Informatique

CPGE

Marc de Falco

Table des matières

1. Graphes

■ **Note 1.1** Une grande partie est issue verbatim de mon ancien poly. À affiner.

■

I Graphes orientés

I.1 Définition

Définition I.1 Un graphe orienté est un couple $G = (S, A)$ où S est un ensemble fini et A est une relation **irréflexive** sur S, c'est-à-dire $S \subset \{(x, y) \mid x, y \in S, x \neq y\}$.

Les éléments de S sont appelés les **sommets** du graphe G et les éléments de A les **arêtes**. Si $(x, y) \in A$, on dit que x est la source de l'arête et y est son but. Quand le contexte n'est pas ambigu, on notera $x \to y$.

■ **Exemple 1.1**
- $G = (\{a, b, c\}, \{(a, b), (b, c), (c, b)\})$
- $D_n = (\llbracket 1, n \rrbracket, \{(a, b) \mid a \text{ divise } b\})$

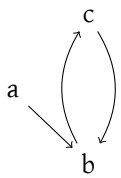
■

■ **Remarque 1.1** Ici, on a exclu les graphes infinis ainsi que les graphes avec des boucles ou des arêtes parallèles. On est dans le contexte des **graphes finis simples**.

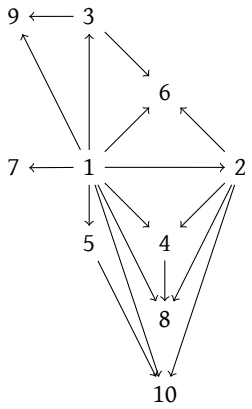
■

On représente graphiquement un graphe sous la forme d'un diagramme sagittaire où les sommets sont des points et où une arêtes (a, b) est une flèche allant du point a au point b.

■ **Exemple 1.2**
- G est représenté par



- D_{10} est représenté par



■

I.2 Voisins et degrés

Définition I.2 Soit $G = (S, A)$ un graphe orienté et $x \in S$.

On appelle **voisins sortants**, ou **successeurs**, de x les éléments de

$$v_+(x) = \{\ y \in S \mid (x, y) \in A\ \}$$

et on appelle **degré sortant** de x le cardinal de cet ensemble $d_+(x) = |v_+(x)|$.

De même, on parle de **voisins entrants**, ou **prédéceseurs** pour les éléments de

$$v_-(x) = \{\ z \in S \mid (z, x) \in A\ \}$$

et on parle de **degré entrant** pour $d_-(x) = |v_-(x)|$.

Théorème I.1 Soit $G = (S, A)$ un graphe, on a $\sum_{x\in S} d_+(x) = \sum_{x\in S} d_-(x) = |A|$

Démonstration.

On peut partitionner A en regroupant les arêtes de même source, on a ainsi :

$$A = \bigcup_{x\in S} \{\ (x, y) \mid y \in S, (x, y) \in A\ \} = \bigcup_{x\in S} \{\ (x, y) \mid y \in v_+(x)\ \}$$

Et en prenant la cardinal de cette égalité, on obtient directement $|A| = \sum_{x\in S} d_+(x)$. L'autre égalité est symétrique en considérant les arêtes de même but.

■

I.3 Chemin

Définition I.3 Soit $G = (S, A)$ un graphe orienté et $x, y \in S$.

Une suite finie $\varphi = (s_0, s_1, \ldots, s_p)$ où $p \in \mathbb{N}, s_0 = x, s_p = y$ et

$$\forall i \in [\![1, p]\!], (s_{i-1}, s_i) \in A$$

est appelée un **chemin**, de longueur p, de x vers y. On a donc

$$s_0 \to s_1 \to \cdots \to s_p$$

On notera $\varphi : x \rightsquigarrow y$ pour signifier que φ est un tel chemin et $x \rightsquigarrow y$ pour signifier qu'il existe un chemin de x vers y. On dit alors que y est accessible depuis x.

Théorème I.2 $\rightsquigarrow$ est la plus petite relation sur S contenant $\rightarrow$ et qui soit réflexive et transitive.

Démonstration.
On considère $\rightarrow \subset R \subset S^2$ telle que R soit réflexive et transitive.
Soient $x, y \in S$ et $\varphi = (s_0, \dots, s_p) : x \rightsquigarrow y$.
Comme $s_{i-1} \rightarrow s_i$ on a $s_{i-1} R s_i$ et, par transitivité, $x = s_0 R s_p = y$. Donc $x \rightsquigarrow y \Rightarrow xRy$.
Ainsi $\rightsquigarrow \subset R$.

■

On dit que $\rightsquigarrow$ est la clôture réflexive et transitive de $\rightarrow$.

■ **Remarque 1.2** Vocabulaire additionnel :
- Si tous les sommets sont distincts, sauf éventuellement le premier et le dernier, on dit que le chemin est **élémentaire**.
- Si toutes les arêtes sont distinctes, on dit que le chemin est **simple**.
- Si $x = y$, on dit que le chemin est **fermé**.
- Un chemin simple dont tous les sommets sont distincts est appelé une **chaîne**.
- Un chemin élémentaire fermé simple de longueur au moins 1 est appelé un **cycle**. *Comme le chemin vide issu de x est fermé, il est nécessaire de considérer des chemins non vides.*
- Un graphe contenant au moins un cycle est dit **cyclique**. Dans le cas contraire, on dit qu'il est **acyclique**.

■

■ **Remarque 1.3** Si $\varphi : x \rightsquigarrow y$ et $\psi : y \rightsquigarrow z$ on note $\varphi\psi : x \rightsquigarrow z$ la concaténée des deux chemins.

■

Définition I.4 On note $\leftrightarrow y$ la plus grande relation d'équivalence incluse dans $\rightsquigarrow$.
Plus précisément, on a

$$x \leftrightarrow y \iff (x \rightsquigarrow y \wedge y \rightsquigarrow x)$$

Les classes d'équivalences pour $\leftrightarrow$ sont appelées les composantes **fortement connexes** du graphe. S'il n'y a qu'une classe, on dit que le graphe est **fortement connexe**.

I.4 Sous-graphe

Définition I.5 Soit $G = (S, A)$ et $X \subset S$. On appelle **sous-graphe** induit par X le graphe (X, A_X) où

$$A_X = \{\ (a, b) \in A \mid a \in X \wedge b \in X\ \}$$

Un graphe G' est un **sous-graphe** de G quand c'est le sous-graphe de G induit par les sommets de G' (ils sont alors nécessairement des sommets de G).

■ **Remarque 1.4** Les composantes fortement connexes sont les sous-graphes fortement connexes maximaux pour l'inclusion.

■

I.5 Implémentation

I.5.i Énumération de sommets

Si on considère un graphe $G = (S, A)$, il est assez naturel de représenter ses sommets dans un tableau. Pour cela, on fixe naturellement un ordre sur ces sommets et on va associer à chaque sommet son indice dans le tableau.

Par exemple, si $S = \{a, b, c, d\}$ on va pouvoir considérer `[ a, b, c, d ]` et ainsi associer à `a` son indice `0` dans le tableau. L'ordre est arbitraire, on aurait pu considérer `[ b, c, a, d ]` et l'indice de `a` aurait alors été 2.

Ce qui compte, c'est de pouvoir travailler directement sur les indices et pas sur les éléments. Une manière de s'en convaincre est d'imaginer le graphe d'un réseau social où un sommet correspond au profil d'une personne, et contient donc beaucoup (*trop*) d'informations. Il est bien plus raisonnable de lui associer un identifiant unique et d'utiliser cet identifiant ensuite.

