set(Gu.neighbors(a))
    b_nb = set(Gu.neighbors(b))
    neighbours = (a_nb | b_nb) - set((a, b))

    if j > len(neighbours):
65    return (a, b, False, 0, tuple())

    trigger = False
    meta_ab['sepset'] = tuple()

70  for others in combinations(neighbours, j):

      if len(others) == 0:
        # R: pvalue = cor.test(table[,a], table[,b])$p.value
        pvalue = stats.cor_test(table[:,a], table[:,b])[2][0]
75    else:
        u = (a+1, b+1) + tuple((o+1) for o in others)
        # R: parcor = pcor(u, S)
        parcor = ggm.pcor(IntVector(u), S)
        # R: pvalue = pcor.test(parcor, length(others), rows)$pvalue
80      pvalue = ggm.pcor_test(parcor, len(others), rows)[2][0]

      if pvalue > meta_ab['pvalue']:
        if final or pvalue > threshold:
          meta_ab['pvalue'] = pvalue
85        meta_ab['sepset'] = others

        if verbose and (pvalue > threshold or (final and pvalue > 1e-9)):
          lock.acquire(True)
          sys.stdout.flush()
90        print "test", a, "⫫", b, "|", str(others)+":\t", "p =", pvalue,

        if pvalue > threshold: # if a and b independent conditional on others
          if verbose:
            print "\t", "=> independent"
95          sys.stdout.flush()
            lock.release()
```

```
            trigger = True
            if not confidence:
              break
100       elif verbose and final and pvalue > 1e-9:
            print
            sys.stdout.flush()
            lock.release()

105     return (a, b, trigger, meta_ab['pvalue'], meta_ab['sepset'])

    except KeyboardInterrupt:
      pass


110
  def causal_search(table, threshold=0.05, firstpass=None,
                    confidence=False, verbose=False):

    (rows, n) = table.shape
115 Gu   = nx.complete_graph(n)   # Gu is undirected
    meta = nx.complete_graph(n)   # data structure to hold separating sets
    for (a, b) in meta.edges_iter():
      meta[a][b]['pvalue'] = 0    # highest p-value found so far


120
    ## Phase 1: find conditional independencies and delete edges

    # compute the estimated covariance matrix and store it in S
    S = r.var(table)            # R: S = var(table)
125
    if firstpass is not None:
      if verbose:
        print 'first pass:'
      phase_1(Gu, table, meta, (S, firstpass, confidence, verbose), final=False)
130   if verbose:
        print 'second pass:'
    phase_1(Gu, table, meta, (S, threshold, confidence, verbose), final=True)

    Gd = nx.DiGraph(Gu)  # Gd is Gu, directed, with edges pointing both ways
135

    ## Phase 2: identify V-structures

    if n < 3:
140   if confidence:
        return (Gd, meta)
      else:
        return Gd

145 for (a, b, c) in permutations(Gu.nodes(), 3):
```

```
      if ( Gu.has_edge(a, b) and Gu.has_edge(c, b) and not Gu.has_edge(a, c) and
           b not in meta[a][c]['sepset'] and
           ('arrow' not in Gd[a][b] or 'arrow' not in Gd[c][b])
         ):
        if verbose:
          print a,'->',b,'<-',c,'because',a,'⫫',c,'|',meta[a][c]['sepset']
        Gd[a][b] = {'arrow': True, 'pvalue': meta[a][c]['pvalue']}
        Gd[c][b] = {'arrow': True, 'pvalue': meta[a][c]['pvalue']}

    for (u, v) in Gu.edges_iter():
      direction = edge_type(Gd, u, v)
      if direction == EdgeT.forward:
        Gd.remove_edge(v, u)
      elif direction == EdgeT.back:
        Gd.remove_edge(u, v)


    ## Phase 2a: fix orientation contradictions
    if confidence:
      for (u, v) in Gu.edges_iter():
        if edge_type(Gd, u, v) == EdgeT.both:
          if Gd[u][v]['pvalue'] > Gd[v][u]['pvalue']:
            if verbose:
              print u, '->', v, 'more likely than', u, '<-', v
            orient(Gd, u, v)
          elif Gd[v][u]['pvalue'] > Gd[u][v]['pvalue']:
            if verbose:
              print v, '->', u, 'more likely than', v, '<-', u
            orient(Gd, v, u)


