Structure hiérarchique, les graphes

19 avril 2024

Plan

Rendu du DM

Moyenne	Écart type	Q1	Médiane	Q3	Max
17,0	2,11	16,25	17,5	18,5	20

Preuves très mal réussies, exercices de code variables.

Complexités

- Attention à bien expliquer un pire des cas ;
- Il suffit d'un majorant pour les grand O, mais il faut du coup justifier que ce n'est pas un calcul exacte ;
- Le pire et meilleur des cas doivent exister pour toutes les tailles d'entrées ;
- Dans le cas de plusieurs tailles, vous devez pouvoir considérer toutes les combinaisons de taille.

Induction

- Il vous faut vos cas de base, et tous vos cas récursifs ;
- Si vous voulez définir une relation, il faut préciser que les autres cas n'ont pas la relation.

Remarques générales

- Ce qui est rayé ne sera pas lu.

C'était le dernier DM

Il n'y aura probablement plus de DM d'ici à la fin de l'année.

Organisation des colles

- Dans la mesure du possible, je vous encourage à échanger avec d'autres élèves qui passent la même semaine si vous avez un imprévu.
- J'ai demandé aux colleurs si je pouvais vous communiquer leurs adresses e-mail, voici les adresses pour lesquelles j'ai obtenue l'autorisation : marius.goyet@ens-lyon.fr, martin.trucchi@ens-lyon.fr

DS

Le DS de ce vendredi sera plus court que d'habitude, mais je noterai plus sévèrement la rédaction. Si les codes ne correspondent pas à mes attentes (code écrit sur une seule page à chaque fois, code qui n'est pas présenté par une phrase, code mal indenté ou pas clair, ...) je pénaliserai.
De la même manière, les démonstrations qui ne sont pas claires, ou les résultats qui ne sont pas encadrés seront pénalisées.

Cours sur les graphes

Nous commençons cette semaine le cours sur les graphe, ce qui devrait nous occuper pour quelques semaines.
Peu d'illustrations figurent dans le polycopiés, je vous encourage grandement à bien prendre les illustrations en note.
Je n'ai toujours pas mis de place pour les démonstrations ou les exercices.

Découpage du cours sur les graphes

Le chapitre sera découpé en plusieurs sections.

Théorie des graphes

Définition 1 : Graphe orienté

Un graphe orienté G est un couple (S, A) tel que $A \subset \{(u, v) \in S^2 | x \neq y\}$. On suppose par ailleurs S fini.
S est l'ensemble des *sommets*.
A est l'ensemble des *arêtes*, que l'on nomme aussi dans le cas orienté *arcs*.

Définition 1 : Graphe orienté

Un graphe orienté G est un couple (S, A) tel que $A \subset \{(u, v) \in S^2 | x \neq y\}$. On suppose par ailleurs S fini.
S est l'ensemble des *sommets*.
A est l'ensemble des *arêtes*, que l'on nomme aussi dans le cas orienté *arcs*.

Exemple 1 : Quelques utilisations des graphes orientés

- Graphes de relations binaires ;
- Automates à état finis ;
- Graphes de flux de contrôle.

Définition 2 : Degrés entrant et sortant

Soit (S, A) un graphe orienté. Soit $s \in S$ un sommet.
On note $d_-(s)$ le **degré entrant** de s le nombre d'arêtes dont la première composante est s. On note $d_+(s)$ le **degré sortant** de s le nombre d'arêtes dont la deuxième composante est s.
Ainsi :

$$d_+(s) = |\{(x, y) \in A | x = s\}|$$

$$d_-(s) = |\{(x, y) \in A | y = s\}|$$

Définition 2 : Degrés entrant et sortant

Soit (S, A) un graphe orienté. Soit $s \in S$ un sommet.
On note $d_-(s)$ le **degré entrant** de s le nombre d'arêtes dont la première composante est s. On note $d_+(s)$ le **degré sortant** de s le nombre d'arêtes dont la deuxième composante est s.
Ainsi :

$$d_+(s) = |\{(x, y) \in A | x = s\}|$$

$$d_-(s) = |\{(x, y) \in A | y = s\}|$$

Proposition 1 : Relation entre le degré entrant et le degré sortant

La somme des degrés entrant et celle des degrés sortants des nœuds dans un graphe sont égales au nombre d'arêtes.
Ainsi, pour tout graphe $G = (S, A)$:

$$|A| = \sum_{s \in S} d_+(s) = \sum_{s \in S} d_-(s)$$

Définition 3 : Chemin (à partir des arêtes)

Soit $G = (S, A)$ un graphe orienté. Un **chemin** $a_0 a_1 \cdots a_n$ est une suite finie d'arêtes de A telle que pour tout $1 \leqslant k \leqslant n$, en notant $a_k = (s_k, s'_k)$ et $a_{k-1} = (s_{k-1}, s'_{k-1})$, on ait $s'_{k-1} = s_k$, c'est-à-dire que le nœud d'arrivée d'une arête est le nœud de départ de la suivante.
La **longueur** de ce chemin est le nombre d'arêtes traversées, c'est-à-dire $n + 1$.

Définition 4 : Chemin (à partir des nœuds)

Soit $G = (S, A)$ un graphe orienté. Un **chemin** $s_0 s_1 \cdots s_n$ dans G est une suite finie de sommets de S telle que pour tout $1 \leqslant k \leqslant n$, (s_{k-1}, s_k) soit dans A, c'est-à-dire dont tous les couples consécutifs sont des arêtes du graphe.
La **longueur** de ce chemin est le nombre d'arêtes traversées, c'est-à-dire n. On dit qu'il s'agit d'un chemin de s_0 à s_n.

Définition 5 : Cycle dans un graphe orienté

Un **cycle** est un chemin de longueur non nulle tel que le premier nœud est égal au dernier nœud, c'est-à-dire que si il s'écrit $s_0 \cdots s_n$, alors $s_0 = s_n$, et qui passe par des arêtes toutes distinctes.

Définition 5 : Cycle dans un graphe orienté

Un **cycle** est un chemin de longueur non nulle tel que le premier nœud est égal au dernier nœud, c'est-à-dire que si il s'écrit $s_0 \cdots s_n$, alors $s_0 = s_n$, et qui passe par des arêtes toutes distinctes.

Définition 6 : Chemin sans cycle

Un chemin est dit **sans cycle** si et seulement si il ne passe pas deux fois par le même nœud.

Définition 5 : Cycle dans un graphe orienté

Un **cycle** est un chemin de longueur non nulle tel que le premier nœud est égal au dernier nœud, c'est-à-dire que si il s'écrit $s_0 \cdots s_n$, alors $s_0 = s_n$, et qui passe par des arêtes toutes distinctes.

Définition 6 : Chemin sans cycle

Un chemin est dit **sans cycle** si et seulement si il ne passe pas deux fois par le même nœud.

