TI301 – Algorithms and Data Structures 2

Computer Science and Mathematics Project

This project is written in collaboration with the Mathematics Department

Study of Markov Graphs – PART 2

We will now look at the properties of Markov graphs.

To obtain all of these properties, two steps are essential :

Step 1 – Grouping vertices into classes

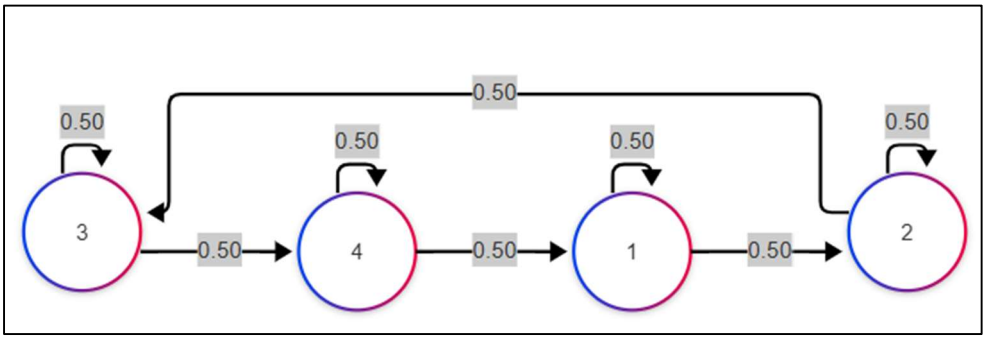
In these graphs, we will group certain vertices together according to the following criterion:

> Vertices must be grouped **into a class** if they all communicate with each other**, i.e. from one of these vertices, we can access all the vertices in this class (including itself).**

These groupings of vertices are called **strongly connected components** or **classes.**