    ## Phase 3: orient remaining edges if possible

    change = True

    while change:
      change = False

      for (a, b, c) in permutations(Gd.nodes(), 3):
        if (edge_type(Gd, a, b) == EdgeT.forward    and
            edge_type(Gd, b, c) == EdgeT.undirected and
            edge_type(Gd, a, c) == EdgeT.none):
          if verbose:
            print b, '->', c, 'because', a, '->', b, \
                  'and no V-structure', a, '->', b, '<-', c
          orient(Gd, b, c)
          change = True

      for (a, b, c) in permutations(Gd.nodes(), 3):
```

```
195         if (edge_type(Gd, a, b) == EdgeT.forward     and
                edge_type(Gd, b, c) == EdgeT.forward     and
                edge_type(Gd, a, c) in (EdgeT.undirected, EdgeT.both)):
              if verbose:
                print a,'->',c,'because',a,'->',b,'->',c,'and cycles not allowed'
200           orient(Gd, a, c)
              change = True

          if n < 4:
            continue
205
          for (a, b, c, d) in permutations(Gd.nodes(), 4):
            if (edge_type(Gd, a, b) in (EdgeT.undirected, EdgeT.both) and
                edge_type(Gd, b, c) == EdgeT.undirected and
                edge_type(Gd, c, d) == EdgeT.forward    and
210             edge_type(Gd, d, a) == EdgeT.back       and
                edge_type(Gd, a, c) == EdgeT.none       and
                edge_type(Gd, b, d) in (EdgeT.undirected, EdgeT.both)):
              if verbose:
                print b, '->', d, 'by rule 3 with [a, c] =', [a, c]
215           orient(Gd, b, d)
              change = True

          for (a, b, c, d) in permutations(Gd.nodes(), 4):
            if (edge_type(Gd, a, b) not in (EdgeT.none, EdgeT.back) and
220             edge_type(Gd, c, b) not in (EdgeT.none, EdgeT.back) and
                edge_type(Gd, c, d) in (EdgeT.undirected, EdgeT.both) and
                edge_type(Gd, d, a) == EdgeT.forward    and
                edge_type(Gd, a, c) != EdgeT.none       and
                edge_type(Gd, b, d) == EdgeT.none):
225           if (edge_type(Gd, b, c) == EdgeT.both and
                  edge_type(Gd, c, d) == EdgeT.both):
                continue
              if (edge_type(Gd, a, b) == EdgeT.forward and
                  edge_type(Gd, c, b) == EdgeT.forward):
230             continue
              if verbose:
                print a, '->', b, '<-', c, 'by rule 4 with d =', d
              orient(Gd, a, b)
              orient(Gd, c, b)
235           change = True

        if confidence:
          return (Gd, meta)
        else:
240       return Gd
```

A.3 edgetypes.py

```
from enum import Enum
import networkx as nx


EdgeT = Enum('none', 'undirected', 'forward', 'back', 'both')

def edge_type(G, a, b):
  if not isinstance(G, (nx.DiGraph, nx.MultiDiGraph)):
    raise(TypeError('G is not a directed graph.'))

  if not ( G.has_edge(a, b) or G.has_edge(b, a) ):
    return EdgeT.none

  if G.has_edge(a, b) and G.has_edge(b, a):
    if 'arrow' not in G[a][b] and 'arrow' not in G[b][a]:
      return EdgeT.undirected
    if 'arrow' in G[a][b] and 'arrow' in G[b][a]:
      return EdgeT.both

  if G.has_edge(a, b) and 'arrow' in G[a][b] and (
      not G.has_edge(b, a) or 'arrow' not in G[b][a] ):
    return EdgeT.forward
  if G.has_edge(b, a) and 'arrow' in G[b][a] and (
      not G.has_edge(a, b) or 'arrow' not in G[a][b] ):
    return EdgeT.back

  assert False


def orient(G, u, v):
  direction = edge_type(G, u, v)
  if direction == EdgeT.forward:
    return
  if direction not in (EdgeT.undirected, EdgeT.both):
    raise(RuntimeError(
      'Cannot orient '+ str(u) +' - '+ str(v) +' as '+ str(u) +' -> '+ str(v)
    ))
  G[u][v]['arrow'] = True
  G.remove_edge(v, u)
```

B Results

B.1 few_tests()

```
1  few_tests(graphs_per_test=1, distns_per_g=1, verbosity=2):
```

```
   really simple graph:
5  Gtrue.edges() = [(0, 1), (2, 1)]

   n = 3   rows = 250      batch 1:
   Gtrue.nodes(data=True) = [(0, {'mu': -2.4688053134710573, '
       sigma': 0.93002558479706887}), (1, {'mu':
       -2.8281837644124033, 'sigma': 0.86486109746092277}), (2, {'
       mu': 3.1715244389987269, 'sigma': 0.782077286706235})]
   Gtrue.edges(data=True) = [(0, 1, {'weight':
       -0.77904232736013845}), (2, 1, {'weight':
       0.82851789176197532})]
10
   basic PC algorithm:
   test 0 ⫫ 2 | ():       p = 0.246276441985      => independent
   0 -> 1 <- 2 because 0 ⫫ 2 | ()
   Guess.edges() = [(0, 1), (2, 1)]
15 edge existence accuracy   = 2/2         = 1.0
   edge orientation accuracy = 2/2         = 1.0

   two-pass PC algorithm:
   first pass:
20 second pass:
   test 0 ⫫ 2 | ():       p = 0.246276441985      => independent
   0 -> 1 <- 2 because 0 ⫫ 2 | ()
   Guess.edges() = [(0, 1), (2, 1)]
   edge existence accuracy   = 2/2         = 1.0
25 edge orientation accuracy = 2/2         = 1.0