Proposition 2 : Taille d'un chemin sans cycle

Soit $G = (S, A)$ un graphe. La longeur de tous chemin sans cycle est majoré par le nombre de sommets.

Définition 7 : Graphe acyclique

Un graphe est dit **acyclique** s'il ne contient pas de cycle.

Définition 8 : Graphe fortement connexe

Un graphe orienté $G = (S, A)$ est dit **fortement connexe** si pour toute paire de sommets $s \neq s' \in S$, il existe un chemin de s à s'.

Définition 8 : Graphe fortement connexe

Un graphe orienté $G = (S, A)$ est dit **fortement connexe** si pour toute paire de sommets $s \neq s' \in S$, il existe un chemin de s à s'.

Définition 9 : Graphe faiblement connexe

Soit un graphe orienté $G = (S, A)$. Soit $G' = (S, A')$ avec A' défini de la manière suivante :

$$\forall s, s' \in S,\ (s, s') \in A' \Leftrightarrow (s, s') \in A \vee (s', s) \in A$$

G est dit **faiblement connexe** si et seulement si G' est fortement connexe.

Proposition 3 : Forte connexité implique faible connexité

Tout graphe fortement connexe est faiblement connexe.

Exercice 1. Donne un contre-exemple pour la réciproque.

Définition 10 : Sous-graphe

Soit $G = (S, A)$ un graphe orienté. $G' = (S', A')$ est un **sous-graphe** de G si G' est un graphe et que $S' \subset S$ et $A' \subset A$.

Définition 10 : Sous-graphe

Soit $G = (S, A)$ un graphe orienté. $G' = (S', A')$ est un **sous-graphe** de G si G' est un graphe et que $S' \subset S$ et $A' \subset A$.

Définition 11 : Sous-graphe induit

Soit $G = (S, A)$. Soit $S' \subset S$, et soit $A' = (S' \times S') \cap A$. $G' = (S', A')$ est le **sous-graphe de G induit par S'**.

Définition 10 : Sous-graphe

Soit $G = (S, A)$ un graphe orienté. $G' = (S', A')$ est un **sous-graphe** de G si G' est un graphe et que $S' \subset S$ et $A' \subset A$.

Définition 11 : Sous-graphe induit

Soit $G = (S, A)$. Soit $S' \subset S$, et soit $A' = (S' \times S') \cap A$. $G' = (S', A')$ est le **sous-graphe de G induit par S'**.

Exercice 2. Montrer qu'un sous-graphe induit est bien un sous-graphe.

Définition 12 : Composante fortement connexe

Une **composante fortement connexe** est un sous-graphe fortement connexe maximal pour l'inclusion, c'est-à-dire qu'on ne peut pas agrandir en conservant la propriété de forte connexité.

Définition 13 : Composante faiblement connexe

Une **composante faiblement connexe** est un sous-graphe faiblement connexe maximal pour l'inclusion.

Proposition 4 : Décomposition d'un graphe en ses composantes connexes

L'ensemble des composantes fortement connexes d'un graphe forment une partition des nœuds de ce graphe.
L'ensemble des composantes faiblement connexes d'un graphe forment une partition des nœuds de ce graphe.

Exercice 3. Montrer que l'ensemble des composantes faiblement connexes d'un graphe orienté forment une partition des arêtes de ce graphe.

Définition 14 : Paire

Une paire est un ensemble à deux éléments.

Définition 14 : Paire

Une paire est un ensemble à deux éléments.

Définition 15 : Graphe non-orienté

Un graphe non-orienté G est un couple (S, A) tel que A est un ensemble de paires de S.

Définition 14 : Paire

Une paire est un ensemble à deux éléments.

Définition 15 : Graphe non-orienté

Un graphe non-orienté G est un couple (S, A) tel que A est un ensemble de paires de S.

Exemple 2 : Quelques utilisations de graphes non-orientés

- Graphes pour des relations binaires symmétriques ;
- Graphes d'intervalles.

Définition 16 : Degré dans un graphe non-orienté

Le degré d'un nœud est le nombre d'arêtes à laquel il appartient. Ainsi, pour un graphe non-orienté (S, A) et un sommet $s \in S$, le degré de s noté $d(s)$ (ou parfois $\deg(s)$) est égal à :

$$d(s) = \left|\left\{a \in A \middle| s \in a\right\}\right|$$

Définition 16 : Degré dans un graphe non-orienté

Le degré d'un nœud est le nombre d'arêtes à laquel il appartient. Ainsi, pour un graphe non-orienté (S, A) et un sommet $s \in S$, le degré de s noté $d(s)$ (ou parfois $\deg(s)$) est égal à :

$$d(s) = |\{a \in A | s \in a\}|$$

Proposition 5 : Les poignées de main

La somme des degré des sommets d'un graphe est égal à deux fois le nombre d'arêtes.
Ainsi, pour (S, A) un graphe :

$$\sum_{s \in S} d(s) = 2\,|A|$$

Définition 17 : Graphe non-orienté complet

Un graphe non-orienté complet $G = (S, A)$ est un graphe dont les arêtes sont toutes les paires possibles de S :

$$A = \left\{ \left\{ s, s' \right\} | s, s' \in S, s \neq s' \right\}$$

Définition 17 : Graphe non-orienté complet

Un graphe non-orienté complet $G = (S, A)$ est un graphe dont les arêtes sont toutes les paires possibles de S :

$$A = \left\{ \left\{ s, s' \right\} | s, s' \in S, s \neq s' \right\}$$

Proposition 6 : Borne sur le nombre d'arêtes dans un graphe non-orienté

Dans un graphe (S, A), on a la relation suivante :

$$\frac{|S](|S| - 1)}{2} \geqslant |A|$$

Le cas d'égalité étant exactement le cas du graphe complet.

Définition 18 : Chemin

Soit $G = (S, A)$ un graphe non-orienté. Un **chemin** de G, $s_0 s_1 \cdots s_n$ pour $n > 0$, est une suite finie de sommet telle que pour tout $0 \leqslant k \leqslant n-1$, on ait $\{s_k, s_{k+1}\} \in A$.
On note $n-1$ la longueur de ce chemin, et on dit que ce chemin est un chemin est entre s_0 et s_n.

Définition 19 : Graphe connexe

Un graphe $G = (S, A)$ non-orienté est dit **connexe** si pour tout $(s, s') \in S$, il existe un chemin entre s et s'.

Définition 19 : Graphe connexe

Un graphe $G = (S, A)$ non-orienté est dit **connexe** si pour tout $(s, s') \in S$, il existe un chemin entre s et s'.

Définition 20 : Composante connexe dans un graphe non-orienté

Une composante connexe est un sous-arbre connexe maximum pour l'inclusion.

Définition 19 : Graphe connexe

Un graphe $G = (S, A)$ non-orienté est dit **connexe** si pour tout $(s, s') \in S$, il existe un chemin entre s et s'.

