# Statistical Programming, Unassessed Practical

Marco Scutari

November 16, 2018

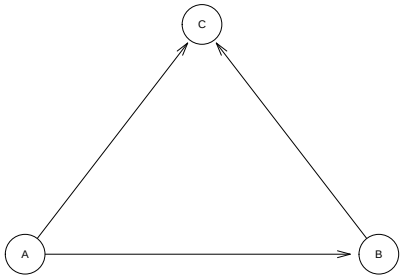
## Exercise 1: Algorithms and Computational Complexity

Many complex statistical models can be represented graphically as *directed acyclic graphs*, in which each node is associated with a random variable and each arc represents a probabilistic dependence relationship. For consistency, we assume that the directed graph 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

$$\mathrm{P}(A, B, C) = \mathrm{P}(A)\,\mathrm{P}(B \mid A)\,\mathrm{P}(C \mid B, A)$$

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 whole paper.

1. Write a pseudocode description of the algorithm in Section 2, including input and output. 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 node set $A$ uniquely identifies a graph so the two can be referred to interchangeably.

2. Write a function definition in R for this algorithm and describe the contract of the function. Write the R code to sanitise the arguments and make they contain legal values according to the function contract.

3. Implement the function following the contract from the previous point and call it `melancon()`. You can use the following functions from the **bnlearn** package to perform the various steps:

   - `empty.graph()` created an empty DAG (with no arcs).
   - `amat()` creates an adjacency matrix from a DAG such as that returned by `empty.graph()`. (*Hint: 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.*)
   - `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* $i$*.*)

4. Compute the time and space complexity of `melancon()` assuming arcs are represented as an adjacency matrix as suggested above. (*Hint: determining if there is a path between two nodes is done with either depth-first search or breadth first search.*)

5. 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 with the adjacency list:

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

   Reconsider 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.*)

### Solution to Exercise 1

1. The basic pseudocode is as follows:

---

**Input:** a set of nodes $\mathbf{V}$ (possibly with associated labels), the number $N$ of graphs to generate.

**Output:** a set $\mathbf{G}$ of $N$ directed acyclic graphs.

(a) Initialise an empty graph with nodes $\mathbf{V}$ and arcs $A_0 = \{\varnothing\}$.

(b) Initialise an empty set of graphs $\mathbf{G}$.

(c) For a large number of iterations $n = 1, \ldots, N$:

i. Sample two random nodes $v_i$ and $v_j \in \mathbf{V}$ with $v_i \neq v_j$.
ii. If $\{v_i \to v_j\} \in A_{n-1}$, then $A_n \leftarrow A_{n-1} \setminus \{v_i \to v_j\}$.
iii. If $\{v_i \to v_j\} \notin A_{n-1}$, check whether the graph is still acyclic after adding $\{v_i \to v_j\}$.
   A. If the graph is still acyclic, $A_n \leftarrow A_{n-1} \cup \{v_i \to v_j\}$.
   B. If the graph is no longer acyclic, nothing is done.
iv. $\mathbf{G} \leftarrow \mathbf{G} \cup A_n$.

---

A better version of this algorithm would include *burn-in* and drop the first $B$ graphs, generating $N + B$ graphs in total and returning the last $N$.

2. The prototype of the function in R is the following.

```
melancon = function(nodes, n) {

}#MELANCON
```

The contract of the function can be defined as follows:

- The argument `nodes` should be a vector of character strings, the labels of the nodes. Labels should be unique, and there should be at least `length(nodes) >= 1` labels.
- The argument `n` should be a numeric or integer vector of length one, the number of DAGs that will be sampled. Its value should be strictly positive; zero can be handled as a special case, but it is fine to classify it as an illegal value as a design decision.
- The function should always return a list of length `n`; each element of the list should hold one DAG.