   using confidence levels to make best guesses:
   test 0 ⫫ 2 | ():       p = 0.246276441985      => independent
   0 -> 1 <- 2 because 0 ⫫ 2 | ()
30 Guess.edges() = [(0, 1), (2, 1)]
   edge existence accuracy   = 2/2         = 1.0
   edge orientation accuracy = 2/2         = 1.0

   using two passes and confidence levels:
35 first pass:
   second pass:
   test 0 ⫫ 2 | ():       p = 0.246276441985      => independent
   0 -> 1 <- 2 because 0 ⫫ 2 | ()
   Guess.edges() = [(0, 1), (2, 1)]
40 edge existence accuracy   = 2/2         = 1.0
```

```
edge orientation accuracy = 2/2          = 1.0

-------------------------------------------------------------------------------
############## Overall Accuracy: ##############

basic PC algorithm:
mean edge existence accuracy   = 1.0
mean edge orientation accuracy = 1.0
total time taken (seconds)     = 0.035887

two-pass PC algorithm:
mean edge existence accuracy   = 1.0
mean edge orientation accuracy = 1.0
total time taken (seconds)     = 0.03411

using confidence levels to make best guesses:
mean edge existence accuracy   = 1.0
mean edge orientation accuracy = 1.0
total time taken (seconds)     = 0.018832

using two passes and confidence levels:
mean edge existence accuracy   = 1.0
mean edge orientation accuracy = 1.0
total time taken (seconds)     = 0.035281
-------------------------------------------------------------------------------

rule 3 test, graph 1:
(a, b, c, d) = (1, 2, 0, 3)
Gtrue.edges() = [(0, 2), (0, 3), (1, 3), (2, 1), (2, 3)]

n = 4    rows = 10000     batch 1:
Gtrue.nodes(data=True) = [(0, {'mu': -2.1193126226685703, '
    sigma': 1.012959090901004}), (1, {'mu':
    -4.1868789296938198, 'sigma': 1.2733076209627052}), (2, {'
    mu': -2.2224013764501414, 'sigma': 1.1280650488896926}),
    (3, {'mu': 2.6484178208841493, 'sigma':
    0.78709552772276459})]
Gtrue.edges(data=True) = [(0, 2, {'weight':
    -0.6244671497927553}), (0, 3, {'weight':
    1.0442039011477304}), (1, 3, {'weight': 1.149938489288763})
    , (2, 1, {'weight': -1.1991701654756353}), (2, 3, {'weight
    ': -0.78495733545713553})]

basic PC algorithm:
test 0 ⫫ 1 | (2,):     p = 0.284900137543      => independent
0 -> 3 <- 1 because 0 ⫫ 1 | (2,)
2 -> 3 by rule 3 with [a, c] = [0, 1]
Guess.edges() = [(0, 2), (0, 3), (1, 2), (1, 3), (2, 0), (2,
    1), (2, 3)]
```

```
edge existence accuracy   = 5/5          = 1.0
edge orientation accuracy = 3/5          = 0.6

two-pass PC algorithm:
first pass:
second pass:
test 0 ⫫ 1 | (2,):     p = 0.284900137543      => independent
0 -> 3 <- 1 because 0 ⫫ 1 | (2,)
2 -> 3 by rule 3 with [a, c] = [0, 1]
Guess.edges() = [(0, 2), (0, 3), (1, 2), (1, 3), (2, 0), (2,
    1), (2, 3)]
edge existence accuracy   = 5/5          = 1.0
edge orientation accuracy = 3/5          = 0.6

using confidence levels to make best guesses:
test 0 ⫫ 1 | (2,):     p = 0.284900137543      => independent
0 -> 3 <- 1 because 0 ⫫ 1 | (2,)
2 -> 3 by rule 3 with [a, c] = [0, 1]
Guess.edges() = [(0, 2), (0, 3), (1, 2), (1, 3), (2, 0), (2,
    1), (2, 3)]
edge existence accuracy   = 5/5          = 1.0
edge orientation accuracy = 3/5          = 0.6

using two passes and confidence levels:
first pass:
second pass:
test 0 ⫫ 1 | (2,):     p = 0.284900137543      => independent
0 -> 3 <- 1 because 0 ⫫ 1 | (2,)
2 -> 3 by rule 3 with [a, c] = [0, 1]
Guess.edges() = [(0, 2), (0, 3), (1, 2), (1, 3), (2, 0), (2,
    1), (2, 3)]
edge existence accuracy   = 5/5          = 1.0
edge orientation accuracy = 3/5          = 0.6

-------------------------------------------------------------

############## Overall Accuracy: ##############

basic PC algorithm:
mean edge existence accuracy   = 1.0
mean edge orientation accuracy = 0.6
total time taken (seconds)     = 0.035894

two-pass PC algorithm:
mean edge existence accuracy   = 1.0
mean edge orientation accuracy = 0.6
total time taken (seconds)     = 0.058689

using confidence levels to make best guesses:
mean edge existence accuracy   = 1.0
```