Quand on va implémenter des graphes, on peut donc supposer que les sommets sont les entiers de `0` à `n-1` où $|S| = n$. Il sera toujours possible de retrouver la correspondance avec les sommets eux-mêmes.

■ **Remarque 1.5** Si on est bloqué par le coût de l'opération permettant d'obtenir l'indice d'un sommet, qui est en $O(|S|)$ par une recherche linéaire, on peut très bien définir un dictionnaire associant à chaque sommet son indice en $O(1)$.

Il se trouve qu'on ne va pas forcément s'intéresser à ces questions mais plus aux raisonnements sur la structure du graphe.

■

I.5.ii Matrice d'adjacence

Définition I.6 Soit $G = (S, A)$ un graphe et $S = \{s_1, \dots, s_n\}$ une énumération sans répétition des sommets de G. La matrice $M_G = (m_{ij})_{1 \leq i,j \leq n} \in \mathcal{M}_n(\mathbb{R})$ définie par

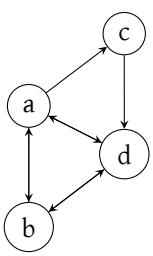
$$\forall i, j \in [\![1, n]\!], m_{ij} = \begin{cases} 1 & \text{si } s_i \to s_j \\ 0 & \text{sinon} \end{cases}$$

est appelée une **matrice d'adjacence** de G.

■ **Remarque 1.6** Il s'agit bien d'**une** matrice d'adjacence et pas de **la** matrice car elle dépend de l'énumération des sommets.

■

■ **Exemple 1.3** Si on considère le graphe



On aura pour l'énumération a, b, c, d la matrice :

$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

En fait, la ième ligne correspond au ième sommet de l'énumération. Ici, la dernière ligne 1100 correspond au sommet d et donne dans l'ordre a, b, c, d la présence ou non d'une arête $d \to *$.

Si on considère l'énumération b, a, d, c on aura la matrice :

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

■

On en déduit ainsi une implémentation directe d'un graphe en représentant la matrice d'adjacence comme l'a fait pour des images.

Python
```
m = [ [ 0, 1, 1, 1 ],
      [ 1, 0, 0, 1 ],
      [ 0, 0, 0, 1 ],
      [ 1, 1, 0, 0 ] ]
```

■ **Remarque 1.7** Il y a le même lien entre les différentes matrices d'adjacence d'un graphe donnée et la matrice d'un endomorphisme dans les différentes permutations d'une même base, à chaque fois on obtient la nouvelle matrice en permutant de même ses lignes et ses colonnes.

■

Cette représentation permet d'accéder, en lecture comme en écriture, à une arête en temps constant. Cependant, pour récupérer les voisins d'un sommet, il est nécessaire de parcourir toute la ligne correspondante, donc en $O(|S|)$.

■ **Remarque 1.8** Dans la majorité des cas, A est petit par rapport à $|S|^2$. Ainsi, la matrice contient très peu de 1 et beaucoup de 0, on dit qu'elle est *creuse*. Une telle matrice *creuse* peut-être efficacement représenté par l'ensemble fini des arêtes (i, j). Par exemple, avec une table de hachage. Cependant, dans le cas des graphes, il est possible d'avoir une structure creuse plus efficace.

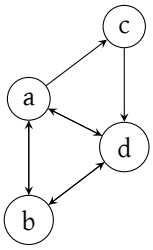
A titre d'exemple, un graphe issu d'un réseau social est présenté plus bas, il contient 300000 arêtes pour 8000 sommets. Donc le rapport $|A|/|S|^2$ est de 4 pour 1000 !

■

I.5.iii Listes d'adjacences

La donnée de $v_+(x)$ pour chaque sommet $x \in S$ suffit à reconstruire A.

■ **Exemple 1.4** Si on reprend le graphe précédent,



on a $S = \{a, b, c, d\}$ et $A = \{(a, b), (a, c), (a, d), (b, a), (b, d), (c, d), (d, a), (d, b)\}$.
Or $v_+(a) = \{b, c, d\}, v_+(b) = \{a, d\}, v_+(c) = \{d\}$ et $v_+(d) = \{a, b\}$.
Si on pose $V_+(x) = \{\ (x, y) \mid y \in v_+(x)\ \}$ on a directement $A = \bigcup_{x \in S} V_+(x)$.

■

On en déduit ainsi une représentation d'un graphe où on place dans un tableau chaque $v_+(x)$ représenté par une liste chaînée ou un tableau dynamique.

■ **Remarque 1.9** Pour représenter $v_+(x)$ on a plusieurs choix : on peut utiliser directement les valeurs des sommets ou utiliser des indices dans une énumération. C'est le second choix qu'on fera en général car il est plus simple. Cependant, en cas de suppression d'un sommet, les indices changent, et il faut renuméroter dans les listes.

Pour le graphe précédent, on pourra donc considérer une énumeration `[ a, b, c, d ]` et la représentation sous forme de liste d'adjacence pourra être :

!listing2(python)(ocaml)

```
l = [ [ 1, 2, 3 ], [ 0, 3 ], [ 3 ], [ 0, 1 ] ]
```

```
let l = [|
    [ 1; 2; 3 ]; [ 0; 3 ]; [ 3 ]; [ 0; 1 ]
    |]
```

■

L'accès en lecture ou en écriture à une arête est alors en $O(|A|)$ mais on peut parcourir les voisins sortant en $O(|A|)$ également. Pour un sommet donné, on peut même préciser $O(d_+(x))$. Accéder à la liste peut même se faire en $O(1)$.

Pour obtenir les voisins entrants, il est par contre nécessaire de tester la présence de x dans chacune des autres listes, on obtient donc un algorithme en $O(|S| + |A|)$: on parcourt chaque case du tableau des listes puis chaque maillon de listes d'adjacence.

Il est possible d'améliorer cela en utilisant un structure plus efficace pour stocker les ensembles. Cela peut-être un dictionnaire reposant sur une table de hachage. L'avantage de cela est que pour tester l'appartenance $y \in v_+(x)$ on sera en $O(1)$ avec un dictionnaire alors qu'on sera en $O(d_+(x))$ avec une liste.

I.5.iv Comparaison

opération	Matrice	Listes	Dictionnaire
complexité spatiale	$O(\|S\|^2)$	$O(\|S\| + \|A\|)$	$O(\|S\| + \|A\|)$
arête test	$O(1)$	$O(\|A\|)$	$O(1)$
arête ajout	$O(1)$	$O(1)$	$O(1)$
arête suppression	$O(1)$	$O(\|A\|)$	$O(1)$
sommet ajout	$O(\|S\|^2)$	$O(\|S\|)$	$O(\|S\|)$

opération	Matrice	Listes	Dictionnaire
sommet suppression	$O(\|S\|^2)$	$O(\|S\|)$	$O(\|S\|)$
voisins/degré +	$O(\|S\|)$	$O(1)$	$O(1)$
voisins/degré -	$O(\|S\|)$	$O(\|S\| + \|A\|)$	$O(\|S\|)$

Notons qu'il est possible d'améliorer certaines complexités en utilisant des tableaux dynamiques, notamment les ajouts et suppressions de sommets.