The R code to sanitise the arguments `nodes` and `n` is as follows.

```
  if (missing(nodes))
    stop("the node labels are missing.")
  if (!is.character(nodes))
    stop("the node labels must be a vector of character strings.")
  if (length(nodes) == 0)
    stop("at least one node label is required.")
  if (any(nodes %in% c("", NA)))
    stop("node label must be non-empty strings.")
  if (anyDuplicated(nodes) > 0)
    stop("node labels must e unique.")

  if (missing(n))
    stop("the number of DAGs to generate is missing.")
  if (!is.numeric(n) && !is.integer(n))
    stop("the number of DAGs must be a numeric value.")
  if (length(n) != 1)
    stop("the number of DAGs must be a single numeric value.")
  if ((n < 1) || is.na(n) || is.infinite(n))
    stop("the number of DAGs must be a positive finite number.")

  n = round(n)
```

3.
```r
library(bnlearn)

melancon = function(nodes, n) {

# add the argument sanitisation code above here at the beginning.

# step (a)
dag = empty.graph(nodes)
adjmat = matrix(0, nrow = length(nodes), ncol = length(nodes),
           dimnames = list(nodes, nodes))
# step (b)
ret = vector(n, mode = "list")

  for (i in seq(n)) {

    # step (c)-i
    candidate.arc = sample(nodes, 2, replace = FALSE)

    # step (c)-ii
    if (adjmat[candidate.arc[1], candidate.arc[2]] == 1) {

      adjmat[candidate.arc[1], candidate.arc[2]] = 0
      amat(dag) = adjmat

    }#THEN
    else {

      # step (c)-iii
      if (!path(dag, from = candidate.arc[2], to = candidate.arc[1])) {

        adjmat[candidate.arc[1], candidate.arc[2]] = 1
        amat(dag) = adjmat

      }#THEN

    }#ELSE

    # step (c)-iv
    ret[[i]] = dag

  }#FOR

  return(ret)

}#MELANCON
```

4. *Time complexity:* For each iteration, adding and removing arcs is $O(1)$ since we just read or write a value in a specific cell of the adjacency matrix. Choosing a pair of nodes at random can also be considered $O(1)$, since we choose two nodes regardless of $|\mathbf{V}|$ or $|A_n|$. Both that depth-first and breadth-first search have time complexity $O(|\mathbf{V}|^2)$ since we have to scan the few adjacency matrix to look for each node's children; and we only perform such

a search if we sample a candidate arc that is not already present. That in turn happens with probability $O(|A_n|/|\mathbf{V}|^2)$. So overall

$$O\left(1 + |\mathbf{V}|^2 \frac{|A_n|}{|\mathbf{V}|^2}\right) \approx O(|A_n|) \approx O(|\mathbf{V}|^2)$$

assuming that $O(|A_n|) \approx O(|\mathbf{V}|^2/4)$ on average as reported in the paper.

Transforming the adjacency matrix into a DAG is necessarily $O(|\mathbf{V}|^2)$ since we need to read each cell in the adjacency matrix to find out which arcs are in the graphs, making each step $O(|\mathbf{V}|^2)$. We do not always perform that transformation, but it is difficult to evaluate how often. A guess is almost always for sparse graphs, since there will typically be no path between $v_j$ and $v_i$ (that is the only case in which we do not change the DAG, and we do not need the transformation). As $|A_n| \to |\mathbf{V}|$ that condition will become easier to meet, so we can say that for a large number of iterations $\approx O(|\mathbf{V}|^2 * 0) = O(1)$ for most iterations.

Therefore, for a small number of iterations time complexity is $O(|\mathbf{V}|^2)$.

*Space complexity:* the adjacency matrix uses $O(|\mathbf{V}|^2)$ space, and that is the most wasteful way of representing a graph. So space complexity cannot exceed $O(|\mathbf{V}|^2)$.

5. *Time complexity:* Breadth-first search, depth-first search and transformation to DAG are all $O(|\mathbf{V}| + |A_n|)$ for adjacency lists. However, this still results in quadratic time complexity:

$$O\left(1 + (|\mathbf{V}| + |A_n|)\frac{|A_n|}{|\mathbf{V}|^2}\right) \approx O\left(1 + (|\mathbf{V}| + |\mathbf{V}|^2)\frac{|\mathbf{V}|^2}{|\mathbf{V}|^2}\right) \approx O(|\mathbf{V}|^2)$$

again assuming that $O(|A_n|) \approx O(|\mathbf{V}|^2/4)$.

*Space complexity:* space complexity of an adjacency list is $O(|\mathbf{V}| + |A_n|)$, that is, the largest between the number of nodes and the number of arcs. On average, this means $O(|\mathbf{V}|^2)$ following the same reasoning as for time complexity.