```
mean edge orientation accuracy = 0.6
total time taken (seconds)     = 0.034603

using two passes and confidence levels:
mean edge existence accuracy   = 1.0
mean edge orientation accuracy = 0.6
total time taken (seconds)     = 0.057645
```

```
rule 4 test, graph 1:
(a, b, c, d, e) = (0, 4, 2, 3, 1)
Gtrue.edges() = [(0, 4), (1, 0), (2, 0), (2, 1), (2, 3), (2,
    4), (3, 0)]

n = 5    rows = 40000    batch 1:
Gtrue.nodes(data=True) = [(0, {'mu': 2.9838066155177017, '
    sigma': 0.77585883733143746}), (1, {'mu':
    2.3442179767387246, 'sigma': 0.957700439395615}), (2, {'mu
    ': 2.05178592491818, 'sigma': 1.0776267316181181}), (3, {'
    mu': -2.6755916651074609, 'sigma': 0.84735824783542102}),
    (4, {'mu': -2.883988363205106, 'sigma': 1.367733072503923})
    ]
Gtrue.edges(data=True) = [(0, 4, {'weight':
    1.1604687002397529}), (1, 0, {'weight':
    -0.99789875545884943}), (2, 0, {'weight':
    0.80558220690490356}), (2, 1, {'weight':
    0.72325295648218735}), (2, 3, {'weight':
    -1.2673396402227159}), (2, 4, {'weight':
    0.79683882259846173}), (3, 0, {'weight':
    0.63342700457198187})]

basic PC algorithm:
test 2 ⫫ 4 | ():       p = 0.000698072289982
test 1 ⫫ 3 | (2,):     p = 0.906162997574      => independent
test 0 ⫫ 2 | (1,):     p = 0.452841089975      => independent
test 3 ⫫ 4 | (0, 2):   p = 0.869756262709      => independent
test 1 ⫫ 4 | (0, 2):   p = 0.930036569582      => independent
0 -> 3 <- 2 because 0 ⫫ 2 | (1,)
0 -> 4 <- 2 because 0 ⫫ 2 | (1,)
1 -> 0 <- 3 because 1 ⫫ 3 | (2,)
Guess.edges() = [(0, 3), (0, 4), (1, 0), (1, 2), (2, 1), (2,
    3), (2, 4), (3, 0)]
edge existence accuracy   = 6/7         = 0.857142857143
edge orientation accuracy = 4/6         = 0.666666666667

two-pass PC algorithm:
first pass:
test 1 ⫫ 3 | (2,):     p = 0.906162997574      => independent
test 0 ⫫ 2 | (1,):     p = 0.452841089975      => independent
```

```
test 3 ⫫ 4 | (0, 2):   p = 0.869756262709      => independent
test 1 ⫫ 4 | (0, 2):   p = 0.930036569582      => independent
second pass:
test 2 ⫫ 4 | ():       p = 0.000698072289982
0 -> 3 <- 2 because 0 ⫫ 2 | (1,)
0 -> 4 <- 2 because 0 ⫫ 2 | (1,)
1 -> 0 <- 3 because 1 ⫫ 3 | (2,)
Guess.edges() = [(0, 3), (0, 4), (1, 0), (1, 2), (2, 1), (2,
    3), (2, 4), (3, 0)]
edge existence accuracy   = 6/7         = 0.857142857143
edge orientation accuracy = 4/6         = 0.666666666667

using confidence levels to make best guesses:
test 2 ⫫ 4 | ():       p = 0.000698072289982
test 1 ⫫ 3 | (2,):     p = 0.906162997574      => independent
test 0 ⫫ 2 | (1,):     p = 0.452841089975      => independent
test 3 ⫫ 4 | (0, 2):   p = 0.869756262709      => independent
test 1 ⫫ 4 | (0, 2):   p = 0.930036569582      => independent
0 -> 3 <- 2 because 0 ⫫ 2 | (1,)
0 -> 4 <- 2 because 0 ⫫ 2 | (1,)
1 -> 0 <- 3 because 1 ⫫ 3 | (2,)
3 -> 0 more likely than 3 <- 0
Guess.edges() = [(0, 4), (1, 0), (1, 2), (2, 1), (2, 3), (2,
    4), (3, 0)]
edge existence accuracy   = 6/7         = 0.857142857143
edge orientation accuracy = 5/6         = 0.833333333333

using two passes and confidence levels:
first pass:
test 1 ⫫ 3 | (2,):     p = 0.906162997574      => independent
test 0 ⫫ 2 | (1,):     p = 0.452841089975      => independent
test 3 ⫫ 4 | (0, 2):   p = 0.869756262709      => independent
test 1 ⫫ 4 | (0, 2):   p = 0.930036569582      => independent
second pass:
test 2 ⫫ 4 | ():       p = 0.000698072289982
0 -> 3 <- 2 because 0 ⫫ 2 | (1,)
0 -> 4 <- 2 because 0 ⫫ 2 | (1,)
1 -> 0 <- 3 because 1 ⫫ 3 | (2,)
3 -> 0 more likely than 3 <- 0
Guess.edges() = [(0, 4), (1, 0), (1, 2), (2, 1), (2, 3), (2,
    4), (3, 0)]
edge existence accuracy   = 6/7         = 0.857142857143
edge orientation accuracy = 5/6         = 0.833333333333