■ **Remarque 1.10** Il semble a priori clair qu'on devrait utiliser des dictionnaires pour manipuler des graphes en voyant ce tableau. Pourtant, on utilisera presque exclusivement des listes d'adjacence. Pourquoi ? En grande partie parce que les opérations rendues efficaces sont rares et ne justifient pas la difficulté accrue de manipulation. En effet, l'opération la plus importante est de pouvoir énumérer les voisins, et une liste chaînée le permet facilement et efficacement.

■

II Graphes non orientés

II.1 Définition et adaptation du vocabulaire

Définition II.1 Un graphe non orienté est un couple $G = (S, A)$ où S est un ensemble fini et A est un ensemble de paires d'éléments de S, c'est-à-dire d'ensembles à deux éléments $\{x, y\}$ où $x, y \in S$.

Les éléments de S sont appelés les **sommets** du graphe G et les éléments de A les **arêtes**. Si $\{x, y\} \in A$, on note $x \smile y$.

■ **Remarque 1.11** Si $G = (S, A)$ est un graphe orienté, on peut en déduire deux graphes non orientés :

- un graphe par défaut $G^- = (S, A^-)$ où

$$\forall x, y \in S, x \smile y \iff (x \to y \wedge y \to x)$$

- un graphe par excès $G^+ = (S, A^+)$ où

$$\forall x, y \in S, x \smile y \iff (x \to y \vee y \to x)$$

On a également A symétrique $\iff G^+ = G^-$. Cela correspond à un graphe où chaque arête est dans les deux sens $x \leftrightarrow y$ et, donc, on peut oublier l'orientation.

De la même manière, on peut associer à un graphe non orienté G son graphe orienté symétrique obtenu en doublant chaque arête. C'est à dire en posant si $x \to y$ si $x \smile y$. On a donc $\forall x, y \in S, x \to y \iff y \to x$ et le graphe est bien symétrique.

■

On reprend directement l'essentiel du vocabulaire des graphes orientés symétriques avec des simplifications :

Définition II.2 Soit $G = (S, A)$ un graphe et $x \in S$.

- On appelle *voisins* de x les éléments de

$$v(x) = \{\ y \in S \mid x \smile y\ \}$$

- On appelle *degré* de x l'entier $d(x) = |v(x)|$.

Théorème II.1 $\sum_{x \in S} d(x) = 2|A|$

On étend directement la notion de **chemin** mais il faut faire attention au fait que **simple** n'a pas le même sens entre un graphe non orienté symétrique et un graphe non orienté. En effet, on $x \to y \to x$ est simple pour un graphe orienté alors que $x \smile y \smile x$ ne l'est pas vu qu'il s'agit de la même arête.

II.2 Connexité

Définition II.3 On définit $\smile^*$ comme étant la clôture réflexive et transitive de $\smile$.

Théorème II.2
- $\smile^*$ est une relation d'équivalence dont les classes sont appelées les **composantes connexes** du graphe.
- $x \smile^* y \iff$ il existe un chemin de x à y.

Définition II.4 Un graphe n'ayant qu'une classe d'équivalence pour $\smile^*$ est dit **connexe**. Cela signifie qu'il existe un chemin entre toute paire de sommets.

II.3 Graphe acyclique connexe

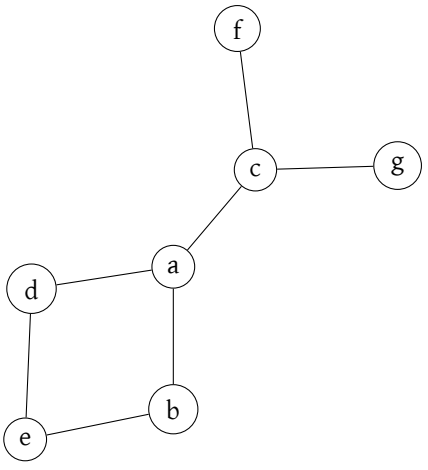
Définition II.5 On appelle **arbre** un graphe acyclique connexe. Quand on a distingué un sommet, on parle d'**arbre enraciné**.

II.4 Graphe biparti

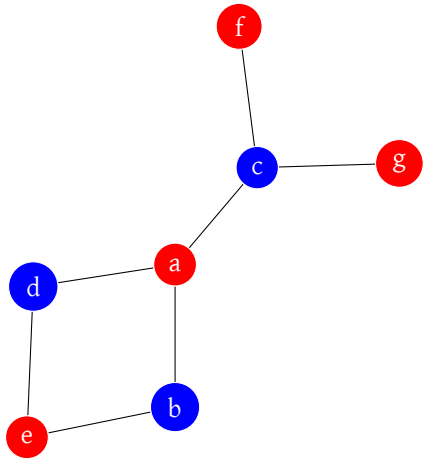
Définition II.6 Soit $G = (S, A)$ un graphe, on dit que le graphe est **biparti** lorsqu'il existe une partition $S = S_1 \cup S_2$ et que

$$\forall \{x, y\} \in A, x \in S_1 \iff y \in S_2$$

Ici, le graphe est biparti avec $S_1 = \{a, e, f, g\}$ et $S_2 = \{b, c, d\}$.



On peut s'en rendre compte en colorant les sommets.



En fait, être biparti est équivalent à pouvoir être coloré en deux couleurs en sorte que deux sommets reliés soient de couleur différente.

III Parcours

ITC- Sup | MP2I- S2

III.1 Principe

La grande majorité des algorithmes sur les graphes consistent à parcourir les sommets de voisins en voisins pour effectuer des traitements. La manière dont on les parcourt pouvant changer selon les différentes applications.

Citons, par exemple, le fait de déterminer les composantes connexes d'un graphe ou de trouver le plus court chemin entre deux sommets.

On va se placer dans le cadre d'un graphe orienté représenté par listes d'adjacence et en supposant que les sommets sont identifiés par leur indice. Dans ce cadre, si on a, en fait, un graphe non orienté, il sera représenté par son graphe orienté symétrique comme on l'a vu plus haut.

On va également considérer qu'on veut effectuer un traitement, ou une visite, pour ces sommets ou les arêtes empruntées.

III.2 Parcours en profondeur récursif

On présente ici une première manière élémentaire de les parcourir en tirant partie de la récursivité :

- on considère une fonction `parcours` et l'appel à `parcours` pour le sommet `x` va effectuer des appels récursifs à `parcours` pour chaque sommet y de $v_+(x)$.

Le problème est qu'on ne veut pas traiter deux fois un sommet et on veut que les appels terminent. Pour cela, on introduit une notion d'état associé à chaque sommet. Un sommet peut-être

- **Inconnu**, cela correspond au fait qu'il n'est pas encore apparu en tant que voisin.
- **Découvert**, il est apparu mais n'a pas encore été traité complètement.
- **Traité** (ou **Visité**), il a non seulement été traité, mais également tous les sommets parcouru grâce à lui.

Pour maintenir cet état dans le code, le plus simple est de considérer un tableau d'entiers `etat` où `etat[i]` donne l'état du sommet `i`.

Définition III.1 Ce parcours est appelé un **parcours en profondeur** récursif. En anglais, on parle de **depth-first search** et on utilise couramment l'acronyme **DFS**.

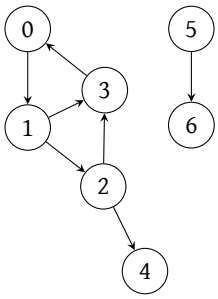
III.2.i Première version

On adapte directement le principe précédent en un programme.