## Exercise 2: Unit and Systems Testing

Following up from the previous exercise.

1. Modify the `melancon()` function to sample from the uniform distribution over the space of the DAGs that have at most three parents for each node. Pass that as an argument `maxp` to `melancon()`. (*Hint:* **bnlearn** *implements both* `parents()` *and* `in.degree()`.)

2. Describe how the contract of the function is extended to include `maxp`.

3. Implement a set of unit tests for the three arguments of `melancon()`.

4. Implement a system test that checks that `melancon()` actually generates all possible DAGs with uniform probability, and showcase it for DAGs with 3 nodes. (*For the purpose of identifying and counting DAGs, use the unique string identifiers generated by* `modelstring()`.)

### Solution to Exercise 2

1.
```
check.nodes = function(nodes) {

  if (missing(nodes))
    stop("the node labels are missing.")
  if (!is.character(nodes))
    stop("the node labels must be a vector of character strings.")
  if (length(nodes) == 0)
    stop("at least one node label is required.")
  if (any(nodes %in% c("", NA)))
    stop("node label must be non-empty strings.")
  if (anyDuplicated(nodes) > 0)
    stop("node labels must e unique.")

}#CHECK.NODES
```

```
check.n = function(n) {

  if (missing(n))
    stop("the number of DAGs to generate is missing.")
  if (!is.numeric(n) && !is.integer(n))
    stop("the number of DAGs must be a numeric value.")
  if (length(n) != 1)
    stop("the number of DAGs must be a single numeric value.")
  if ((n < 1) || is.na(n) || is.infinite(n))
    stop("the number of DAGs must be a positive finite number.")

  return(round(n))

}#CHECK.N
```

```
check.maxp = function(maxp, nodes) {

  if (missing(maxp))
    stop("the maximum number of parents is missing.")
  if (!is.numeric(maxp) && !is.integer(maxp))
    stop("the maximum number of parents must be a numeric value.")
  if (length(maxp) != 1)
    stop("the maximum number of parents must be a single numeric value.")
  if ((maxp < 0) || (maxp > length(nodes) - 1) || is.na(maxp) ||
       is.infinite(maxp))
    stop("the maximum number of parents must be a positive finite
      number in [0, ", length(nodes) - 1, "].")

  return(round(maxp))

}#CHECK.MAXP
```

```
melancon = function(nodes, n, maxp) {

n = check.n(n)
maxp = check.maxp(maxp, nodes)

# step (a)
dag = empty.graph(nodes)
adjmat = matrix(0, nrow = length(nodes), ncol = length(nodes),
           dimnames = list(nodes, nodes))
# step (b)
ret = vector(n, mode = "list")

  for (i in seq(n)) {

    # step (c)-i
    candidate.arc = sample(nodes, 2, replace = FALSE)

    # step (c)-ii
    if (adjmat[candidate.arc[1], candidate.arc[2]] == 1) {

      adjmat[candidate.arc[1], candidate.arc[2]] = 0
      amat(dag) = adjmat

    }#THEN
    else {

      # step (c)-iii
      if (!path(dag, from = candidate.arc[2], to = candidate.arc[1]) &&
          (length(parents(dag, candidate.arc[2])) < maxp)) {

        adjmat[candidate.arc[1], candidate.arc[2]] = 1
        amat(dag) = adjmat

      }#THEN

    }#ELSE

    # step (c)-iv
    ret[[i]] = dag

  }#FOR

  return(ret)

}#MELANCON
```

2. The contract for this new `melancon()` function is the same as the previous one, with the following addition:

    - The argument `maxp` should be a numeric or integer vector of length one. Its value should be strictly positive. Zero is a trivial case that can be handled as a special case or declared illegal as a design decision. (If every node has zero parents, all the generated

DAGs will be empty.) Its value should be also smaller than `nodes` since a node can have at most `n - 1` parents.

3. A good set of unit test would test a choice set of boundary values, legal values and illegal values for each argument, while setting the other arguments to legal values. (A more extensive set of unit tests would set all combinations of choice boundary, legal and illegal for the arguments; that can be done with a little work extending the solution below.)

