MSc Long Unassessed Practical 2

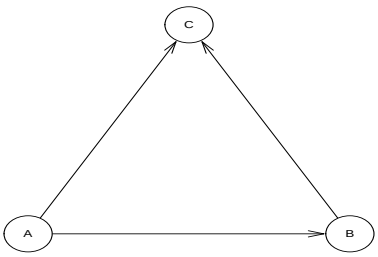
Graphs and Computational Complexity

Many complex statistical models can be represented graphically as *directed acyclic graphs*, in which each node, or vertex, is associated with a random variable and each arc, a directed edge, represents a probabilistic dependence relationship. We are interested here in a directed graph that is *acyclic* so that we can apply the chain rule to write the conditional distributions resulting from the the probabilistic dependencies. (A cycle is a circular pattern of arcs like $A \to B \to C \to A$.)

So for instance if we have

$$P(A, B, C) = P(A)P(B|A)P(C|B, A)$$

we can represent it as:



Some examples of this kind of models are structural model equations, Bayesian networks, hierarchical models, most Monte Carlo and Markov chain Monte Carlo simulations in BUGS/JAGS and vector autoregressive (VAR) time series.

When evaluating various aspects of these models, it is very useful to be able to generate random directed acyclic graphs to use in simulation studies. In particular, we often want to generate directed acyclic graphs with uniform probability since this is considered a non-informative prior distribution on the space of the possible models. A simple Markov chain Monte Carlo algorithm to do this is illustrated in the following paper:

`https://hal.inria.fr/docs/00/10/85/49/PDF/D405.PDF`

Before working on the following points, read and get familiar with the first two sections of the whole paper at a minimum, and optionally the rest of the paper.

1. Write a pseudocode description of the algorithm in Section 2, including input and output. In our version of this algorithm we only want to return one object, the final digraph, not intermediate values along the chain. For the notation, assume the graph is defined over a node set $\mathbf{V}$ (with nodes identified by labels) and an arc set A. The arc set A uniquely identifies a graph so the two can be referred to interchangeably.

2. Now let's write this into a function. Later on we'll care about validating the inputs, here we just assume they're OK. We'll call our function `melancon()`. We'll take as input `nodes` a vector of character strings, which are the labels of the nodes, and `n`, the number of MCMC iterations to perform. To actually implement the algorithm, make use of the following functions from the **bnlearn** package to perform the various steps.

    - `empty.graph()` creates an empty DAG (with no arcs).

- `amat()` creates an adjacency matrix from a DAG such as that returned by `empty.graph()`. (*Note that an adjacency matrix is a square matrix in which a cell* (i, j) *is equal to* `1` *if there is an arc from node* i *to node* j *and* `0` *otherwise (note this is not a symmetric matrix)*)
- `amat()<-` modifies a DAG to contain the the arcs encoded by an adjacency matrix.
- `path()` checks whether there is a path between two nodes. (*Hint: if an arc from nodes* i *to node* j *introduces a cycle, there must be a path going from node* j *to node* i.)

Once you are done return the DAG (not an adjacency matrix).

3. Now try the code out and see if it makes sense for some normal input. Try running it a few times for real, for a simple toy example with 5 nodes the first 5 letters of the alphabet, and and `n=100`. Try sampling a few graphs this way, and plotting the results, using `plot()` on the DAG returned by `melancon()`. Also try some larger toy example, and plot it.

4. Now let's investigate the time and space complexity of `melancon()`. When you perform these calculations, immediately simplify the computational complexity of steps into their class of computational complexity. For instance if something takes 4 steps, or if it is clear it takes a constant number of steps irrespective of input sizes, just write it is $O(1)$ instead of $O(4)$ or similar. Similarly write $O(n)$ if it would take $O(32n)$ time or just generally has linear computational complexity. Make use of the following in thinking about your answer.

   (a) Determining if there is a path between two nodes is done with either a depth-first search (DFS) or a breadth-first search (BFS)

   (b) Wikipedia will tell you the computational complexity of either of DFS or BFS in terms of the number of vertices and edges (here arcs) of a graph

   (c) The paper tells you in the first sentence of Section 3 the average number of arcs for a random acyclic digraph

5. Check the real computational complexity against expectations, by generating DAGs using `melancon()` and recording how long it takes, summarizing your results with two plots, first where you vary the size of `nodes`, and second where you vary the size of `n`

6. Adjacency matrices are one of many possible choices to represent the structure of a graph. Another option are adjacency lists, which are lists in which each element is associated with a node and lists the children of that node. As an example, if we consider a graph with 4 nodes and arcs $\{\{v_1 \to v_3\}, \{v_2 \to v_3\}, \{v_3 \to v_4\}\}$ with the adjacency matrix:

$$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

   and at the same time we can consider the same information represented in an adjacency list:

```
list(v1 = c("v3"), v2 = c("v3"), v3 = c("v4"), v4 = character(0))
```

   Briefly consider the time and space complexity of `melancon()` assuming you implemented it using adjacency lists instead of adjacency matrices. (*Hint: you do not need to actually reimplement the function to do that.*)

7. Now let's work on adding some tests to `melancon`. Let's start with some unit tests. Think about the contract of the function *i.e.* what values it should accept as legal values and throw errors for otherwise. Write one validation function each to ensure `nodes` and `n` take legal values, and throw errors otherwise. Let's suppose we allow `n=0` to allow for no iterations and an empty graph (a graph with no arcs). Let's also suppose we allow the length of `nodes` to be allowed to be 1, even though this graph won't have any arcs. Modify `melancon` as appropriate if need be to accomodate this desired boundary behaviour.

8. Implement a set of unit tests for the to two validation functions you just wrote. Also check briefly that they cause the normal function to break as expected.

9. Implement an acceptance test that checks that `melancon()` actually generates all possible DAGs with uniform probability, for DAGs with 3 nodes. Set a seed to make it reproducible. *Hint: For the purpose of identifying and counting DAGs, use the unique string identifiers generated by modelstring().*