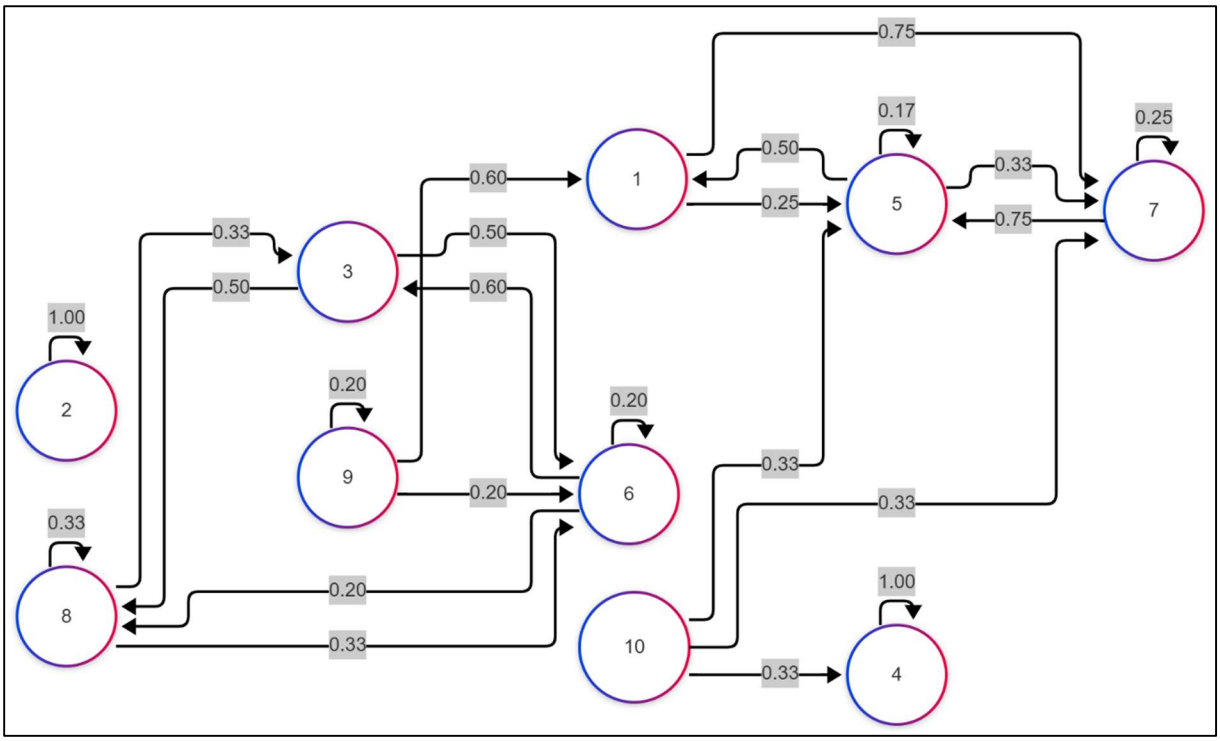
Some examples

1) Graph with 4 vertices, its matrix is $M = \begin{pmatrix} 0.5 & 0.5 & 0 & 0 \\ 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0.5 & 0.5 \\ 0.5 & 0 & 0 & 0.5 \end{pmatrix}$



Here, all the vertices communicate with each other: starting from one vertex, we can access all the others and return to it. We must group the 4 vertices into a **class** that we will call {1,2,3,4}

2) Graph with 10 vertices, used to validate step 3 of part 1



- Vertex **2** only communicates with itself: it forms a **class** on its own
- Vertex **4** only communicates with itself: it forms a **class** on its own
- Vertex **10** also forms a **class** on its own: we can move from **10** to other vertices, but we cannot return to it.
- Vertex **9** is in the same situation: we can start from it but cannot return to it
- Vertices **1**, **5** and **7** form a class: starting from **1**, **5** or **7**, you can always return to **1**, **5** or **7**
- Vertices **3**, **6** and **8** form a class: starting from **3**, **6** or **8**, you can always return to **3**, **6** or **8**.

For this graph, the set of classes is therefore:$\{1,5,7\}, \{2\}, \{3,6,8\}, \{4\}, \{9\}$ and$\{10\}$.

Class properties

- ✓ Every vertex of the graph belongs to one and only one class;
- ✓ Every vertex of the graph must belong to a class;
- ✓ Every class has at least one vertex;
- ✓ A graph contains at least one class;
- ✓ The set of classes in a graph is called a **partition**.

Determining these classes

There are many algorithms for determining these classes, each with its own design challenges (is it easy to design and understand?) and complexity (is it efficient?). The algorithms that are easiest to understand and design generally have poor complexity, at least in $O(\mathrm{N}^3)$ (where N is the number of vertices).

We will use a very efficient algorithm to partition the graph into classes: **Tarjan's algorithm** (see https://fr.wikipedia.org/wiki/Algorithme_de_Tarjan). This is the version of this algorithm that you will need to implement.

As this algorithm is quite complex (in particular, its presentation on the Wikipedia page requires a function within a function, which is not easily achievable with what we know how to do at the moment), it will be broken down into steps.

Step 1 for Tarjan's algorithm – data structures

1.1) This algorithm uses information associated with each vertex:

- ✓ An **identifier (`integer`)**: this is the node number in the graph, as seen in Part 1
- ✓ A **number (`integer`)**: to perform temporary numbering to determine the classes – this number is not the number of the vertex in the graph!
- ✓ An **accessible number** (`integer`): to group vertices that communicate with each other
- ✓ A **Boolean indicator** (an `integer` with a value of 0 or 1) indicating whether the vertex is in the processing stack or not (the algorithm uses a stack).

➢ **You therefore need to create a corresponding data structure.** To continue with the example, let's assume that this type is called `t_tarjan_vertex` (this is just an example, you can choose any name you like).

1.2) This algorithm needs to know the state of all vertices when it runs.

➢ **You therefore need to create an array** storing all the `t_tarjan_vertex` for a graph.

1.3) This algorithm uses classes to store the vertices as it goes along. A class stores `t_tarjan_vertex` 'vertices' and has a name: 'C1', 'C2', etc.

➢ **You therefore need to create a corresponding data structure** that stores a name (the name of the class) and 'a certain number of' `t_tarjan_vertex` vertices. To continue with the example, let's assume that this type is called `t_class`.

It is up to you to choose how to store this, bearing in mind that we do not know in advance how many vertices a class will contain.

1.4) This algorithm provides the partition of the graph into classes. A partition is a set of classes.