Définition 20 : Composante connexe dans un graphe non-orienté

Une composante connexe est un sous-arbre connexe maximum pour l'inclusion.

Proposition 7 : Composantes connexes et partition dans un graphe non-orienté

Les composantes connexes d'un graphe forment une partition de ce graphe pour les sommets et pour les arêtes.

Définition 21 : Graphe pondéré

Un **graphe pondéré** (pour ses arêtes) est un graphe muni d'une fonction $p : A \to \mathbb{R}$.
p est la fonction de poids, et ses valeurs sont les poids.

Définition 22 : Poids d'un graphe

Le poids d'un graphe est égal à la somme des poids de ses éléments.
Ainsi, pour un graphe pondéré par ses arêtes :

$$P(G) = \sum_{a \in A} p(A)$$

Définition 22 : Poids d'un graphe

Le poids d'un graphe est égal à la somme des poids de ses éléments.
Ainsi, pour un graphe pondéré par ses arêtes :

$$P(G) = \sum_{a \in A} p(A)$$

Exercice 4. Montrer que si tous les poids sont strictement positifs, le sous-graphe de poid maximal est exactement le graphe lui-même. Que se passe-t-il s'il y a des poids égaux à 0 ou strictement négatifs ?

Définition 23 : Arbre au sens de la théorie des graphes

Un **arbre** au sens de la théorie des graphes est un graphe non-orienté connexe acyclique.

Définition 24 : Sous-arbre au sens de la théorie des graphes

Un **sous-arbre** d'un arbre au sens de la théorie des graphes est un sous-graphe connexe et acyclique de cet arbre.

Proposition 8 : Nombre de chemins dans un arbre

Soit G un graphe. G est un arbre si et seulement si pour chaque paire de sommet, il existe un unique chemin entre ces sommets.

Définition 25 : Arbres enracinés au sens de la théorie des graphes

Un **arbre enraciné** $T = (S, A, r)$ est un triplet tel que (S, A) est un arbre au sens de la théorie des graphes, et $r \in S$.
On dit que r est la **racine** de l'arbre enraciné.

Définition 25 : Arbres enracinés au sens de la théorie des graphes

Un **arbre enraciné** $T = (S, A, r)$ est un triplet tel que (S, A) est un arbre au sens de la théorie des graphes, et $r \in S$.
On dit que r est la **racine** de l'arbre enraciné.

Proposition 9 : Équivalence des définitions

Il existe une bijection naturelle entre les arbres enracinés au sens de la théorie des graphes et les arbres d'arité arbitraires définis par induction.

Proposition 10 : Nombre d'arêtes dans un arbre

Soit $T = (S, A)$ un arbre non vide. Alors $|S| - 1 = |A|$.

Proposition 10 : Nombre d'arêtes dans un arbre

Soit $T = (S, A)$ un arbre non vide. Alors $|S| - 1 = |A|$.

Proposition 11 : Nombre d'arêtes dans un graphe connexe

Soit $G = (S, A)$ un graphe connexe non vide. Alors on a la relation suivante :

$$|S| - 1 \leqslant |A|$$

Le cas d'égalité correspond exactement aux arbres.

Définition 26 : Isomorphisme de graphes orientés

Soit $G = (S, A)$ et $G' = (S', A')$ deux graphes orientés. Un **isomorphisme de graphes orientés** entre G et G' est une bijection φ de S dans S' telle que (u, v) soit une arête de A si et seulement si $(\varphi(u), \varphi(v))$ est une arête de A'.

Définition 26 : Isomorphisme de graphes orientés

Soit $G = (S, A)$ et $G' = (S', A')$ deux graphes orientés. Un **isomorphisme de graphes orientés** entre G et G' est une bijection φ de S dans S' telle que (u, v) soit une arête de A si et seulement si $(\varphi(u), \varphi(v))$ est une arête de A'.

Définition 27 : Isomorphisme de graphes non-orientés

Soit $G = (S, A)$ et $G' = (S', A')$ deux graphes non-orientés. Un **isomorphisme de graphes non-orientés** entre G et G' est une bijection φ de S dans S' telle que $\{u, v\}$ soit une arête de A si et seulement si $\{\varphi(u), \varphi(v)\}$ est une arête de A'.

Définition 28 : Graphes isomorphes

Deux graphes G et G' sont isomorphes s'il existe un isomorphisme entre les deux.

Définition 28 : Graphes isomorphes

Deux graphes G et G' sont isomorphes s'il existe un isomorphisme entre les deux.

Exercice 5. Combien de graphes non-orientés à 3 sommets existent-ils à isomorphisme près ?

Définition 28 : Graphes isomorphes

Deux graphes G et G' sont isomorphes s'il existe un isomorphisme entre les deux.

Exercice 5. Combien de graphes non-orientés à 3 sommets existent-ils à isomorphisme près ?

Proposition 12 : Unicité du graphe complet à n sommets

Tous les graphes complets à n sommets sont isomorphes deux à deux.

Résultat du DS

Moyenne	Écart type	Q1	Médiane	Q3	Max
8,60	4,36	5,7	8,5	10,8	19

Coefficient des questions :

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
6	4	4	6	8	2	4	4	4	4	4	6	4	6	8

Méthode de notation

- Toutes les questions sont notées sur 1, mais avec un coefficient qui est ajusté à posteriori ;
- Des points retirés en pourcentage de la note totale en fonction des problèmes dans la présentation (pagination absente ou incomplète, code peu ou pas commenté ou présenté, résultats qui ne sont pas mis en valeur, ...).

Quelques remarques

- On ne peut pas montrer un résultat asymptotique directement avec une récurrence / induction ;
- On ne peut pas comparer des chaînes de caractère facilement en C ;
- assert attend un booléen en argument en C, pas un message d'erreur à afficher ;
- Attention aux pointeurs, ça a été très mal traité ;
- C'est un peu mieux pour la mémoire, mais il y a des problèmes de mémoire mal gérés ;
- Il faut utiliser les résultats des questions précédentes au lieu de refaire la preuve à chaque fois ;
- Beaucoup de codes inutilement long.

Représentation des graphes

Définition 29 : Représentation d'un graphe orienté par une matrice

Soit $G = (S, A)$ un graphe orienté avec $S = \{1, ..., n\}$.
La matrice d'adjacence M de G est une matrice de taille $n \times n$ dont les coefficients sont dans $\{0, 1\}$ et sont donnés par :

$$m_{i,j} = \begin{cases} 1 & \text{si } (i, j) \in A \\ 0 & \text{sinon} \end{cases}$$

Définition 29 : Représentation d'un graphe orienté par une matrice

Soit $G = (S, A)$ un graphe orienté avec $S = \{1, ..., n\}$.
La matrice d'adjacence M de G est une matrice de taille $n \times n$ dont les coefficients sont dans $\{0, 1\}$ et sont donnés par :

$$m_{i,j} = \begin{cases} 1 & \text{si } (i, j) \in A \\ 0 & \text{sinon} \end{cases}$$

Définition 30 : Représentation d'un graphe non-orienté par une matrice

Soit $G = (S, A)$ un graphe non-orienté avec $S = \{1, \cdots, n\}$.
La matrice d'adjacence M de G est une matrice de taille $n \times n$ dont les coefficients sont dans $\{0, 1\}$ et sont donnés par :

$$m_{i,j} = \begin{cases} 1 & \text{si } \{i, j\} \in A \\ 0 & \text{sinon} \end{cases}$$

Proposition 13 : Propriété des matrices d'adjacence des graphes non-orientés

Une matrice d'adjacence d'un graphe non-orienté est symmétrique.