Python
```python
INCONNU = 0
DECOUVERT = 1
TRAITE = 2

def parcours(ladj, x, etat):
    if etat[x] != TRAITE:
        print(x)
        for y in ladj[x]:
            if etat[y] == INCONNU:
                etat[y] = DECOUVERT
                parcours(ladj, y, etat)
        etat[x] = TRAITE

def lance_parcours(ladj, x):
    # État initial inconnu pour tous
    etat = [ INCONNU ] * len(ladj)
    etat[x] = DECOUVERT
    parcours(ladj, x, etat)
```

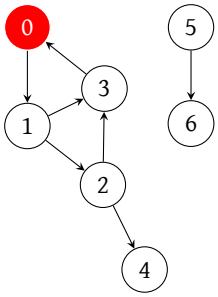
Ici, on se contente d'afficher les sommets rencontrés.

■ **Exemple 1.5** On va réaliser un parcours en profondeur du graphe suivant en partant du sommet 0 :

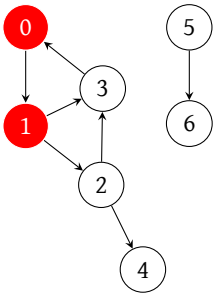


On indique les sommets inconnus en blanc, les sommets découvert en rouge et les sommets traités en vert.

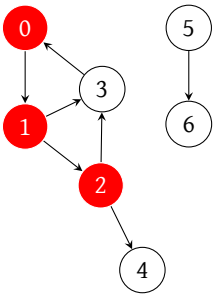
On affiche 0 et on appelle relance le parcours du voisin 1.



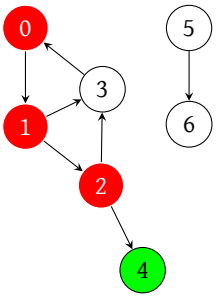
On affiche 1 et on relance le parcours du voisin 2.



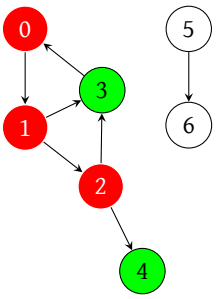
On affiche 3 et on appelle relance le parcours du voisin 4.



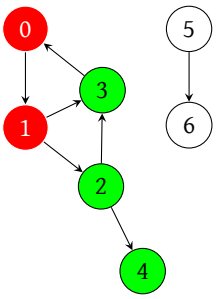
On affiche 4 et comme il n'a pas de voisins non traités, on traite 4 et on sort de l'appel.



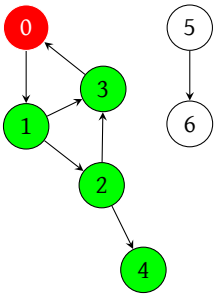
On revient à 2 et on relance le parcours du voisin 3. On affiche 3, et on le traite car tous ses voisins sont connus.



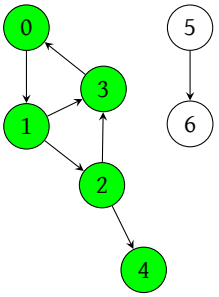
On revient à 2 et cette fois-ci il n'y a plus de voisins, on peut marquer 2 comme étant traité.



On revient à 1 et on peut marquer 1 comme étant traité.



On revient à 0 et on peut marquer 0 comme étant traité.



On a donc une vague de descente dans le graphe qui marque les sommets comme découverts et ensuite on remonte la pile d'appels récursifs en marquant les sommets comme étant traités.

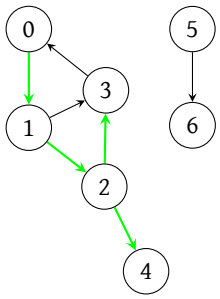
Les sommets sont ici affichés dans l'ordre 0 1 2 4 3. On remarque que les sommets 5 et 6 sont inaccessibles, ils restent ainsi inconnus tout le long du parcours.

■

III.2.ii Arbre de parcours

On peut noter les arêtes empruntées lors du parcours précédents pour découvrir un nouveau sommet, et donc relancer un appel récursif. On obtient alors une structure arborescente qui est appelé l'arbre du parcours en profondeur.

■ **Exemple 1.6** Dans le parcours précédent, on obtient l'arbre suivant :



■

Pour obtenir cet arbre, on va construire un tableau `parent` pendant le parcours. Lorsqu'on emprunte une arête $x \to y$ pour découvrir y, on note $parent[y] = x$. Par défaut, $parent[y]$ est indéfini (`None` ou une valeur d'indice invalide comme -1).

Théorème III.1 Si $parent[y]$ est défini à l'issue du parcours depuis x, alors il existe un chemin $\varphi : x \rightsquigarrow y$ où les arêtes $s \to s'$ empruntées lors du chemin vérifient toutes $parent[s'] = s$.

Comme $parent[y]$ est défini pour tous les sommets découverts et que les sommets découverts finissent tous par être traités, on peut en déduire directement le théorème suivant :

Théorème III.2 A l'issue du parcours depuis x, les sommets traités sont les sommets accessibles depuis x, c'est-à-dire les éléments de

$$\{\, y \in S \mid x \rightsquigarrow y \,\}$$

On présente le calcul de ces chemins dans le programme suivant :

Python

```python
INCONNU = 0
DECOUVERT = 1
TRAITE = 2

def parcours(ladj, x, etat, parent):
    if etat[x] != TRAITE:
        print(x)
        for y in ladj[x]:
            if etat[y] == INCONNU:
                parent[y] = x
                etat[y] = DECOUVERT
                parcours(ladj, y, etat)
        etat[x] = TRAITE

def arbre_parcours(ladj, x):
    etat = [ INCONNU ] * len(ladj)
    parent = [ None ] * len(ladj)
    etat[x] = DECOUVERT
    parcours(ladj, x, etat, parent)
    return parent

def chemin(parent, y):
    # Renvoie le chemin x -> ... -> y
    # en sens inverse
```

```
    p = [ y ]
    while parent[y] != None:
        y = parent[y]
        p.append(y)
    return p
```

III.2.iii Composantes connexes

On peut déduire directement du théorème précédent, le corollaire suivant dans le cas non orienté :

Théorème III.3 Dans un graphe non orienté, les sommets traités depuis un parcours en profondeur issu d'un sommet sont exactement les sommets de sa composante connexe.

On en déduit alors un algorithme pour obtenir les composantes connexes d'un graphe non orienté :

Algorithme - COMPOSANTES CONNEXES
- Entrées :
  Un graphe non orienté $G = (S, A)$
- ⋆ Tant qu'il y a des sommets non traités
  - On choisit un sommet non traité x
  - On effectue un parcours récursif depuis x
  - Tous les sommets traités par ce parcours forment une composante connexe

On va en déduire le programme suivant :

Python
```
INCONNU = 0
DECOUVERT = 1
TRAITE = 2

def parcours(ladj, x, etat, composante):
    if etat[x] != TRAITE:
        composante.append(x)
        for y in ladj[x]:
            if etat[y] == INCONNU:
                etat[y] = DECOUVERT
                parcours(ladj, y, etat)
        etat[x] = TRAITE

def composantes_connexes(ladj):
    composantes = []
    etat = [ INCONNU ] * len(ladj)
    for i in range(len(ladj)):
        if etat[i] == INCONNU:
            composante = [ ]
            parcours(ladj, x, etat, composante)
            composantes.append( composante )
    return composantes
```

III.2.iv Détection de cycles

Si on est en train de traiter le sommet x et qu'on rencontre une arête $x \to y$ où y est découvert mais non traité, c'est qu'on est dans l'appel récursif de y et donc que x est un de ses

descendants : $y \rightsquigarrow x$ en rajoutant la nouvelle arête $y \rightsquigarrow x \to x$ on en déduit un cycle dans le cas d'un graphe orienté.

