TI301 – Algorithms and Data Structures 2

Computer Science and Mathematics Project

This project designed jointly with the Mathematics Department

Study of Markov Graphs - PART 1

Markov graphs are representations of discrete-time Markov chains, a very useful tool in probability, with many applications. The aim of this project is to carry out automatic processing on these graphs in order to extract some of their characteristics and visualise them.

In this project, only the theoretical notions essential for the data structure part will be presented.

What is a Markov graph?

A Markov graph is a graph of states (usually simply numbered), with probabilities of transition from one state to another.

A Markov graph is composed of:

- A set of vertices
  - Each vertex represents a state;
  - Each vertex will be identified by a number.
- A set of edges
  - Each edge links two vertices;
  - Each edge has a weight, which is a **strictly positive real value less than or equal to 1**. This value represents the probability of moving from one vertex to another (from one state to another);
  - An edge can link a vertex to itself.

This type of graph must also respect the following constraint:

The sum of the 'outgoing' probabilities of a vertex (of all the edges starting from that vertex) must be equal to 1.

Representations

Classically, these graphs are represented in two ways.

Matrices

By a probability matrix M, where each value M_{ij} indicates the probability of moving from state i to state j. A line i corresponds to the 'outgoing' probabilities of state (or vertex) i.

Example

$$M = \begin{pmatrix} 0.95 & 0.04 & 0.01 & 0 \\ 0 & 0.9 & 0.05 & 0.05 \\ 0 & 0 & 0.8 & 0.2 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Line 1 indicates, for state 1, that:

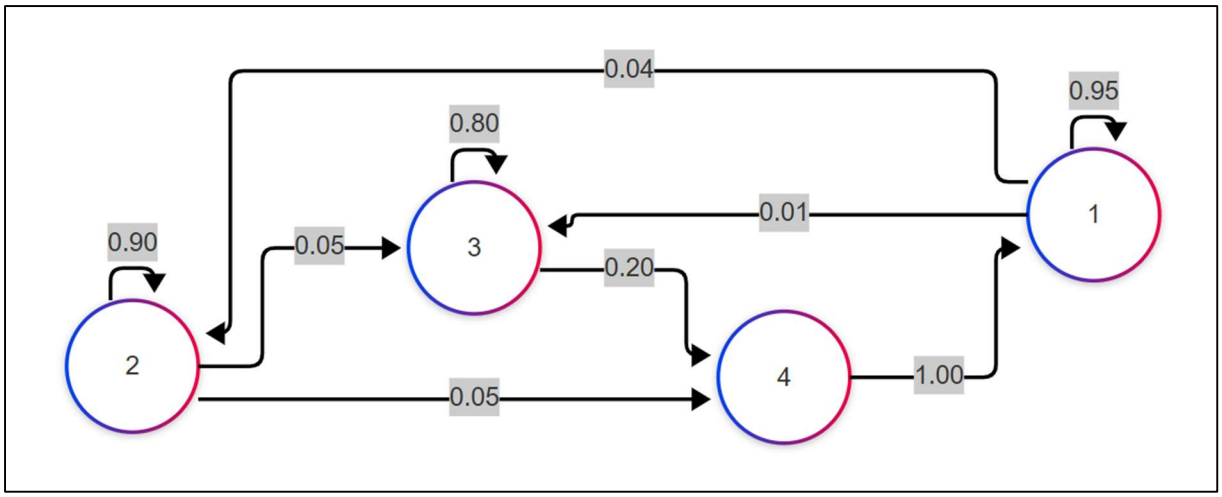
the probability of remaining in state 1 is 0.95
the probability of moving to state 2 is 0.04
the probability of moving to state 3 is 0.01
the probability of moving to state 4 is 0

We can also see that the sum of all the coefficients in this line is equal to 1.

Drawing the graph

By drawing the corresponding directed graph, where these same probabilities are written on the edges.

Example of the graph associated with the matrix M above



It's the type of design you're going to produce!

Specifications

Step 1 - Create a graph, check that it is a Markov graph, draw the graph

Data files

To create graphs, you'll need **data files**, so you don't have to make repetitive input with **`scanf()`**.

These files will have the following format:

Line 1 **: number of vertices**

Next lines: **start_vertex End_vertex probability**

Example: again, for the same graph with 4 vertices, the **'example1.txt'** file is as follows:

```
4
1 1 0.95
1 2 0.04
1 3 0.01
2 2 0.9
2 3 0.05
2 4 0.05
3 3 0.8
3 4 0.2
4 1 1
```

It contains all the non-zero values of the matrix M.