Proposition 14 : Caractérisation d'un graphe à isomorphise près par sa matrice d'adjacence

Deux graphes G et G' de matrices d'adjacence respectives M et M' sont isomorphes si et seulement si les deux conditions suivantes sont vérifiées :

1. G et G' ont le même nombre de sommets n.
2. Il existe une permutation σ de $\{1, ..., n\}$ telle que, pour tout $1 \leqslant i, j \leqslant n$, on a $m_{i,j} = m'_{\sigma(i),\sigma(j)}$.

Proposition 15 : Exponentiation de la matrice d'adjacence

Soit G un graphe dont la matrice d'adjacence est M.
Pour tout $k > 0$, M^k représente la matrice dont l'élément d'indice (i, j) correspond au nombre de chemins de longueur exactement k qui vont de i à j.

Définition 31 : Représentation d'un graphe orienté complet pondéré par une matrice d'adjacence

Soit $G = (S, A, p)$ un graphe pondéré orienté complet avec $S = \{1, \cdots, n\}$. La matrice d'adjacence de G est une matrice M de taille $n \times n$ telle que :

$$m_{i,j} = \begin{cases} 0 & \text{si } i = j \\ p((i,j)) & \text{sinon.} \end{cases}$$

Définition 32 : Représentation d'un graphe orienté complet pondéré par une matrice d'adjacence

Soit $G = (S, A, p)$ un graphe pondéré non-orienté complet avec $S = \{1, \cdots, n\}$. La matrice d'adjacence de G est une matrice M de taille $n \times n$ telle que :

$$m_{i,j} = \begin{cases} 0 & \text{si } i = j \\ p(\{i,j\}) & \text{sinon.} \end{cases}$$

Définition 32 : Représentation d'un graphe orienté complet pondéré par une matrice d'adjacence

Soit $G = (S, A, p)$ un graphe pondéré non-orienté complet avec $S = \{1, \cdots, n\}$. La matrice d'adjacence de G est une matrice M de taille $n \times n$ telle que :

$$m_{i,j} = \begin{cases} 0 & \text{si } i = j \\ p(\{i, j\}) & \text{sinon.} \end{cases}$$

Exercice 6. Proposer une matrice d'adjacence pour le graphe des trajets SNCF. Quelle valeur par défaut peut-on utiliser ?

Exemple 3 : Matrice d'adjacence pour des graphes en OCaml

On peut utiliser le type suivant en OCaml pour représenter une matrice d'adjacence en OCaml :

```
type graphe = int array array
```

Dans ce cas, on cherche à ce que `m.(i).(j)` soit égal à 1 si et seulement si $m_{i,j}$ vaut 1

Exemple 4 : Nombre d'arêtes dans un graphe orienté

```
let nb_aretes mat_adj =
    let n = Array.length mat_adj in
    if n = 0 then 0 else
    begin
        let res = ref 0 in
        for i = 0 to n-1 do
            for j = 0 to n-1 do
                res:= !res + mat_adj.(i).(j)
            done
        done ;
        !res
    end
```

Exemple 4 : Nombre d'arêtes dans un graphe orienté

```
let nb_aretes mat_adj =
    let n = Array.length mat_adj in
    if n = 0 then 0 else
    begin
        let res = ref 0 in
        for i = 0 to n-1 do
            for j = 0 to n-1 do
                res:= !res + mat_adj.(i).(j)
            done
        done ;
        !res
    end
```

Exercice 7. Proposer une fonction qui renvoie le nombre d'arêtes dans un graphe non-orienté.

Exemple 5 : Tableau de taille statique en C passé sur la pile

```c
const int taille = 10;

void afficher(int t[taille][taille]){
    for (int i = 0; i<taille; i++){
        for (int j = 0; j<taille;j++){
            printf("%d ", t[i][j]);
        }
        printf("\n");
    }
}
```

Exemple 6 : Tableau de taille statique en C passé par pointeur

```c
const int taille = 10;

void afficher(int (*t)[taille][taille]){
    for (int i = 0; i<taille; i++){
        for (int j = 0; j<taille;j++){
            printf("%d ", (*t)[i][j]);
        }
        printf("\n");
    }
}
```

Exercice 8. Proposer une fonction de prototype
`int nombre_aretes(int (*mat)[ taille ][ taille ])` qui calcule le nombre d'arêtes dans un graphe non-orienté donné par sa matrice d'adjacence.

Exemple 7 : Tableau de tableaux dynamiques en C

```c
int ** creer_matrice(int n, int m){
    int ** mat = (int **) malloc(n * sizeof(int *));
    for (int i = 0; i< n;i++){
        int * ligne = (int *) malloc(m * sizeof(int));
        mat[i] = ligne;
    }
    return mat;
}
int acceder(int * mat, int i, int j){
    return mat[i][j];
}
```

Exemple 8 : Tableau linéarisé en C

```c
int * creer_matrice(int n, int m){
    int * mat = (int*) malloc(n * m * sizeof(
   int));
    return mat;
}
int acceder(int * mat, int i, int j, int m){
    return mat[i * m + j];
}
```

Exercice 9. Proposer une fonction de prototype
`int * lineariser (int ** mat, int n, int m)` qui crée un tableau linéarisé à partir de la matrice `mat`.
La fonction libèrera l'espace alloué à la matrice `mat`.

Proposition 16 : Complexité avec l'implémentation en C

On obtient les mêmes complexités avec l'implémentation en C par matrice d'adjacence.

Définition 33 : Listes d'adjacence d'un graphe orienté

Soit $G = (S, A)$ un graphe orienté avec $S = \{1, \cdots, n\}$.
Les listes d'adjacence de G sont des listes $(l_i)_{1 \leqslant i \leqslant n}$ tels que, pour tout $1 \leqslant i, j \leqslant n$, $j \in l_i$ si et seulement si $(i, j) \in A$.

Définition 33 : Listes d'adjacence d'un graphe orienté

Soit $G = (S, A)$ un graphe orienté avec $S = \{1, \cdots, n\}$.
Les listes d'adjacence de G sont des listes $(l_i)_{1 \leqslant i \leqslant n}$ tels que, pour tout $1 \leqslant i, j \leqslant n$, $j \in l_i$ si et seulement si $(i, j) \in A$.