- **You must therefore create a corresponding data structure** that stores a certain number of classes `t_class`. (It is up to you to decide how to store this data.)

  It is up to you to choose how to store this information, bearing in mind that we do not know in advance how many classes a partition will contain.

Step 2 for Tarjan's algorithm – utility elements

It is strongly recommended that you implement the following elements:

2.1) A function that is provided with a graph (an adjacency list) and returns an array of `t_tarjan_vertex` (one for each vertex in the adjacency list), initialised with:

- ✓ **identifier (`integer`)** : vertex number in the graph (adjacency list)
- ✓ **number (`integer`)**: -1
- ✓ **accessible number** (`integer`): -1
- ✓ A **Boolean indicator** (an `integer` with a value of 0 or 1): 0

2.2) A stack in which to store `int` or `t_tarjan_vertex`

Step 3 for Tarjan's algorithm – breaking down into functions

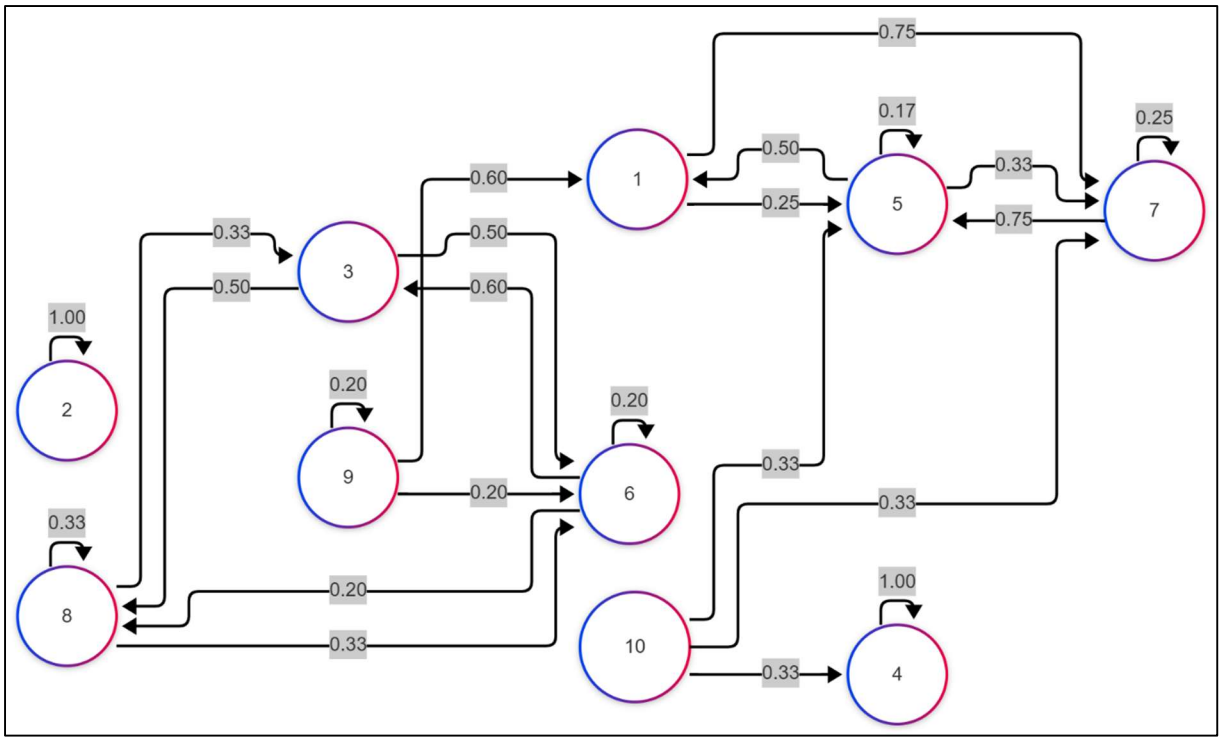
3.1) Write a function that performs the sub-function named '**parcours**' from the Wikipedia page. Please note that this function will require 5 or 6 parameters.

3.2) Write a **'tarjan'** function that initialises an empty **partition**, the `t_tarjan_vertex`, an empty **stack**, and initiates the traversals.