Data structures to be used to store the graph

To store a graph, we could use a 2D array (a matrix) as for the matrix representation - however these matrices for Markov graphs contain a majority of 0s, which are not useful for us at the moment.

- The data structure to be used is therefore an **adjacency list**:

For each vertex, we store, in a (simple) chained list, the edges coming from this vertex, with: the arrival vertex (an **int**), and the associated probability (a **float**)

All these lists (1 per vertex) are stored in an **array**.

Example of the adjacency list for the example graph

```
List for vertex 1:[head @] -> (3, 0.01) @-> (2, 0.04) @-> (1, 0.95)
List for vertex 2:[head @] -> (4, 0.05) @-> (3, 0.05) @-> (2, 0.90)
List for vertex 3:[head @] -> (4, 0.20) @-> (3, 0.80)
List for vertex 4:[head @] -> (1, 1.00)
```

structures to be created

- - a "cell" type structure with: arrival vertex, probability, and pointer to the next "cell". Each "cell" represents an edge.
- - a "list" type structure with a "head" field. Each list stores all the edges 'exiting' from a vertex
- - an "adjacency list" structure: a dynamic "list" array with its size (its size being the number of vertices)

functions to be created

- a function to create a "cell" ;
- a function to create an empty "list" ;
- a function to add a cell to a list ;
- a function for displaying a list ;
- a function to create an 'empty' adjacency list from a given size - an array of lists is created, each initialised with the empty list;
- a function for displaying an adjacency list.

Creating a graph

To create a graph, use one of the text files provided as an example.

Here is a code base to help you create such a list from a text file:

```
adjacency_list readGraph(const char *filename) {
    FILE *file = fopen(filename, "rt"); // read-only, text
    int nbvert, start, end;
    float proba;
    declare the variable for the adjacency list
    if (file == NULL)
    {
        perror("Could not open file for reading");
        exit(EXIT_FAILURE);
    }

    // first line contains number of vertices
    if (fscanf(file, "%d", &nbvert) != 1)
    {
        perror("Could not read number of vertices");
        exit(EXIT_FAILURE);
    }

    Initialise an empty adjacency list using the number of vertices

    while (fscanf(file, "%d %d %f", &start, &end, &proba) == 3)
    {
        // we obtain, for each line of the file, the values
        // start, end and proba
       Add the edge that runs from 'start' to 'end' with the
       probability 'proba' to the adjacency list
    }
```

```
    fclose(file);
    return the completed adjacency list;
}
```

Validation of step 1

You can load a text file;

You can display the adjacency list.

Step 2 - Check the graph

You need to check that the sum of all the 'outgoing' probabilities of a vertex is equal to 1.

Since we're talking about **float** values, there will sometimes be rounding: you'll need to check that the sum of the probabilities is between 0.99 and 1.

Write a function that checks whether a graph is a Markov graph, and displays the messages:
`The graph is a Markov graph`
or

`The graph is not a Markov graph`
with the summits in question, for example

`the sum of the probabilities of vertex 3 is 0.97`

Validation of step 2

Your program is displaying messages correctly - check the various data files (some have been produced by ChatGPT and contain errors).

Step 3 - Drawing the graph

To draw the graph, we're going to use the free online tool Mermaid:
https://www.mermaidchart.com/

This tool lets us generate a drawing from a text file in a format similar to our data files.

Here's the file to create, again for the 4-vertex graph example.

```
---
config:
   layout: elk
   theme: neo
   look: neo
---

flowchart LR
A((1))
B((2))
C((3))
D((4))

A -->|0.01|C
A -->|0.04|B
A -->|0.95|A
B -->|0.05|D
B -->|0.05|C
B -->|0.90|B
C -->|0.20|D
C -->|0.80|C
D -->|1.00|A
```

It includes:

- configuration directives (to be retained)
- Vertex names (A, B, C, D), followed by the numbers to be displayed
- Edges, with their probabilities.

You have a **char *getId(int num);** function which will provide you with a string of characters associated with an integer: "A" for 1, "B" for 2, ..., "Z" for 26, "AA" for 27, "AB" for 28...

- ➢ Write a function that takes an adjacency list (representing a graph) as a parameter, and produces the text file in the expected format.

Then copy/paste the contents of your file into the Mermaid code editor.

Validation of step 3

From the **example_valid_step3.txt** file, you should obtain the following display (after a few very simple manipulations on mermaid)