Exercice 10. Quelles sont les listes d'adjacence du graphe cyclique à n sommets ?

Définition 34 : Listes d'adjacence d'un graphe non-orienté

Soit $G = (S, A)$ un graphe non-orienté avec $S = \{1, \cdots, n\}$.
Les listes d'adjacence de G sont des listes $(l_i)_{1 \leqslant i \leqslant n}$ tels que, pour tout $1 \leqslant i, j \leqslant n$, $j \in l_i$ si et seulement si $\{i, j\} \in A$.

Exemple 9 : Listes d'adjacence en OCaml

On peut représenter les listes d'adjacence d'un graphe à l'aide du type suivant :

```
type graphe = int list array
```

Exemple 10 : Calcul du nœud d'arité maximale grâce aux listes d'adjacence

```
let arite_max g =
    let max_vu = ref 0 in
    let n = Array.length g in
    for i = 0 to n do
        max_vu = max (!max_vu) (List.length g.(i))
    done ;
    !max_vu
```

Exemple 11 : Listes d'adjacence en OCaml pour les graphes pondérés

On peut représenter les listes d'adjacence d'un graphe pondéré par des flottants à l'aide du type suivant :

```
type graphe = (int * float) list array
```

Proposition 17 : Taille mémoire d'un graphe par liste d'adjacence

La taille mémoire d'un graphe représenté par sa liste d'adjacence est en $O(|S| + |A|)$.

Proposition 17 : Taille mémoire d'un graphe par liste d'adjacence

La taille mémoire d'un graphe représenté par sa liste d'adjacence est en $O(|S| + |A|)$.

Proposition 18 : Coût de parcours des arêtes par matrice d'adjacence

Le parcours de toutes les arêtes d'un graphe représenté par sa liste d'adjacence est en $O(|S| + |A|)$.

Proposition 17 : Taille mémoire d'un graphe par liste d'adjacence

La taille mémoire d'un graphe représenté par sa liste d'adjacence est en $O(|S| + |A|)$.

Proposition 18 : Coût de parcours des arêtes par matrice d'adjacence

Le parcours de toutes les arêtes d'un graphe représenté par sa liste d'adjacence est en $O(|S| + |A|)$.

Proposition 19 : Coût d'accès à une arête par matrice d'adjacence

L'accès à une arrête par liste d'adjacence à l'aide des indices se fait en $O(|S|)$.

Exemple 12 : Listes d'adjacence en C

On peut utiliser un tableau de tableaux à l'aide du type `int **` pour représenter les listes d'adjacence en C.

Exemple 12 : Listes d'adjacence en C

On peut utiliser un tableau de tableaux à l'aide du type `int **` pour représenter les listes d'adjacence en C.

Exemple 13 : Recherche d'arête par liste d'adjacence en C

```c
bool est_voisin(int ** g, int i, int j){
    for(int k = 1; k<g[i][0];k++) {
        if (g[i][k] == j){
            return true;
        }
    }
    return false;
}
```

Exemple 12 : Listes d'adjacence en C

On peut utiliser un tableau de tableaux à l'aide du type `int **` pour représenter les listes d'adjacence en C.

Exemple 13 : Recherche d'arête par liste d'adjacence en C

```c
bool est_voisin(int ** g, int i, int j){
    for(int k = 1; k<g[i][0];k++) {
        if (g[i][k] == j){
            return true;
        }
    }
    return false;
}
```

Exercice 11. Proposer une fonction de prototype `int ** cyclique(int n)` qui construit le graphe représenté par listes d'adjacence qui correspond au graphe cyclique de taille n.

Proposition 20 : Coûts de la représentation par listes d'adjacence en C

En C, on obtient les mêmes complexités qu'en OCaml avec les représentations par listes d'adjacence.

	Matrices d'adjacence	Listes d'adjacence
Complexité mémoire	$O(\|S\|^2)$	$O(\|S\| + \|A\|)$
Complétixé de l'accès à une arête donnée par ses indices	$O(1)$	$O(\|S\|)$
Complexité du parcours de toutes les arêtes	$O(\|S\|^2)$	$O(\|S\| + \|A\|)$

Parcours de graphes

Définition 35 : Parcours des sommets d'un graphe

Un **parcours de graphe** est un algorithme qui traite tous les sommets d'un graphe une fois exactement.

Algorithme 1 : Structure générale d'un parcours

Dans la plupart des cas, les parcours prennent la forme suivante.

Pour Chaque Sommet s dans le graphe **Faire**
- **Si** Si s n'est pas vu ou en attente **Alors**
  - On ajoute s aux sommets en attente.
  - **Tant Que** Il y a un sommet en attente **Faire**
    - Soit u un sommet en attente.
    - u est ajouté aux sommets traités.
    - On ajoute les voisins de u qui ne sont pas traités ou en attente aux sommets en attente.

Définition 36 : Parcours en profondeur d'un graphe

Un **parcours en profondeur** d'un graphe est un parcours de nœud qui, pour chaque voisin, traite entièrement les nœuds accessibles depuis ce voisin parmi les nœuds qui sont encore à traiter avant de passer au voisin suivant.

Définition 37 : Parcours préfixe d'un graphe

Un **parcours préfixe** d'un graphe est un parcours en profondeur qui traite le nœud après avant d'avoir traité tous les nœuds accessible depuis lui.

Définition 37 : Parcours préfixe d'un graphe

Un **parcours préfixe** d'un graphe est un parcours en profondeur qui traite le nœud après avant d'avoir traité tous les nœuds accessible depuis lui.

Définition 38 : Parcours postfixe d'un graphe

Un **parcours postfixe** d'un graphe est un parcours en profondeur qui traite le nœud après avoir traité tous les nœuds accessible depuis lui.

```
let afficher_prefixe_graphe g =
    let n = Array.length g in
    let deja_vu = Array.make n false in
    let rec explorer k =
        if not deja_vu.(k) then
            begin
                deja_vu.(k)<- true;
                Printf.printf "%d\n" k ;
                for i = 0 to n-1 do
                    if g.(k).(i) ==1 then
                        explorer i
                done
            end
    in
    for i = 0 to n-1 do
        explorer i
    done
```

```
let afficher_prefixe_graphe g =
    let n = Array.length g in
    let deja_vu = Array.make n false in
    let rec explorer k =
        if not deja_vu.(k) then
            begin
                deja_vu.(k)<- true;
                Printf.printf "%d\n" k ;
                for i = 0 to n-1 do
                    if g.(k).(i) ==1 then
                        explorer i
                done
            end
    in
    for i = 0 to n-1 do
        explorer i
    done
```

Exercice 12. Proposer une implémentation de la fonction précédente quand le graphe est représenté par ses listes d'adjacence.

Définition 39 : Parcours en largeur

Un **parcours en largeur** est un parcours qui traite les voisins avant de traiter les nœuds qui sont encore à traiter parmi les nœuds accessibles depuis les voisins.