Pour un graphe non orienté représenté par un graphe orienté symétrique, il faut faire attention à ne pas prendre un aller-retour $x \to y \to x$ pour un cyle. On demande donc à avoir la condition :

Python
```
etat[y] == DECOUVERT and parent[x] != y
```

On en déduit le programme suivant :

Python
```
INCONNU = 0
DECOUVERT = 1
TRAITE = 2

def parcours(ladj, x, etat, parent, est_oriente):
    if etat[x] != TRAITE:

        for y in ladj[x]:
            if etat[y] == INCONNU:
                parent[y] = x
                etat[y] = DECOUVERT
                cycle = parcours(ladj, y, etat)
                if cycle != None:
                    return cycle
            elif etat[y] == TRAITE and (est_oriente or parent[x] != y):
                return chemin(parent, y)

        etat[x] = TRAITE
    return None

def detecte_cycle(ladj, x):
    etat = [ INCONNU ] * len(ladj)
    parent = [ None ] * len(ladj)
    etat[x] = DECOUVERT
    cycle = parcours(ladj, x, etat, parent)
    if cycle != None:
        print('Cyclique : ', cycle)
    else:
        print('Acyclique')
```

III.2.v Classification des arêtes

MP2I- S2 | OI- Spé

III.2.vi Temps d'entrée et de sortie

MP2I- S2 | OI- Spé

III.3 Parcours quelconque

On va considérer ici que l'on dispose d'une structure de donnée `sac` dans laquelle on peut placer des éléments et en sortir. On considère aussi qu'on a une moyen de marquer les sommets. Par défaut, ils sont non marqués.

On considère alors l'algorithme suivant decrit en pseudo-code et auquel on fera référence comme étant le parcours **quelconque** dans la suite :

Algorithme - PARCOURS QUELCONQUE
- Entrées :
  - ⋆ Un graphe $G = (S, A)$
  - ⋆ Un somme $s \in S$ source du parcours
- ⋆ On crée un sac ne contenant que s
- ⋆ Tant que le sac est non vide
  - ○ On tire un sommet x du sac
  - ○ Si x n'est pas marqué
    - □ On marque x
    - □ pour chaque $x \to y \in A$
      - On ajoute y dans le sac

On remarque que les somets marqués sont exactement les sommets accessibles depuis s, on retrouve ainsi le théorème vu pour le parcours récursif.

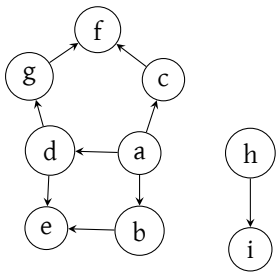
En fait, le parcours quelconque ne permet pas de déduire plus d'information que cela, mais dans un graphe non orienté c'est déjà suffisant pour en déduire les composantes connexes.

La question qui se pose alors est celle de la stratégie d'ajout/tirage dans le sac. Selon ce que l'on considère, l'ordre de visite des sommets va changer. On va considérer trois stratégies :

- LIFO : Last In First Out, le prochain sommet tiré est le dernier ajouté
- FIFO : First In First Out, le prochain sommet tiré est le sommet le plus anciennement ajouté
- aléatoire : on tire aléatoirement et uniformément un sommet du sac

Définition III.2 Un sac avec une stratégie LIFO est appelé une **pile**.
Un sac avec une stratégie FIFO est appelé une **file**.

■ **Exemple 1.7** Si on considère le graphe suivant :

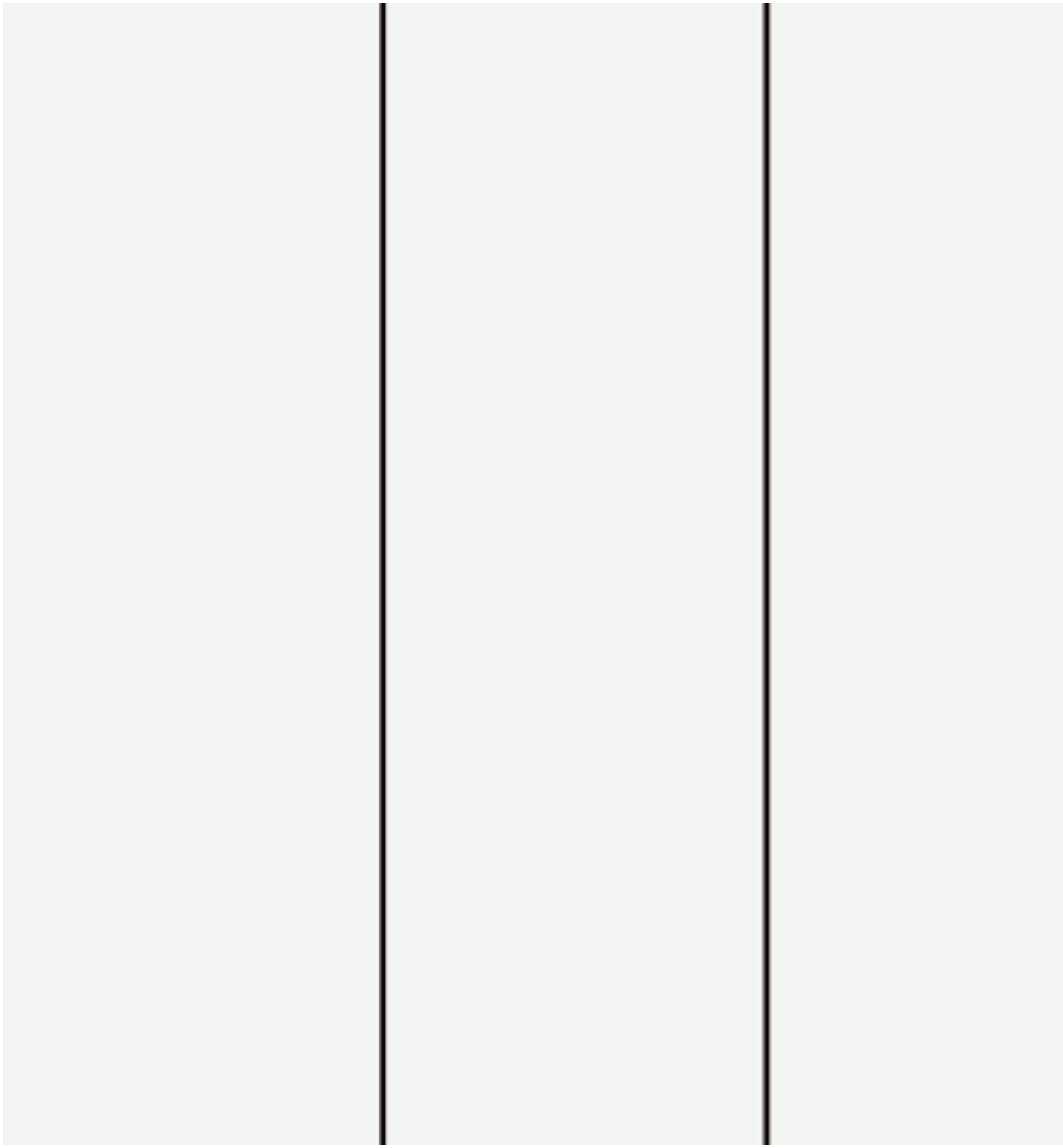


En effectuant un parcours quelconque issu de a :

- en stratégie LIFO, on va voir les sommets dans l'ordre a, b, e, c, f, d, g.
- en stratégie FIFO, on va voir les sommets dans l'ordre a, b, c, d, e, f, g
- en stratégie aléatoire, on pourrait avoir a, b, c, e, d, f, g

■

Une manière de *voir* ces trois différentes stratégies est de considérer une grille et de colorer les sommets marqués avec une couleur changeant à chaque marquage. On peut voir dans l'ordre LIFO, FIFO, aléatoire l'effet de ces trois stratégies en partant du même point sur l'animation suivante :



■ **Remarque 1.12** La stratégie LIFO semble *bornée* : elle choisit un cap et elle s'y tient tant que c'est possible. La stratégie FIFO va elle s'étendre comme une onde partant du point de départ. De manière assez étonnante, la stratégie aléatoire semble être une bonne approximation de la stratégie FIFO, ce qui montre le caractère exceptionnel du LIFO.