Validation of step 1

Here is the expected output for the result of Tarjan's algorithm.

With the graph:



The programme output should be: (only the values are important, not the display formats)

```
Component C1: {1,7,5}
Component C2: {2}
Component C3: {3,8,6}
Component C4: {4}
Component C5: {9}
Component C6: {10}
```

The order in which the components are displayed is irrelevant

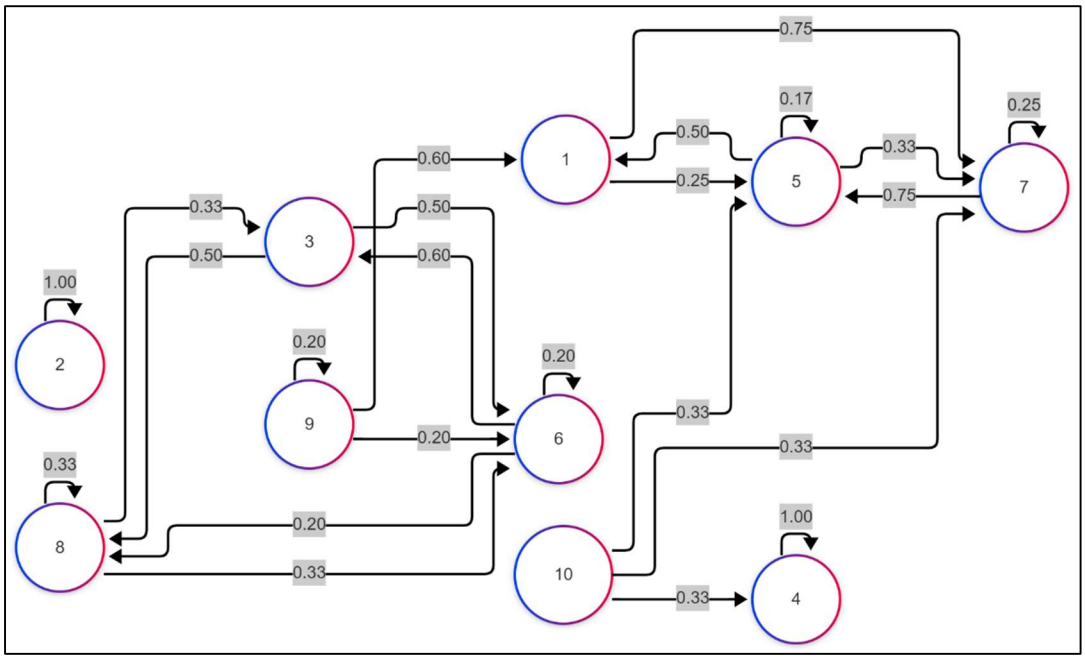
Step 2 – Hasse diagram

We now seek to obtain the Hasse diagram associated with **the classes** obtained in step 1.

This diagram shows the connections between classes. Certain vertices in one class can lead to vertices in other classes (but there is no possible 'return').

This Hasse diagram will allow us to obtain all the properties of the vertices, classes, and Markov graph.

Illustration, still using the same graph as an example.



The classes are:

```
Component C1: {1,7,5}
Component C2: {2}
Component C3: {3,8,6}
Component C4: {4}
Component C5: {9}
Component C6: {10}
```

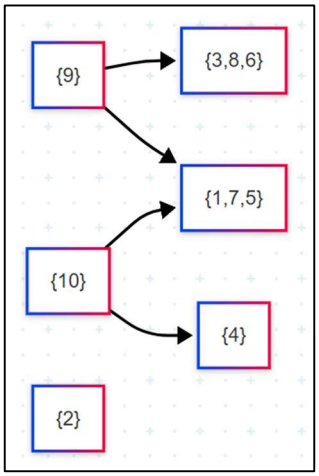
From vertex **10**, we can go to vertex **4**: we therefore have a link from **C6** to **C4**, which we will represent as **C6->C4**

From vertex **10**, we can go to vertex **7**: we therefore have a link from **C6** to **C1**: **C6->C1**

From vertex **10**, we can go to vertex **5**: we therefore have a link from **C6** to **C1** (already identified)

From vertex **1**, we can go to vertices **5** and **7**, which are in the same class **C1**: no link between classes