-------------------------------------------------------------

############## Overall Accuracy: ##############

basic PC algorithm:
mean edge existence accuracy   = 0.857142857143
```

```
    mean edge orientation accuracy = 0.666666666667
    total time taken (seconds)     = 0.137758

    two-pass PC algorithm:
210 mean edge existence accuracy   = 0.857142857143
    mean edge orientation accuracy = 0.666666666667
    total time taken (seconds)     = 0.286312

    using confidence levels to make best guesses:
215 mean edge existence accuracy   = 0.857142857143
    mean edge orientation accuracy = 0.833333333333
    total time taken (seconds)     = 0.151388

    using two passes and confidence levels:
220 mean edge existence accuracy   = 0.857142857143
    mean edge orientation accuracy = 0.833333333333
    total time taken (seconds)     = 0.272511
```

```
225 combined test, graph 1:
    (a3, b3, c3, d3, a4, b4, c4, d4) = (2, 3, 1, 5, 0, 4, 6, 7)
    Gtrue.edges() = [(0, 4), (1, 5), (2, 0), (2, 3), (2, 5), (3,
        1), (3, 5), (6, 0), (6, 2), (6, 4), (6, 7), (7, 0)]

    n = 8   rows = 90000    batch 1:
230 Gtrue.nodes(data=True) = [(0, {'mu': 1.8806516643118609, '
        sigma': 0.82216032369095815}), (1, {'mu':
        -3.5000340776804681, 'sigma': 1.1792953721927175}), (2, {'
        mu': 1.6278204766323161, 'sigma': 1.1029963535730545}), (3,
         {'mu': -2.301923171151719, 'sigma': 1.455735072911132}),
        (4, {'mu': 2.2857613924376539, 'sigma':
        1.1061650895577619}), (5, {'mu': -3.5129920181478593, '
        sigma': 0.95710988997107505}), (6, {'mu':
        3.0798825404882026, 'sigma': 0.66105121614370643}), (7, {'
        mu': 3.6197204859706762, 'sigma': 0.84022920529920464})]
    Gtrue.edges(data=True) = [(0, 4, {'weight':
        0.78590918362388873}), (1, 5, {'weight':
        1.0783454176442311}), (2, 0, {'weight':
        -1.0250544170265679}), (2, 3, {'weight':
        0.93105124297034414}), (2, 5, {'weight':
        -0.99712077315118897}), (3, 1, {'weight':
        0.73337801029807692}), (3, 5, {'weight':
        1.3112105665811107}), (6, 0, {'weight':
        -0.98296928693609831}), (6, 2, {'weight':
        -0.74750758738402368}), (6, 4, {'weight':
        1.8019497100209718}), (6, 7, {'weight':
        0.89231305514057457}), (7, 0, {'weight':
        -1.1522668946371997})]
```

```
    basic PC algorithm:
    test 0 ⫫ 1 | (2,):     p = 0.646640731378      => independent
235 test 0 ⫫ 3 | (2,):     p = 0.509773453134      => independent
    test 0 ⫫ 5 | (2,):     p = 0.888876740784      => independent
    test 1 ⫫ 2 | (3,):     p = 0.775917756983      => independent
    test 1 ⫫ 3 | (5,):     p = 7.96986804525e-05
    test 1 ⫫ 4 | (2,):     p = 0.88547410784       => independent
240 test 1 ⫫ 6 | (2,):     p = 0.0559705230855     => independent
    test 1 ⫫ 7 | (2,):     p = 0.465461362029      => independent
    test 2 ⫫ 7 | (6,):     p = 0.0614075688687     => independent
    test 3 ⫫ 4 | (2,):     p = 0.739370211341      => independent
    test 3 ⫫ 6 | (2,):     p = 0.0548098287832     => independent
245 test 3 ⫫ 7 | (2,):     p = 0.83633769221       => independent
    test 4 ⫫ 5 | (2,):     p = 0.809312143983      => independent
    test 5 ⫫ 6 | (2,):     p = 0.0405652193015
    test 5 ⫫ 7 | (2,):     p = 0.781855787767      => independent
    test 2 ⫫ 4 | (0, 6):   p = 0.919773636857      => independent
250 test 4 ⫫ 7 | (0, 6):   p = 0.257613521896      => independent
    test 5 ⫫ 6 | (1, 2):   p = 0.434653967196      => independent
    1 -> 5 <- 2 because 1 ⫫ 2 | (3,)
    2 -> 0 <- 7 because 2 ⫫ 7 | (6,)
    0 -> 4 because 2 -> 0 and no V-structure 2 -> 0 <- 4
255 3 -> 5 by rule 3 with [a, c] = [1, 2]
    6 -> 0 by rule 3 with [a, c] = [2, 7]
    0 -> 4 <- 6 by rule 4 with d = 2
    Guess.edges() = [(0, 4), (1, 3), (1, 5), (2, 0), (2, 3), (2,
        5), (2, 6), (3, 1), (3, 2), (3, 5), (6, 0), (6, 2), (6, 4),
         (6, 7), (7, 0), (7, 6)]
    edge existence accuracy   = 12/12       = 1.0
260 edge orientation accuracy = 8/12        = 0.666666666667