■

Le parcours quelconque en stratégie FIFO s'appelle un *parcours en largeur*. Comme on va le voir, le parcours avec stratégie LIFO est exactement le *parcours en profondeur* précédent.

III.3.i Implémentation en `Python`

Pour implémenter un sac, on va d'abord utiliser le type `list` de `Python` :

- Pour créer un sac vide, on utilise la valeur `[]`

- Pour ajouter un élément `x` dans le sac `s` : `s.append(x)`
- Pour retirer le dernier élément ajouté, on peut utiliser directement `s.pop()`
- Pour retirer l'élément le plus anciennement ajouté, il faut remplacer `s` par `s[1:]` et renvoyer la valeur `s[0]`.
- C'est anecdotique, mais pour retirer un élement au hasard, il faut choisir un indice `i` et remplacer `s` par `s[:i] + s[i+1:]`.

Si on peut supposer que les trois premières opérations sont en temps constant, les deux dernières sont linéaire en le nombre de sommets dans le sac. Ce qui est assez coûteux.

En effet, le parcours quelconque va accéder en pire cas à chaque sommet et à chaque liste d'adjacence et ajouter chaque sommet dans le sac. Donc, une complexité en $O(|S| + |A| + |S|f(|S|))$ où $f(n)$ est la complexité du retrait dans un sac de taille n.

On obtient donc $O(|S|+|A|)$ en LIFO mais $O(|S|^2)$ en FIFO car $|A| = O(|S|^2)$ vu que $A \subset S^2$.

En `Python`, il est possible d'avoir une file (`queue` en anglais) permettant de réaliser le retrait FIFO en $O(1)$:

- On importe le type `deque` : `from collections import deque`
- On crée une file vide avec `s = deque()`
- On ajoute un élément `x` avec `s.append(x)`
- Mais on dispose d'un fonction efficace `s.popleft()` pour retirer le premier élément.

En fait, on dispose également d'un `appendleft` et du `pop`. Toutes ces opérations étant en $O(1)$.

■ **Remarque 1.13** OI- Sup

La structure de donnée `deque` est une liste doublement chaînée. Elle permet de remonter dans la chaîne d'un maillon vers le maillon dont il est le suivant.

■

III.4 Parcours en largeur

On va vu que le parcours quelconque en FIFO s'appelait le parcours en largeur. Il permet de parcourir les sommets d'un graphe en rayonnant à partir du sommet source. En effet on traite d'abord :

- le sommet source src
- les voisins de src
- les voisins des voisins de src
- ...

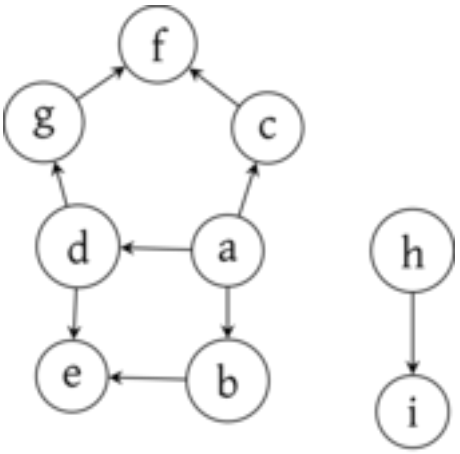
Pour pouvoir parler de chemins, il est nécessaire d'introduire une notion de parenté. Pour cela, on modifie le parcours *quelconque* pour en déduire un parcours quelconque qui a l'air a priori d'être *optimisé*.

Algorithme - PARCOURS QUELCONQUE *OPTIMISÉ*

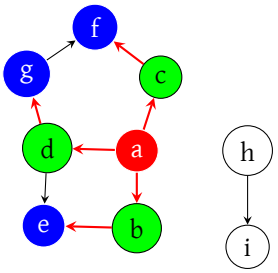
- Entrées :
  - ⋆ Un graphe $G = (S, A)$
  - ⋆ Un somme $s \in S$ source du parcours
- ⋆ On crée un sac ne contenant que s
- ⋆ Tant que le sac est non vide
  - ○ On tire un sommet x du sac
    - □ pour chaque $x \to y \in A$
      - Si y est non marqué
        - ⋆ On indique que $parent[y] = x$

⋆ On ajoute y dans le sac
⋆ On marque y

■ **Exemple 1.8** On va reprendre l'exemple précédent



On obtient avec un parcours en largeur le schéma suivant :



où on a noté les arêtes de parenté ainsi que d'une même couleur les éléments à égale distance de a dans le parcours en largeur.

On remarque qu'on a obtenu le chemin $a \to c \to f$ là où un parcours en profondeur aurait pu donner le chemin plus long $a \to d \to g \to f$.

■

Comme semble le suggérer l'exemple précédent, on peut démontrer le théorème suivant :

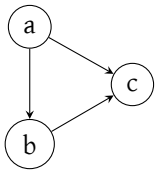
Théorème III.4 Les chemins donnés par un parcours en largeur depuis x sont de plus petite longueur :
Si $\varphi : x \rightsquigarrow y$ est donné par le parcours, alors $\forall \psi : x \rightsquigarrow y$, $|\psi| \geq |\varphi|$.

III.4.i Preuve

OI- Spé | MP2I- S2

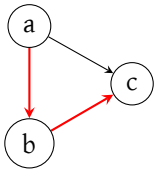
III.5 Pseudo-parcours en profondeur

On peut se poser la question de la nature du parcours quelconque optimisé effectué avec une stratégie LIFO. Naïvement, on peut penser qu'il s'agit d'un parcours en profondeur. Cependant, si on considère le graphe :

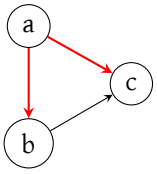


On va avoir les deux arbres différents suivants :

Avec un parcours en profondeur :



Avec un pseudo-parcours en profondeur :



On perd une propriété fondamentale du parcours en profondeur qui est de qu'un parent ne peut empecher un de ses enfants de découvrir un sommet. Ici, dans le pseudo-parours, a bloque la découverte de c par b.

Exercice 1.1 Montrer qu'il n'existe aucune stratégie permettant au parcours quelconque *optimisé* d'être un parcours en profondeur.