Définition 39 : Parcours en largeur

Un **parcours en largeur** est un parcours qui traite les voisins avant de traiter les nœuds qui sont encore à traiter parmi les nœuds accessibles depuis les voisins.

Exercice 13. En utilisant une file, proposer une implémentation d'une fonction qui affiche les nœuds dans l'ordre obtenu par le parcours en largeur.

Proposition 21 : Arbre obtenu à partir d'un parcours de proche en proche

Lors d'un parcours de proche en proche d'un graphe connexe, le sous-graphe constitué des nœuds du graphe et dont les arêtes sont les arêtes qui ont permis d'ajouter un nœud à la liste d'attente est un arbre.
On parle de l' **arbre de parcours d'un graphe**.

Proposition 21 : Arbre obtenu à partir d'un parcours de proche en proche

Lors d'un parcours de proche en proche d'un graphe connexe, le sous-graphe constitué des nœuds du graphe et dont les arêtes sont les arêtes qui ont permis d'ajouter un nœud à la liste d'attente est un arbre.
On parle de l' **arbre de parcours d'un graphe**.

Exercice 14. Proposer une fonction qui réalise un parcours en profondeur et qui créer l'arbre de ce parcours.

Recherche de chemin dans un graphe

Définition 40 : Accesibilité dans un graphe orienté

Un nœud v est accessible depuis un autre nœud u dans un graphe orienté s'il existe un chemin de u à v.

Définition 41 : Accessibilité dans un graphe non-orienté

Un nœud v est accesible depuis un autre nœud u dans un graphe non-orienté s'il existe un chemin entre u et v.

Proposition 22 : Connexité et accessibilité dans un graphe orienté

Dans un graphe orienté, tous les éléments d'une composante fortement connexe sont accessibles depuis chaque élément de la composante fortement connexe.
De plus, pour chaque composante fortement connexe et un sommet de u de cette composante connexe, la composante fortement connexe est exactement les nœuds v accessibles depuis u tels que u soit accessible depuis u, ainsi que u.

Proposition 23 : Connexité et accessibilité dans un graphe non-orienté

Dans un graphe non-orienté, tous les éléments d'une composante connexe sont accessibles depuis chaque élément de la composante connexe.
De plus, pour chaque composante connexe et un u de cette composante connexe, la composante connexe est exactement l'ensemble des nœuds accessibles depuis u.

Algorithme 2 : Trouver les composantes connexes dans un graphe non-orienté

$\mathcal{P} \leftarrow \varnothing$

Tant Que il existe un sommet qui ne soit pas dans un ensemble de la partition $\mathcal{P}$ **Faire**

> $s \leftarrow$ un sommet qui n'est pas dans un ensemble de la partition $\mathcal{P}$.
>
> $C \leftarrow$ la composante connexe de s obtenue par parcours de proche en proche.
>
> $\mathcal{P} \leftarrow \mathcal{P} \cup \{C\}$

Renvoyer P la partition obtenue

Exercice 15. Proposer une implémentation en C de cet algorithme en utilisant une matrice de taille statique.
On utiliseras un tableau de taille $|S|$ pour stocker les numéros des composantes connexes des éléments.
Quelle est sa complexité temporelle ?

Définition 42 : Plus court chemin entre deux nœuds dans un graphe orienté

Soit $G = (S, A)$ un graphe orienté. Soit $u, v \in G$. On note $d(u, v)$ la **distance de u à v** la grandeur définie de la manière suivante :

$$d(u,v) = \begin{cases} 0 & \text{si } u = v, \\ +\infty & \text{s'il n'existe pas de chemin de } u \text{ à } v, \\ l & \text{sinon.} \end{cases}$$

Où l est le minimum des distances parmi les chemins de u à v s'il existe.
Un chemin de u à v qui atteint ce minimum est un **plus court chemin de u à v**.

Définition 43 : Plus court chemin entre deux nœuds dans un graphe non-orienté

Soit $G = (S, A)$ un graphe non orienté. Soit $u, v \in G$. On note $d(u, v)$ la **distance de u à v** la grandeur définie de la manière suivante :

$$d(u,v) = \begin{cases} 0 & \text{si } u = v, \\ +\infty & \text{s'il n'existe pas de chemin entre } u \text{ et } v, \\ l & \text{sinon.} \end{cases}$$

Où l est le minimum des distances parmi les chemins entre u et v s'il existe.
Un chemin de u à v qui atteint ce minimum est un **plus court chemin de u à v**.

Définition 44 : Rayon d'un graphe

Le rayon d'un graphe est le minimum sur les nœuds des maximum des distances aux autres nœuds.
Ainsi, pour $G = (S, A)$ un graphe, on a le rayon de G noté $r(G)$ égal à :

$$r(G) = \min_{u \in S} \max_{v \in S} d(u, v)$$

Définition 44 : Rayon d'un graphe

Le rayon d'un graphe est le minimum sur les nœuds des maximum des distances aux autres nœuds.
Ainsi, pour $G = (S, A)$ un graphe, on a le rayon de G noté $r(G)$ égal à :

$$r(G) = \min_{u \in S} \max_{v \in S} d(u, v)$$

Définition 45 : Diamètre d'un graphe

Soit $G = (S, A)$. Le **diamètre** de ce graphe est le maximum des distances entre deux nœuds.
Ainsi, pour $G = (S, A)$ un graphe, on a le rayon de G noté $\mathrm{diam}(G)$ égal à :

$$\mathrm{diam}(G) = \max_{u, v \in S} d(u, v)$$

Proposition 24 : Diamètre et rayon d'un graphe non connexe

Le diamètre et le rayon d'un graphe non-orienté non connexe sont infinis.

Algorithme 3 : Plus court chemin dans un graphe

Pour obtenir le plus court chemin dans un graphe non-orienté (*resp.* orienté) entre u et v (*resp.* de u à v), on peut réaliser un parcours en largeur du graphe depuis u et s'arêter dès que l'on trouve v.

Définition 46 : Poids d'un chemin dans un graphe pondéré

Le **poids d'un chemin** $c = u_0 \cdots u_n$ est donné par la somme des poids des arêtes qui le compose.
Ainsi, dans un graphe orienté :

$$p(u_0 \cdots u_n) = \sum_{k=0}^{n-1} p((u_k, u_{k+1}))$$

Et dans un graphe non-orienté :

$$p(u_0 \cdots u_n) = \sum_{k=0}^{n-1} p(\{u_k, u_{k+1}\})$$

Définition 47 : Distance entre deux sommets dans un graphe pondéré orienté

La distance de u à v est déterminer par l'infimum sur les poids des chemins de u à v, en prenant 0 si $u = v$, et $+\infty$ s'il n'y a pas de chemins de u à v.
Ainsi :

$$d(u,v) = \begin{cases} 0 & \text{si } u = v, \\ +\infty & \text{s'il n'existe pas de chemins de } u \text{ à } v, \\ \inf_{c \in \mathcal{C}(u,v)} p(c) & \text{sinon.} \end{cases}$$

Où $\mathcal{C}(u,v)$ est l'ensemble des chemins de u à v.