    two-pass PC algorithm:
    first pass:
    test 0 ⫫ 1 | (2,):     p = 0.646640731378      => independent
265 test 0 ⫫ 3 | (2,):     p = 0.509773453134      => independent
    test 0 ⫫ 5 | (2,):     p = 0.888876740784      => independent
    test 1 ⫫ 2 | (3,):     p = 0.775917756983      => independent
    test 1 ⫫ 4 | (2,):     p = 0.88547410784       => independent
    test 1 ⫫ 6 | (3,):     p = 0.376395231104      => independent
270 test 1 ⫫ 7 | (2,):     p = 0.465461362029      => independent
    test 3 ⫫ 4 | (2,):     p = 0.739370211341      => independent
    test 3 ⫫ 7 | (2,):     p = 0.83633769221       => independent
    test 4 ⫫ 5 | (2,):     p = 0.809312143983      => independent
    test 5 ⫫ 7 | (2,):     p = 0.781855787767      => independent
275 test 2 ⫫ 4 | (0, 6):   p = 0.919773636857      => independent
    test 3 ⫫ 6 | (1, 2):   p = 0.390755949081      => independent
    test 5 ⫫ 6 | (1, 2):   p = 0.434653967196      => independent
    second pass:
    test 0 ⫫ 1 | (3,):     p = 0.947853162789      => independent
```

```
test 1 ⫫ 3 | (5,):     p = 7.96986804525e-05
test 1 ⫫ 7 | (6,):     p = 0.635312443964      => independent
test 2 ⫫ 7 | (6,):     p = 0.0614075688687     => independent
test 3 ⫫ 6 | (2, 5):   p = 0.823194894316      => independent
test 4 ⫫ 7 | (0, 6):   p = 0.257613521896      => independent
test 5 ⫫ 6 | (2, 3):   p = 0.455818669813      => independent
1 -> 5 <- 2 because 1 ⫫ 2 | (3,)
2 -> 0 <- 7 because 2 ⫫ 7 | (6,)
0 -> 4 because 2 -> 0 and no V-structure 2 -> 0 <- 4
3 -> 5 by rule 3 with [a, c] = [1, 2]
6 -> 0 by rule 3 with [a, c] = [2, 7]
0 -> 4 <- 6 by rule 4 with d = 2
Guess.edges() = [(0, 4), (1, 3), (1, 5), (2, 0), (2, 3), (2,
    5), (2, 6), (3, 1), (3, 2), (3, 5), (6, 0), (6, 2), (6, 4),
     (6, 7), (7, 0), (7, 6)]
edge existence accuracy   = 12/12       = 1.0
edge orientation accuracy = 8/12        = 0.666666666667

using confidence levels to make best guesses:
test 0 ⫫ 1 | (2,):     p = 0.646640731378      => independent
test 0 ⫫ 1 | (3,):     p = 0.947853162789      => independent
test 0 ⫫ 3 | (2,):     p = 0.509773453134      => independent
test 0 ⫫ 5 | (2,):     p = 0.888876740784      => independent
test 1 ⫫ 2 | (3,):     p = 0.775917756983      => independent
test 1 ⫫ 3 | (5,):     p = 7.96986804525e-05
test 1 ⫫ 4 | (2,):     p = 0.88547410784       => independent
test 1 ⫫ 6 | (2,):     p = 0.0559705230855     => independent
test 1 ⫫ 6 | (3,):     p = 0.376395231104      => independent
test 1 ⫫ 7 | (2,):     p = 0.465461362029      => independent
test 1 ⫫ 7 | (6,):     p = 0.635312443964      => independent
test 2 ⫫ 7 | (6,):     p = 0.0614075688687     => independent
test 3 ⫫ 4 | (2,):     p = 0.739370211341      => independent
test 3 ⫫ 6 | (2,):     p = 0.0548098287832     => independent
test 3 ⫫ 7 | (2,):     p = 0.83633769221       => independent
test 4 ⫫ 5 | (2,):     p = 0.809312143983      => independent
test 5 ⫫ 6 | (2,):     p = 0.0405652193015
test 5 ⫫ 7 | (2,):     p = 0.781855787767      => independent
test 2 ⫫ 4 | (0, 6):   p = 0.919773636857      => independent
test 4 ⫫ 7 | (0, 6):   p = 0.257613521896      => independent
test 5 ⫫ 6 | (1, 2):   p = 0.434653967196      => independent
test 5 ⫫ 6 | (2, 3):   p = 0.455818669813      => independent
1 -> 5 <- 2 because 1 ⫫ 2 | (3,)
2 -> 0 <- 7 because 2 ⫫ 7 | (6,)
0 -> 4 because 2 -> 0 and no V-structure 2 -> 0 <- 4
3 -> 5 by rule 3 with [a, c] = [1, 2]
6 -> 0 by rule 3 with [a, c] = [2, 7]
0 -> 4 <- 6 by rule 4 with d = 2
Guess.edges() = [(0, 4), (1, 3), (1, 5), (2, 0), (2, 3), (2,
    5), (2, 6), (3, 1), (3, 2), (3, 5), (6, 0), (6, 2), (6, 4),
```