■

IV Travaux Pratiques

IV.1 Parcours de graphes en C

IV.1.i Représentation

On va considérer le type suivant pour les graphes qui suppose qu'on n'aura jamais plus que MAXV sommets. On utilise ici une constante avec l'alias #define : partout où on écrit MAXV, il sera remplacé par la valeur 100.

```c
#define MAXV 100 /* nombre maximum de sommets */

struct edgenode {
    int y; // le voisin
    struct edgenode *next; // la suite de la liste
};
typedef struct edgenode edgenode;

struct graph {
    edgenode *edges[MAXV]; // tableau de listes d'adjacence
    int degree[MAXV]; // le degré de chaque sommet
    int nvertices;
    int nedges;
    bool directed; // indique si le graphe est orienté
};
typedef struct graph graph;
```

Question IV.1 Écrire une fonction

```c
void initialize_graph(graph *g, bool directed);
```

qui initialise le graphe g passé par pointeur comme étant le graphe vide, dirigé ou non selon la valeur de directed.

Attention vous êtes libres d'initialiser les listes d'adjancence à la liste vide ou de considérer que sera fait dans la fonction read_graph ci-dessous.

■

Démonstration.

```c
void initialize_graph(graph *g, bool directed)
{
    g->directed = directed;
    g->nvertices = 0;
    g->nedges = 0;
    for (int i = 0; i < MAXV; i++)
    {
        g->edges[i] = NULL;
        g->degree[i] = 0;
    }
}
```

■

Question IV.2 Écrire une fonction

```c
void insert_edge(graph *g, int x, int y, bool directed)
```

qui insère une arête de x à y en considérant g comme orienté ou non selon `directed` **et donc en ignorant** `g->directed`.

Attention il faudra prendre garde au fait que g soit orienté ou non. Dans le second cas, on représenté les arêtes comme dans un graphe orienté symétrique, donc il faut en ajouter deux. C'est la raison pour laquelle on utilise le paramètre `directed` plutôt que de consulter `g->directed`.

■

Démonstration.

```c
void insert_edge(graph *g, int x, int y, bool directed)
{
    edgenode *edge = malloc(sizeof(edgenode));
    edge->y = y;
    edge->next = g->edges[x];
    g->edges[x] = edge;
    if (!directed)
        insert_edge(g, y, x, true);
}
```

■

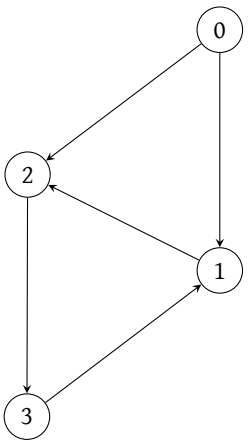
Pour travailler sur des graphes, on va écrire une fonction permettant de lire un fichier contenant le graphe sous le format suivant :

- première ligne contenant trois entiers, le nombre de sommets n, le nombre d'arêtes p et 0 ou 1 selon que le graphe soit non orienté ou orienté
- ensuite p lignes contenant deux entiers i et j et indiquant qu'il y a une arête de i vers j

Par exemple :

```c
4 5 1
0 1
0 2
1 2
2 3
3 1
```

sera représenté par le graphe :



■ **Remarque 1.14** On va utiliser ici l'entrée standard, c'est-à-dire l'entrée de l'utilisateur depuis le terminal. Cependant, il est possible de rédiriger cette entrée depuis un fichier. En effet, si on appelle

```
./monprogramme < monfichier
```

Alors l'entrée standard sera le contenu de `monfichier`. Cela permet de ne pas avoir à se préoccuper d'ouvrir de fichier et d'utiliser directement `scanf`.

Rappel `scanf("%d %d", &x, &y);` va lire une ligne avec deux entiers et placer la valeur du premier dans `x` et la valeur du second dans `y`.

■

Question IV.3 Écrire une fonction

```
void read_graph(graph *g)
```

qui lit un graphe depuis l'entrée standard et le place dans `g` **après l'avoir initialisé**.

■

Démonstration.

```c
void read_graph(graph *g)
{
    int directed;
    initialize_graph(g, false);
    scanf("%d %d %d", &g->nvertices, &g->nedges, &directed);
    g->directed = directed == 1;
    for (int i = 0; i < g->nedges; i++)
    {
        int x, y;
        scanf("%d %d", &x, &y);
        insert_edge(g, x, y, g->directed);
    }
}
```

■

Question IV.4 Écrire une fonction

```
void free_edges(graph *g)
```

qui libère les listes d'adjacence.

■

Démonstration.

```c
void free_edges(graph *g)
{
    for (int i = 0; i < g->nvertices; i++)
    {
        edgenode *n = g->edges[i];
        while(n != NULL)
        {
```

```
            edgenode *temp = n->next;
            free(n);
            n = temp;
        }
    }
}
```

■

IV.1.ii Parcours en profondeur récursif

On va modifier la structure de graphe et rajouter trois nouveaux champs :

```c
bool discovered[MAXV]; // Quels sommets sont connus
bool processed[MAXV]; // Quels sommets sont traités
int parent[MAXV]; // parent[x] est le père de x dans le parcours
                  // s'il n'y en a pas, c'est -1
```

Question IV.5 Écrire une fonction

```c
void initialize_search(graph *g);
```

qui initialise ces tableaux pour commencer une nouvelle recherche. ■

Démonstration.

```c
void initialize_search(graph *g)
{
    for (int i = 0; i < g->nvertices; i++)
    {
        g->discovered[i] = false;
        g->processed[i] = false;
        g->parent[i] = -1;
    }
}
```

■

On va définir trois fonctions qui seront appelées lors d'un parcours et qu'on pourra redéfinir.

```c
void process_vertex_early(graph *g, int v)
{
    printf("processing vertex %d\n", v);
}

void process_vertex_late(graph *g, int v)
{
}

void process_edge(graph *g, int x, int y)
{
    printf("processed edge %d --> %d\n", x, y);
}
```

Question IV.6 Écrire une fonction récursive

```c
void dfs(graph *g, int x)
```

qui effectue un parcours en profondeur en partant du sommet `x`.

Cette fonction va appeler `process_vertex_early` au début du traitement de `x`, `process_vertex_la` à la fin et `process_edge` pour chaque arête rencontrée.

Attention si le graphe est non orienté et qu'on a rencontré `x -> y` dans cet ordre, on ne fera pas de traitement dans le sens `y -> x`.

■

Démonstration.

```c
void dfs(graph *g, int x)
{
    // important uniquement pour le premier
    g->discovered[x] = true;
    process_vertex_early(g, x);

    edgenode *n = g->edges[x];
    while(n != NULL)
    {
        process_edge(g, x, n->y);
        if (!g->discovered[n->y])
        {
            g->discovered[n->y] = true;
            g->parent[n->y] = x;
            g->color[n->y] = !g->color[x];
            dfs(g, n->y);
        }
        n = n->next;
    }

    process_vertex_late(g, x);
    g->processed[x] = true;
}
```

■

Question IV.7 Comme on l'a démontré, le parcours dans un graphe non orienté permet de traiter exactement la composante connexe du sommet de départ.