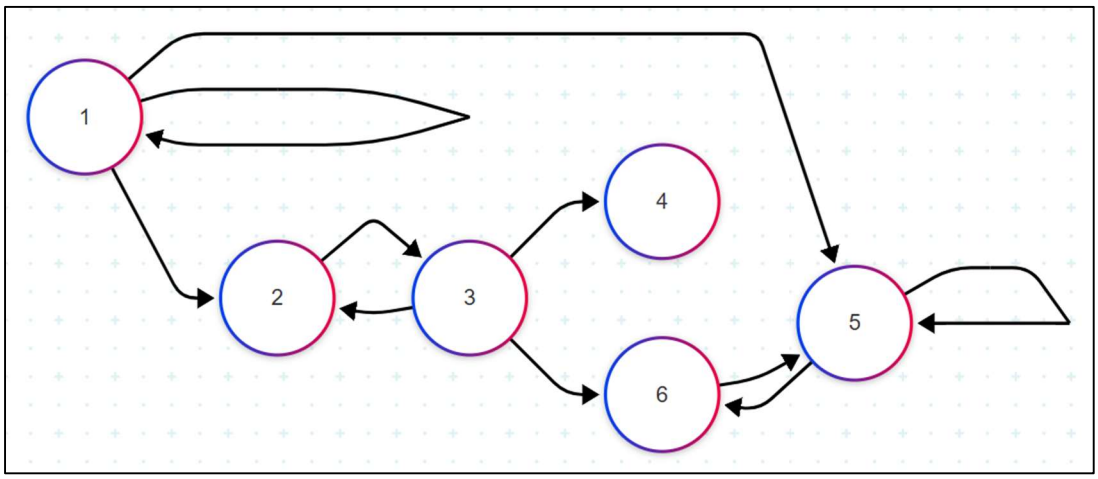
Following the same principle for all vertices, we finally obtain (final visual with Mermaid)



[OPTIONAL] Final definition

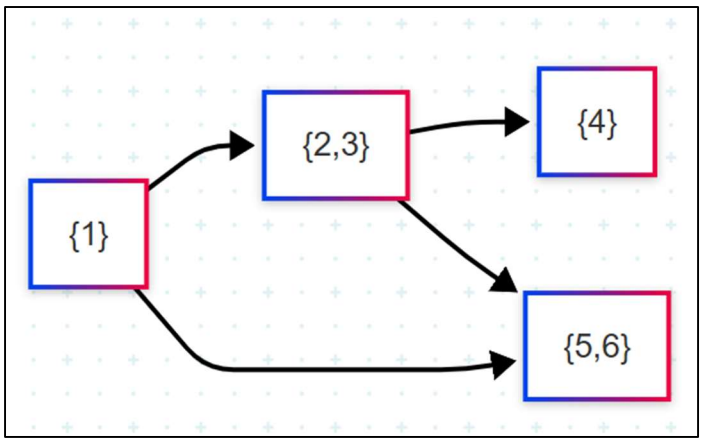
There should be no redundancy in this diagram.

For example, for the following graph:



The classes are {1}, {2,3}, {4}, {5,6}

The links between classes produce the following diagram:



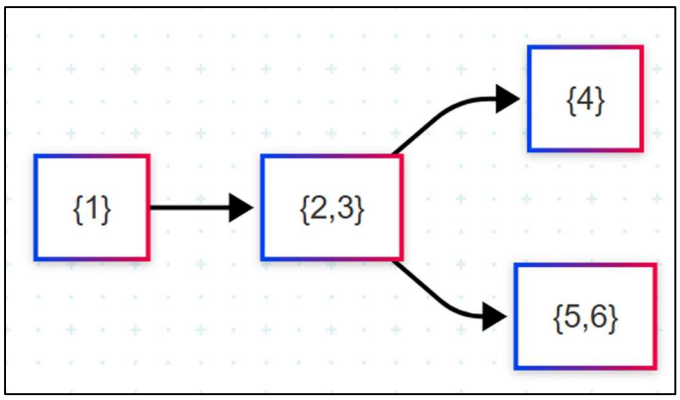
There is redundancy, because

the link from **{1}** to **{5,6}**

is redundant with

the link from **{1}** to **{2,3}** + the link from **{2,3}** to **{5,6}**.