Définition 48 : Distance entre deux sommets dans un graphe pondéré non-orienté

La distance de u à v est déterminer par l'infimum sur les poids des chemins entre u et v, et $+\infty$ s'il n'y a pas de chemins entre u et v.
Ainsi :

$$d(u,v) = \begin{cases} +\infty & \text{s'il n'existe pas de chemins entre } u \text{ et } v, \\ \inf_{c \in \mathcal{C}(u,v)} p(c) & \text{sinon.} \end{cases}$$

Où $\mathcal{C}(u,v)$ est l'ensemble des chemins entre u et v, en condisérant le chemin vide dans le cas $u = v$ (qui nous donne donc une distance maximale de 0).

Définition 49 : Distance d'un sous-graphe à un sommet

Soit $G = (S, A)$ un graphe orienté. Soit $G' = (S', A')$ un sous-graphe de G et soit $s \in S$.
La **distance de** G' **à** s, ou la **distance de** S' **à** s, est la grandeur :

$$d(S', s) = d(G', s) = min_{s' \in S'} d(s', s)$$

Algorithme 4 : Algorithme de Dijkstra

On se donne $G = (S, A, p)$ un graphe orienté pondéré avec des poids positifs. Soit s un sommet source.

$P \leftarrow \varnothing$

$d[u] \leftarrow +\infty$ pour chaque sommet $u \in S$.

$d[s] \leftarrow 0$

Tant Que $S \setminus P$ contient au moins un sommet tel que $d[a] < +\infty$ **Faire**

- $a \leftarrow$ le sommet de $S \setminus P$ de plus petite distance $d[a]$.
- $P \leftarrow \{a\} \cup P$
- **Pour Chaque** sommet $b \in S \setminus P$ voisin de a **Faire**
  - **Si** $d[b] > d[a] + p(a, b)$ **Alors**
    - $d[b] \leftarrow d[a] + p(a, b)$

À la fin de l'algorithme, d contient le tableau des distances de s à chacun des sommets $u \in S$.

Définition 50 : Arbre construit par l'algorithme de Dijkstra

Soit $G = (S, A)$ un graphe orienté et soit $s \in S$ un sommet. L'arbre construit par l'algorithme de Dijkstra est un sous-graphe $G' = (S', A')$ de G dont les sommets sont dans l'ensemble P construit, et les arêtes sont, pour chaque sommet b sauf pour le sommet s, l'arête (a, b) qui ait en dernier modifié la valeur de $d[b]$.

Définition 50 : Arbre construit par l'algorithme de Dijkstra

Soit $G = (S, A)$ un graphe orienté et soit $s \in S$ un sommet. L'arbre construit par l'algorithme de Dijkstra est un sous-graphe $G' = (S', A')$ de G dont les sommets sont dans l'ensemble P construit, et les arêtes sont, pour chaque sommet b sauf pour le sommet s, l'arête (a, b) qui ait en dernier modifié la valeur de $d[b]$.

Proposition 25 : Distance dans l'arbre construit par l'algorithme de Dijkstra

Soit $G = (S, A)$ un graphe orienté.
Soit u un sommet. Soit v un sommet accessible depuis u et soit $G' = (S', A')$ l'arbre construit par l'algorithme de Dijkstra. L'unique chemin de u à v dans G' est un plus court chemin de u à v dans G.

$P \leftarrow \varnothing$
$d[u] \leftarrow +\infty$ pour chaque sommet $u \in S$.
$d[s] \leftarrow 0$
Tant Que $S \setminus P$ contient au moins un sommet tel que $d[a] < +\infty$
Faire
    $a \leftarrow$ le sommet de $S \setminus P$ de plus petite distance $d[a]$.
    $P \leftarrow \{a\} \cup P$
    Pour Chaque sommet $b \in S \setminus P$ voisin de a **Faire**
        Si $d[b] > d[a] + p(a, b)$ **Alors**
            $d[b] \leftarrow d[a] + p(a, b)$
            predecesseur$[b] \leftarrow a$

Proposition 26 : Complexité naïve de l'implémentation de l'algorithme de Dijkstra

L'algorithme de Dijkstra naïf a une complexité temporelle en $O(n^2)$ où n est le nombre de sommets du graphe.

```
let distance_dijkstra g u v =
let n = Array.length g in
let deja_vu = Array.make  n false in
let t = creer_tas () in
let rec aux_liste d l = match l with
| [] -> ()
| (distance, sommet) ::q ->
    ajouter_tas t (distance + d, sommet);
    aux_liste d q
and aux_boucle () =
    let (d, sommet) = retirer_tas t in
    if sommet = v then d
    else
        begin
            if not deja_vu.(sommet) then
                begin
                    deja_vu.(sommet)<-true ;
                    aux_liste d g.(sommet)
                end ;
            aux_boucle ()
        end
in ajouter_tas t (0, u) ; aux_boucle ()
```

Proposition 27 : Complexité en utilisant un tas binaire de l'algorithme de Dijkstra

L'algorithme de Dijkstra avec un tas binaire a une complexité temporelle en $O((n+m)\log n)$ où n est le nombre de sommets du graphe et m est le nombre d'arêtes du graphe.

Algorithme 5 : Algorithme de Floyd-Warshall

Soit $G = (S, A, p)$ un graphe orienté pondéré. On note $n = |S|$.

Pour Chaque $i, j \in S$ **Faire**

$$m[i][j] \leftarrow \begin{cases} 0 & \text{si } i = j, \\ p((i,j)) & \text{si } (i,j) \in A, \\ +\infty & \text{sinon.} \end{cases}$$

Pour Chaque k de 1 à n **Faire**

  Pour Chaque i de 1 à n **Faire**

    Pour Chaque j de 1 à n **Faire**

      Si $m[i][j] > m[i][k] + m[k][j]$ **Alors**

        $m[i][j] \leftarrow m[i][k] + m[k][j]$

À la fin de l'algorithme, la matrice M contient les distances de i à j de sorte à ce que $m[i][j] = d(i,j)$.

$$m_{i,j}^{(0)} = \begin{cases} 0 & \text{si } i = j, \\ p((i,j)) & \text{si } (i,j) \in A, \\ +\infty & \text{sinon.} \end{cases}$$

$$m_{i,j}^{(k)} = \min\left(m_{i,j}^{(k-1)}, m_{i,k}^{(k-1)} + m_{k,j}^{(k-1)}\right)$$

```
void floyd_warshall(int m[taille][taille]){
    for (int k = 0; k< taille; k++){
        for (int i= 0; i< taille; i++){
            for (int j= 0; j< taille; j++){
                if (m[i][k]!=INT_MAX && m[k][j]!=
  INT_MAX){
                    if (m[i][j]>m[i][k]+m[k][j]){
                        m[i][j]= m[i][k]+m[k][j]
                    }
                }
            }
        }
    }
}
```

Proposition 28 : Complexité de l'algorithme de Floyd-Warshall

L'algorithme de Floyd-Warshall a une complexité temporelle en $O(n^3)$ où n est le nombre de sommets du graphe.