   A reasonable choice of values is:

   - `nodes`: `character(0)` (illegal), `NA` (illegal), `c(LETTERS, "")` (illegal), `LETTERS` (legal), `"A"` (boundary), missing value (illegal).
   - `n`: `-1` (illegal), `0` (illegal), `1` (boundary), `3` (legal), `Inf` (illegal), `NA` (illegal), missing value (illegal).
   - `maxp`: `-1` (illegal), `0` (boundary), `1` (legal), `3` (legal), `length(nodes) - 1` (boundary), `length(nodes)` (illegal, )`Inf` (illegal), `NA` (illegal), missing value (illegal).

   Then a test for an illegal value should fail if the function does not return an error.

```
catch = try({ melancon(LETTERS, n = -1, maxp = 1) })

## Error in check.n(n) :
##   the number of DAGs must be a positive finite number.

if (!is(catch, "try-error"))
  stop("this test should have failed (negative 'n')")
```

   All tests for combinations of boundary and legal values should fail if the function returns an error or if the return value is not as expected. For instance:

```
catch = try({ melancon(LETTERS, n = 3, maxp = 3) })
if (is(catch, "try-error"))
  stop("this test should not have failed (negative 'n')")
stopifnot(is(catch, "list"))
stopifnot(all(sapply(catch, is, "bn")))
stopifnot(all(sapply(catch, acyclic)))
for (i in seq_along(catch))
  stopifnot(all(sapply(nodes(catch[[i]]), in.degree, x = catch[[i]])) < 3)
```

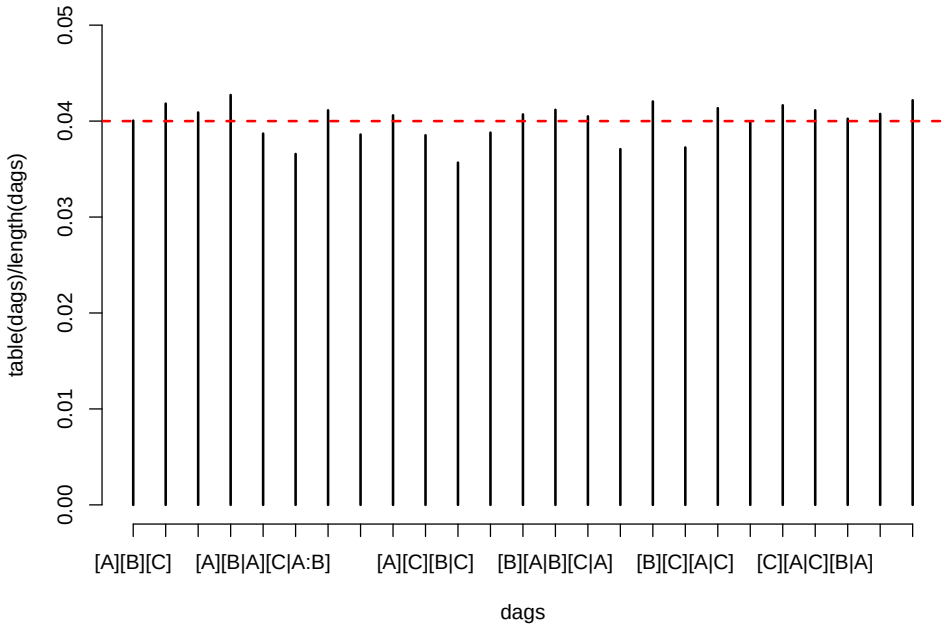
4. We can simulate a large number of DAGs, drop the first few to let the Markov chain Monte Carlo converge to its stationary distributions. Then we plot the relative frequencies of different DAGs, which we identify by their model strings.

```
dags = melancon(LETTERS[1:3], n = 5 * 10^4, maxp = 2)
dags = dags[-(1:10^4)]
dags = sapply(dags, modelstring)
length(unique(dags))

## [1] 25
```

After checking that all 25 possible DAGs with 3 nodes are generated, we can plot their frequency and verify it really is approximately uniform (up to simulation noise).

```
plot(table(dags)/length(dags), ylim = c(0, 0.05))
abline(h = 0.04, col = 2, lwd = 2, lty = 2)
```




A formal test can be either the naive

```
stopifnot(sum(table(dags)/length(dags) - 1/25) < 10^(-8))
```

or a distribution test like Kolmogorov-Smirnov.