The Hasse diagram is a diagram without redundancy. For the same graph, we obtain:



How to list the links

To obtain an efficient algorithm, we will proceed as follows:

- ✓ Create a 'link' structure that will store two integers: one integer for the 'start' class and one integer for the 'end' class.
- ✓ Create a structure to store multiple 'links'.
- ✓ Create an array that indicates, for each vertex of the graph, the class to which it belongs.

Note that, for the moment, each class 'knows' the vertices that comprise it, but each vertex does not 'know' the class to which it belongs.

Here is the algorithm to implement

```
For each vertex i in the graph
     Ci = class to which i belongs // with the table
     For all vertices j in the adjacency list of vertex i
           Cj = class to which j belongs // with the table
           If Ci is different from Cj // edge between classes
               If the link (Ci,Cj) does not exist
                   Add the link (Ci,Cj) to the structure that stores
the links
```

[OPTIONAL] How to remove redundancies

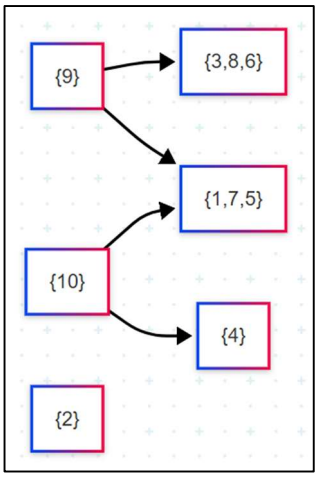
The function that performs this operation is already programmed: it can be found in the **`hasse.h`** / **`hasse.c`** module.

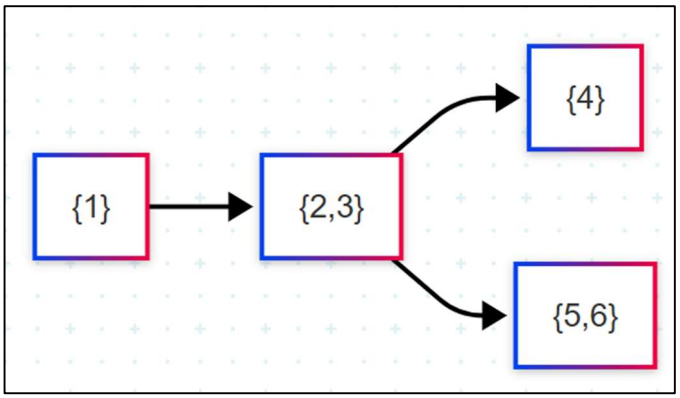
Its prototype is: **`void removeTransitiveLinks(t_link_array *p_link_array);`**

You will probably need to adapt it to your own data structures.

Validation of step 2

Produce a Hasse diagram (with or without redundancies) in Mermaid format, for example:





To construct these diagrams, you will need the graph partition (set of classes) and all the links between classes that you have found.

Step 3: graph characteristics

Once the classes and links have been obtained, these characteristics can be obtained almost immediately:

- ✓ A class is said to be **'transitory'** if it is possible to 'exit' that class: there is an 'outgoing arrow' from that class. All states in that class are said to be **'transitory'.**
- ✓ A class is said to be **'persistent'** if you cannot 'exit' this class: there is no 'outgoing' arrow from this class. All states in this class are said to be **'persistent'.**
- ✓ A state is said to be **'absorbing'** if it is in a persistent class and is the only state in that class.
- ✓ A Markov graph is said to be **'irreducible'** if it is composed of only one class.

In the diagram on the right:

- ✓ The class {1} is transient – state 1 is transient;
- ✓ The class {2,3} is transient – states 2 and 3 are transient;
- ✓ The class {4} is persistent – state 4 is persistent – state 4 is absorbing;
- ✓ The class {5,6} is persistent – states 5 and 6 are persistent;
- ✓ The Markov graph is not irreducible.

Validation of step 3

Your programme displays all the characteristics of the graph:

- ✓ Whether the classes are transient or persistent;
- ✓ Whether there are absorbing states;
- ✓ Whether the graph is irreducible or not.