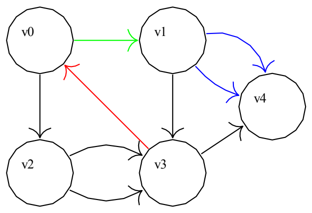
LDD2-Project: Session 3

Aims of this session: To create random matrices of specified forms; to translate between graphs and adjacency matrices.

Remember: Comments and *docs* are expected for (more or less) all the functions and methods in the project.

One possible representation of a (directed, multi) graph $G = (V, E)$ is its adjacency matrix A_G . If $V = \{v_0 , \dots v_{n-1}\}$, we represent an edge (v_i , v_j) of G by a 1 in the i-th row of the j-th column of the matrix A_G. We denote this element by Aij. In a multigraph, if there are k edges from v_i to v_j , we represent it by a k in the matrix, $Aij = k$.

Example: $G =$ ,

$$A_G = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 2 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

(The colours are simply to indicate the nodes corresponding to each row/columns in the matrix.)

This representation is not the most efficient: we need a space of size O(n2), whereas the representation of the previous sessions uses a space of size O(n × Δ) with Δ the maximum degree of the graph. On the other hand, it has many good mathematical properties. In what follows, we'll learn to translate from one representation to the other.

A natural representation of matrices in programming (if you don't use third-party libraries) uses lists of lists. In our case, int list list. For example, the matrix A_G above can be represented in Python by

```
A = [  [0, 1, 1, 0, 0],
       [0, 0, 0, 1, 2],
       [0, 0, 0, 2, 0],
       [1, 0, 0, 0, 1],
       [0, 0, 0, 0, 0] ]
```

You can then access the i'th row using A[i] and the element with index (i,j) using A[i][j]. In the rest of this session, we'll understand a matrix as such a list (of lists) of integers.

Let's start slowly by generating a list of random numbers, which we'll use to randomly define the inputs and outputs of our open graphs.

Exercise 1:
Define a function random_int_list(n, bound) which generates a list of size n containing (random) integers between 0 and bound. We can use the randrange or random functions in the random library (knowing that int(x) rounds the float x down to the nearest integer).

We will then define a few functions for generating random (square) matrices of integers. Each exercise up to the 6th asks you to create a new function which generates a matrix of the required form. You can choose to put them together into a single function, with additional parameters for which you give a default value. For example, at the end we might have something like:

```
def random_matrix(n, bound, null_diag=False,
                  symmetric=False, oriented=False, dag=False):
  <function body>
```

Exercise 2:
Define a function random_int_matrix(n, bound) that generates an n × n matrix with integer elements drawn randomly between 0 and bound.

The matrices defined so far can have non-zero elements on the diagonal. Seen as an adjacency matrix, this would correspond to a graph with edges that "loop" from one node to itself. Although this notion of graph is perfectly valid, it is not useful for our purposes.

Exercise 3:
Add a null_diag parameter to the preceding function, which specifies whether we want the diagonal of the matrix to be zero, and ensure that it takes a default value. For example:

```
def random_int_matrix(n, bound, null_diag=True):
    # to do
```

The adjacency matrix of an undirected graph must be symmetric (this means that $A_{ij} = A_{ji}$).

Exercise 4:
Define a function random_symetric_int_matrix(n, bound, null_diag=True) which returns a symmetric matrix (still with the option of making the diagonal zero).

We may want to impose the following constraint on a graph: if (v_i, v_j) is in E then (v_j, v_i) is not in E (these graphs are sometimes called oriented graphs). This implies that the edges between each pair of nodes can only go in one direction.

Exercise 5:
Define a function random_oriented_int_matrix(n, bound, null_diag=True) that does this.

Finally, the matrix can be made to be the adjacency matrix of an acyclic directed graph (DAG).

Exercise 6:
Apart of the conditions implemented in exercises 3 and 5, what is the missing condition to define the adjacency matrix of a DAG? Hint: have a look at the graph of the example. Can you remove a single edge so that this graph becomes acyclic? What happens to its corresponding adjacency matrix?

Define a function random_dag_int_matrix(n, bound, null_diag=True) that generates random DAGs with integer entries.

We can now proceed to convert an adjacency matrix into a graph, and conversely to extract the adjacency matrix from a given graph.

Exercise 7:
Define a function graph_from_adjacency_matrix which returns a multigraph from a matrix. Leave the input and output attributes of the graph empty.

We can now use everything we've done so far to generate random graphs, with a certain amount of room for manoeuvre.

Exercise 8 (tests required)**:**
Define a method for graphs,

```
@classmethod
def random(cls, n, bound, inputs=0, outputs=0,
                    loop_free=False, DAG=False,
                    oriented=False, undirected=False):
  <function body>
```

which generates a random graph according to the constraints given by the user (and manages conflicting options).

Since many of these options are exclusive, we can instead give a single argument in the form of a string of characters:

```
@classmethod
def random(n, bound, inputs=0, outputs=0, form="free"):
  '''
  Doc
  Be sure to specify the options
  available for form! '
  ''
  if form=="free":
  elif form=="DAG":
  elif form=="oriented":
  elif form=="loop-free":
  elif form=="undirected":
  elif form=="loop-free undirected":
    ...
```

Define the method such that the inputs and outputs of the graph are selected randomly. Here, inputs and outputs represent only the desired number of inputs and outputs.

Let's now try to do the opposite of the previous exercise: to extract an adjacency matrix from a given graph.

To simplify the management of indices, define the following method:

Exercise 9:
Define a method which, when applied to a graph with n nodes, returns a dictionary, associating, to each node id, a unique integer $0 \leq i < n$.

Exercise 10:
Define an adjacency_matrix(self) method which gives an adjacency matrix for the graph (here we ignore inputs and outputs). Pay attention to the consistency of the indices.

Exercise 11 (Bonus)**:**
The function random.randint returns an integer sampled from a uniform probability distribution. This means, for example, that in random_int_matrix(n, 3) there is as much chance of having 0 nodes between each pair of nodes as there is of having 2 or even 3. Modify the function random_int_matrix so that it accepts an optional supplementary argument, which is a float generator function between 0 and 1, given by the user, specifying a possibly non-uniform probability distribution. For example, you could use :

```
random_int_matrix(10, 3, number_generator=(lambda :
                              random.betavariate(1,5)))
```

where betavariate is a function that generates numbers according to a beta distribution, provided by the random library. Other distributions are available in the random library. Adapt the other matrix-generating functions, matrices, as well as the random method for graphs, so that they also use this additional argument.