```
    (6, 7), (7, 0), (7, 6)]
edge existence accuracy   = 12/12       = 1.0
edge orientation accuracy = 8/12        = 0.666666666667

using two passes and confidence levels:
first pass:
test 0 ⫫ 1 | (2,):     p = 0.646640731378      => independent
test 0 ⫫ 1 | (3,):     p = 0.947853162789      => independent
test 0 ⫫ 3 | (2,):     p = 0.509773453134      => independent
test 0 ⫫ 5 | (2,):     p = 0.888876740784      => independent
test 1 ⫫ 2 | (3,):     p = 0.775917756983      => independent
test 1 ⫫ 4 | (2,):     p = 0.88547410784       => independent
test 1 ⫫ 6 | (3,):     p = 0.376395231104      => independent
test 1 ⫫ 7 | (2,):     p = 0.465461362029      => independent
test 1 ⫫ 7 | (6,):     p = 0.635312443964      => independent
test 3 ⫫ 4 | (2,):     p = 0.739370211341      => independent
test 3 ⫫ 7 | (2,):     p = 0.83633769221       => independent
test 4 ⫫ 5 | (2,):     p = 0.809312143983      => independent
test 5 ⫫ 7 | (2,):     p = 0.781855787767      => independent
test 2 ⫫ 4 | (0, 6):   p = 0.919773636857      => independent
test 3 ⫫ 6 | (1, 2):   p = 0.390755949081      => independent
test 3 ⫫ 6 | (2, 5):   p = 0.823194894316      => independent
test 5 ⫫ 6 | (1, 2):   p = 0.434653967196      => independent
test 5 ⫫ 6 | (2, 3):   p = 0.455818669813      => independent
second pass:
test 1 ⫫ 3 | (5,):     p = 7.96986804525e-05
test 2 ⫫ 7 | (6,):     p = 0.0614075688687     => independent
test 4 ⫫ 7 | (0, 6):   p = 0.257613521896      => independent
1 -> 5 <- 2 because 1 ⫫ 2 | (3,)
2 -> 0 <- 7 because 2 ⫫ 7 | (6,)
0 -> 4 because 2 -> 0 and no V-structure 2 -> 0 <- 4
3 -> 5 by rule 3 with [a, c] = [1, 2]
6 -> 0 by rule 3 with [a, c] = [2, 7]
0 -> 4 <- 6 by rule 4 with d = 2
Guess.edges() = [(0, 4), (1, 3), (1, 5), (2, 0), (2, 3), (2,
    5), (2, 6), (3, 1), (3, 2), (3, 5), (6, 0), (6, 2), (6, 4),
     (6, 7), (7, 0), (7, 6)]
edge existence accuracy   = 12/12       = 1.0
edge orientation accuracy = 8/12        = 0.666666666667

-------------------------------------------------------------------

############## Overall Accuracy: ##############

basic PC algorithm:
mean edge existence accuracy   = 1.0
mean edge orientation accuracy = 0.666666666667
total time taken (seconds)     = 1.0439

two-pass PC algorithm:
```

```
    mean edge existence accuracy   = 1.0
    mean edge orientation accuracy = 0.666666666667
    total time taken (seconds)     = 1.705056

    using confidence levels to make best guesses:
    mean edge existence accuracy   = 1.0
    mean edge orientation accuracy = 0.666666666667
    total time taken (seconds)     = 0.957592

    using two passes and confidence levels:
    mean edge existence accuracy   = 1.0
    mean edge orientation accuracy = 0.666666666667
    total time taken (seconds)     = 1.696259
```

B.2 many_tests()

I ran the command below three times. When this function is called with `verbosity=0`, it still outputs various lines about its progress and where it has got to. As these are of no use afterwards, I have not included them. Only the aggregate statistics of the three runs are given below. These are shown in graphical format on pages 20–21.