	Dijkstra	Floyd-Warshall
Résultat	Tous les plus courts chemins depuis un sommet d'origine s	Tous les plus courts chemins entre deux sommets
Représentation	Listes d'adjacence	Matrice d'adjacence
Complexité temporelle	$O((\lvert S\rvert + \lvert A\rvert)\log\lvert S\rvert)$ avec un tas	$O(\lvert S\rvert^3)$

Graphes orientés acycliques

Définition 51 : Puit, source

Soit $G = (S, A)$ un graphe orienté.
Un **puit** est un sommet de degré sortant nul.
Une **source** est un sommet de degré entrant nul.

Définition 51 : Puit, source

Soit $G = (S, A)$ un graphe orienté.
Un **puit** est un sommet de degré sortant nul.
Une **source** est un sommet de degré entrant nul.

Proposition 29 : Présence d'une source et d'un puit dans un graphe orienté acyclique

Un graphe orienté acyclique $G = (S, A)$ avec S non vide admet au moins un puit.
Un graphe orienté acyclique $G = (S, A)$ avec S non vide admet au moins une source.

Proposition 30 : Nombre d'arêtes dans un graphe orienté acyclique

Dans un graphe acyclique $G = (S, A)$, il y a au plus $\frac{|S|(|S|-1)}{2}$ arêtes.

Définition 52 : Tri topologique

Soit $G = (S, A)$ un graphe orienté avec n sommets. Un tri topologique est une énumération des sommets $u_1, ... u_n$. telle que pour pour tout $1 \leqslant i \leqslant j \leqslant n$, on ait $(j, i) \notin A$.
Un tri topologique nous dit que pour tout arrête $(u_j, u_i) \in A$, on a $j < i$.

Définition 52 : Tri topologique

Soit $G = (S, A)$ un graphe orienté avec n sommets. Un tri topologique est une énumération des sommets $u_1, \ldots u_n$. telle que pour pour tout $1 \leqslant i \leqslant j \leqslant n$, on ait $(j, i) \notin A$.
Un tri topologique nous dit que pour tout arrête $(u_j, u_i) \in A$, on a $j < i$.

Proposition 31 : Tri topologique et graphe orienté acyclique

Soit $G = (S, A)$ un graphe orienté. G est acyclique si et seulement si G admet un tri topologique.

Proposition 32 : Algorithme

L'ordre postfixe d'un parcours en profondeur à partir des sources d'un graphe orienté acyclique nous donne un tri topologique de ses nœuds.

```
let tri_topologique g =
    let n = Array.length g in
    let est_source = trouver_sources g in
    let deja_vu = Array.make n false in
    let rec aux_liste l acc = match l with
    | [] -> acc
    | p::q -> aux_liste q (aux_sommet p acc)
    and aux_sommet i acc =
        if deja_vu.(i) then acc
        else
            begin
                deja_vu.(i)<- true ;
                i::(aux_liste g.(i) acc)
            end
    and aux_source i acc =
        if i = n then acc
        else
            if est_source.(i) then
                aux_source (i+1) (aux_sommet i acc)
            else
                aux_source (i+1) acc
    in (aux_source 0 [])
```

Graphes de flux de contrôle

Définition 53 : Graphe de flux de contrôle d'un programme

On se donne P un programme. Le **graphe de flux de contrôle d'un programme** est un graphe orienté dont les sommets sont les instructions de base du programme et dont les arêtes sont les sauts possibles qu'il peut y avoir entre chaque instruction de base.
Souvent, on étiquette les arêtes par la condition nécessaire pour que cette transition soit la transition sélectionnée, et on rajoute les arêtes pour les points d'entrée et de sortie du programme.

Exemple de condition

```c
int f0(int n){
    int i = 0;
    if (n>0){
        i++;
    }
    else {
        i--;
    }
    return i;
}
```

Exemple de boucle

```
int f1(int n){
    int i = 0;
    while (i<n){
        i++;
    }
    return i;
}
```

Exemple de code mort

```c
int f2(int n){
    int i = n;
    return i;
    i++;
}
```

Exemple de boucle infinie

```
int f3(int n){
    int i = 0;
    while (true){
        i++;
    }
}
```

Définition 54 : Chemin d'une exécution

Un **chemin d'exécution** est le chemin parcouru dans le graphe à partir d'un entrée donnée.

Exemple de chemin non-faisable

```c
int f4(int n){
    int i = 0;
    if (n>2){
        i++;
    }
    else{
        i--;
    }
    if (n>0){
        i++;
    }
    else{
        i--;
    }
    return i;
}
```

Définition 55 : Couverture de test

En se donnant un ensemble d'entrées, la **couverture de test** est la partie du graphe de flux de contrôle qui est visitée par les chemins d'exécution avec les entrées.

Définition 56 : Couverture de test pour les sommets

La **couverture de test pour les sommets d'un programme** est la partie des sommets qui est visitée par au moins un des tests.

Définition 56 : Couverture de test pour les sommets

La **couverture de test pour les sommets d'un programme** est la partie des sommets qui est visitée par au moins un des tests.

Définition 57 : Couverture de test pour les arêtes

La **couverture de test pour les arêtes d'un programme** est la partie des arêtes qui est visitée par au moins un des tests.

Définition 56 : Couverture de test pour les sommets

La **couverture de test pour les sommets d'un programme** est la partie des sommets qui est visitée par au moins un des tests.

Définition 57 : Couverture de test pour les arêtes

La **couverture de test pour les arêtes d'un programme** est la partie des arêtes qui est visitée par au moins un des tests.

Définition 58 : Couverture de test pour les chemins

La **couverture de test pour les chemins d'un programme** est la partie des chemins qui est visitée par au moins un des tests.

```
bool dichotomie(int e, int* t, int n){
    int i = 0;
    int j = n-1;
    while(i<=j){
        int m = (i+j)/2;
        if (t[m]==e)
            return true;
        if (t[m]<= e)
            i = m+1;
        else
            j = m-1;
    }
    return false;
}
```

```
bool dichotomie(int e, int* t, int n){
    int i = 0;
    int j = n-1;
    while(i<=j){
        int m = (i+j)/2;
        if (t[m]==e)
            return true;
        if (t[m]<= e)
            i = m+1;
        else
            j = m-1;
    }
    return false;
}
```

Exercice 16. Écrire le graphe de flux de contrôle de ce code. Proposer un jeu de tests qui couvre toutes les arêtes.