```
many_tests(min_n=4, max_n=23, graphs_per_n=3, distns_per_g=3,
    verbosity=0):
```

```
############## Overall Accuracy: ##############

basic PC algorithm:
mean edge existence accuracy   = 0.797461457277
mean edge orientation accuracy = 0.476271336584
total time taken (seconds)     = 1012.78856

two-pass PC algorithm:
mean edge existence accuracy   = 0.81130748734
mean edge orientation accuracy = 0.508165661644
total time taken (seconds)     = 1928.409295

using confidence levels to make best guesses:
mean edge existence accuracy   = 0.798458418075
mean edge orientation accuracy = 0.509417963567
total time taken (seconds)     = 1031.042469

using two passes and confidence levels:
mean edge existence accuracy   = 0.81179246986
mean edge orientation accuracy = 0.494050795495
total time taken (seconds)     = 1909.931945
```

```
############## Overall Accuracy: ##############

basic PC algorithm:
mean edge existence accuracy   = 0.830845043172
mean edge orientation accuracy = 0.496420573378
total time taken (seconds)     = 1064.399092

two-pass PC algorithm:
mean edge existence accuracy   = 0.839410261805
mean edge orientation accuracy = 0.513519574185
total time taken (seconds)     = 1959.177018

using confidence levels to make best guesses:
mean edge existence accuracy   = 0.830874648385
mean edge orientation accuracy = 0.506881838348
total time taken (seconds)     = 1064.743418
```

```
using two passes and confidence levels:
mean edge existence accuracy   = 0.8389167038
mean edge orientation accuracy = 0.503017352908
total time taken (seconds)     = 1970.044403
```

```
############## Overall Accuracy: ##############

basic PC algorithm:
mean edge existence accuracy   = 0.8233319376
mean edge orientation accuracy = 0.50470089249
total time taken (seconds)     = 984.914435

two-pass PC algorithm:
mean edge existence accuracy   = 0.826951027946
mean edge orientation accuracy = 0.526614228535
total time taken (seconds)     = 1941.597033

using confidence levels to make best guesses:
mean edge existence accuracy   = 0.822934179534
mean edge orientation accuracy = 0.540331057265
total time taken (seconds)     = 1027.287899

using two passes and confidence levels:
mean edge existence accuracy   = 0.826912139019
mean edge orientation accuracy = 0.529466242121
total time taken (seconds)     = 1905.895339
```

References

Hume (1740) David Hume, *An Abstract of a Book lately Published; Entituled, A Treatise of Human Nature, &c. Wherein the Chief Argument of that Book is farther Illustrated and Explained*. London, 1740. Reprinted by various publishers. Sometimes attributed to Adam Smith.

Kalisch et al. (2011) Markus Kalisch, Martin Mächler, Diego Colombo, Marloes Maathuis, and Peter Bühlmann, *Causal Inference using Graphical Models with the R Package* ***pcalg***. Technical report available from `http://cran.r-project.org/web/packages/pcalg/vignettes/pcalgDoc.pdf` or by typing the commands `install.packages("pcalg")` followed by `vignette("pcalgDoc")` at the R prompt, on a system with R installed.

Meek (1995) Christopher Meek, Causal inference and causal explanation with background knowledge. In P. Besnard and S. Hanks (editors), *Uncertainty in Artificial Intelligence 11*, pages 403–410. Morgan Kaufmann, San Francisco, 1995.

Pearl (2009) Judea Pearl, *Causality: Models, Reasoning, and Inference*. Cambridge University Press, New York, 2nd Edition, 2009.

Salmon (1977) Wesley Salmon, An "At-At" Theory of Causal Influence. In *Philosophy of Science*, Vol. 44, No. 2, pages 215–224. The University of Chicago Press, Chicago, June 1977.

Spirtes and Glymour (1991) Peter Spirtes and Clark Glymour, An algorithm for fast recovery of sparse causal graphs. *Social Science Computer Review*, 9(1):62–72, 1991.

Spirtes et al. (2000) Peter Spirtes, Clark Glymour, and Richard Scheines, *Causation, Prediction, and Search*. MIT Press, Cambridge, MA, 2nd Edition, 2000.

Verma and Pearl (1990) Thomas Verma and Judea Pearl, Equivalence and synthesis of causal models. In *Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence*, pages 220–227. Cambridge, MA, July 1990. Also in P. Bonissone, M. Henrion, L.N. Kanal and J.F. Lemmer (editors), *Uncertainty in Artificial Intelligence 6*, Elsevier Sience Publishers, B.V., 255–268, 1991.

- I'm somewhat surprised that I haven't cited Verma and Pearl, *A Theory of Inferred Causation*, 1991. Perhaps I should mention it somewhere.

Verma and Pearl (1992) Thomas Verma and Judea Pearl, An algorithm for deciding if a set of observed independencies has a causal explanation. In D. Dubois, M.P. Wellman, B. D'Ambrosio, and P. Smets (editors), *Proceedings of the Eighth Conference on Uncertainty in Artificial Intelligence*, pages 323–330. Morgan Kaufmann, Stanford, CA, 1992.