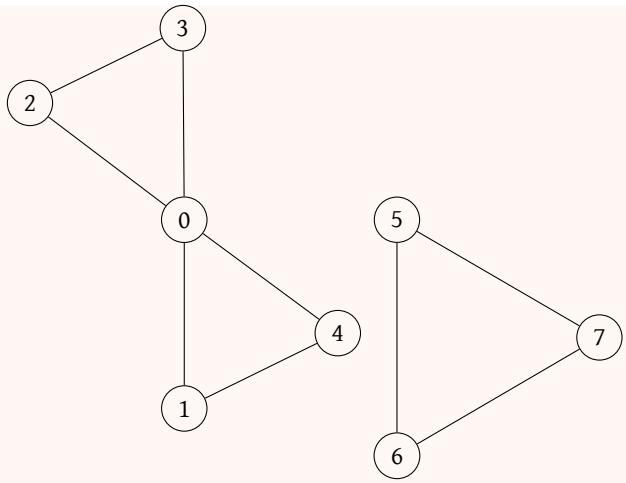
Écrire une fonction

```c
void connected_components(graph *g)
```

qui affiche les composantes connexes sous la forme :

```c
Component 1: 0 1 2 3 4
Component 2: 5 6 7
```

pour le graphe



Il sera nécessaire de modifier les fonctions `process_*`.

Démonstration.

On remplace les fonctions par :

```c
void process_vertex_early(graph *g, int v)
{
    printf(" %d", v);
}

void process_vertex_late(graph *g, int v)
{
}

void process_edge(graph *g, int x, int y)
{
}
```

et on définit :

```c
void connected_components(graph *g)
{
    initialize_search(g);
    int comp = 0;
    for(int i = 0; i < g->nvertices; i++)
    {
        if(!g->processed[i])
        {
            comp = comp+1;
            printf("Component %d:", comp);
            dfs(g, i);
            printf("\n");
        }
    }
}
```

On relance ainsi le parcours en repartant d'un sommet non traité.

Question IV.8 Adaptez le parcours pour détecter des cycles. Quand vous détectez un cycle, affichez les sommets qui le compose. Dans le graphe précédent il faut afficher :

```
Cycle : 0 2 3
Cycle : 0 1 4
Cycle : 5 6 7
```

■

Démonstration.

Il suffit de changer `process_edge` dans le parcours de toutes les composantes pour détecter les arêtes arrières :

```c
void process_edge(graph *g, int x, int y)
{
    // Teste si on a une arête arrière
    if (g->discovered[y]
            && !g->processed[y]
            && (g->directed || g->parent[x] != y))
    {
        printf("Cycle : %d", y);
        int cur = x;
        while(cur != y & cur > -1)
        {
            printf(" %d", cur);
            cur = g->parent[cur];
        }
        printf("\n");
    }
}
```

■

Question IV.9 A l'aide d'un parcours, déterminez si un graphe non orienté est biparti.

■

Démonstration.

On rajoute des champs

```c
bool colors[MAXV];
bool bipartite;
```

dans `graph` et on commence en donnant la couleur `true` au sommet de départ du DFS. Ensuite, il suffit de colorer avec la couleur différente de x quand on voit une arête $x \to y$ puis de vérifier à chaque arête qu'elle relie des sommets de couleur différente.

```c
```

■

IV.1.iii Temps et classification des arêtes

On rajoute à la structure `graph` deux tableaux et un entier :

```c
int time; // l'horloge
int entry_time[MAXV];
int exit_time[MAXV];
```

Question IV.10 Rajouter à `dfs` le maintien de l'horloge et des temps d'entrée et de sortie.

Démonstration.

Il suffit de modifier les traitements de sommets :

```c
void process_vertex_early(graph *g, int v)
{
    g->entry_time[v] = g->time;
    g->time = g->time + 1;
    printf("%d\n", v);
}

void process_vertex_late(graph *g, int v)
{
    g->exit_time[v] = g->time;
    g->time = g->time + 1;
}
```

■

On définit des constantes :

```c
#define TREE 0
#define BACK 1
#define FORWARD 2
#define CROSS 3
```

Question IV.11 Écrire une fonction :

```c
int edge_classification(graph *g, int x, int y)
```

qui pourra être appelée dans `process_edge` pour déterminer la classe d'une arête selon les constantes précédentes. On renverra -1 si l'arête ne peut être déterminée (est-ce possible ?).

Démonstration.

```c
int edge_classification(graph *g, int x, int y)
{
    if (!g->discovered[y])
        return TREE;
    if (!g->processed[y])
        return BACK;
    if (g->entry_time[x] < g->entry_time[y])
        return FORWARD;
    return CROSS;
}
```

■

IV.1.iv Parcours avec une structure

On va réutiliser ici des implémentations de files et de piles dans un tableau de taille fixe.

Question IV.12 Compléter les implémentations en se basant sur ce qui a déjà été fait dans les TP précédents.

```c
struct queue {
    int elts[MAXV];
    int front;
    int back;
};
typedef struct queue queue;

struct stack {
    int elts[MAXV];
    int back;
};
typedef struct stack stack;

void init_stack(stack *s)
{
    // initialise la pile
}

int pop(stack *s)
{
    // depile un élément
}

void push(stack *s, int x)
{
    // empile x sur la pile s
}

void init_queue(queue *s)
{
    // initialise la file
}

int dequeue(queue *s)
{
    // defile un élément
}

void enqueue(queue *s, int x)
{
    // enfile x sur la file s
}
```

■

Démonstration.

```c
void init_stack(stack *s)
{
    s->back = 0;
}

int pop(stack *s)
{
    assert(s->back > 0);
    s->back = s->back-1;
    return s->elts[s->back];
}

void push(stack *s, int x)
{
    assert(s->back < MAXV-1);
    s->elts[s->back] = x;
    s->back = s->back+1;
}

bool empty_stack(stack *s)
{
    return s->back == 0;
}

void init_queue(queue *s)
{
    s->front = 0;
    s->back = 0;
}

int dequeue(queue *s)
{
    int x = s->elts[s->front];
    s->front = (s->front + 1) % MAXV;
    return x;
}

void enqueue(queue *s, int x)
{
    s->elts[s->back] = x;
    s->back = (s->back + 1) % MAXV;
}

bool empty_queue(queue *s)
{
    return s->front == s->back;
}
```

■

Question IV.13 Écrire des fonctions

```c
void dfs(graph *g, int src);
void bfs(graph *g, int src);
```

effectuant un parcours en utilisant pour les sommets à visiter une pile ou une file. On ne pourra pas maintenir les temps de sortie ici car cela n'a plus vraiment de sens.

■

Démonstration.

```c
void dfs(graph *g, int src)
{
    stack s;
    init_stack(&s);
    push(&s, src);
    while(!empty_stack(&s))
    {
        int x = pop(&s);
        if (!g->processed[x])
        {
            g->processed[x] = true;
            process_vertex_early(g, x);

            edgenode *n = g->edges[x];
            while(n != NULL)
            {
                push(&s, n->y);
                n = n->next;
            }
        }
    }
}

void bfs(graph *g, int src)
{
    queue s;
    init_queue(&s);
    enqueue(&s, src);
    while(!empty_queue(&s))
    {
        int x = dequeue(&s);
        if (!g->processed[x])
        {
            g->processed[x] = true;
            process_vertex_early(g, x);

            edgenode *n = g->edges[x];
            while(n != NULL)
            {
                enqueue(&s, n->y);
                n = n->next;
            }
        }
    }
}
```

■

Question IV.14 A l'aide d'un parcours en largeur, le `bfs`, déterminez étant donné un sommet `x` et un sommet `y` de sa composante connexe, le plus court chemin de `x` vers `y`. On l'affichera sous la forme `0 -- 3 -- 1 -- 5`.

■

Démonstration.

On rajoute un champ comme vu dans le cours :

```c
int d[MAXV];
```

on marque `d[x] = 0` et on utilise le any-search modifié pour tenir compte de la parenté :

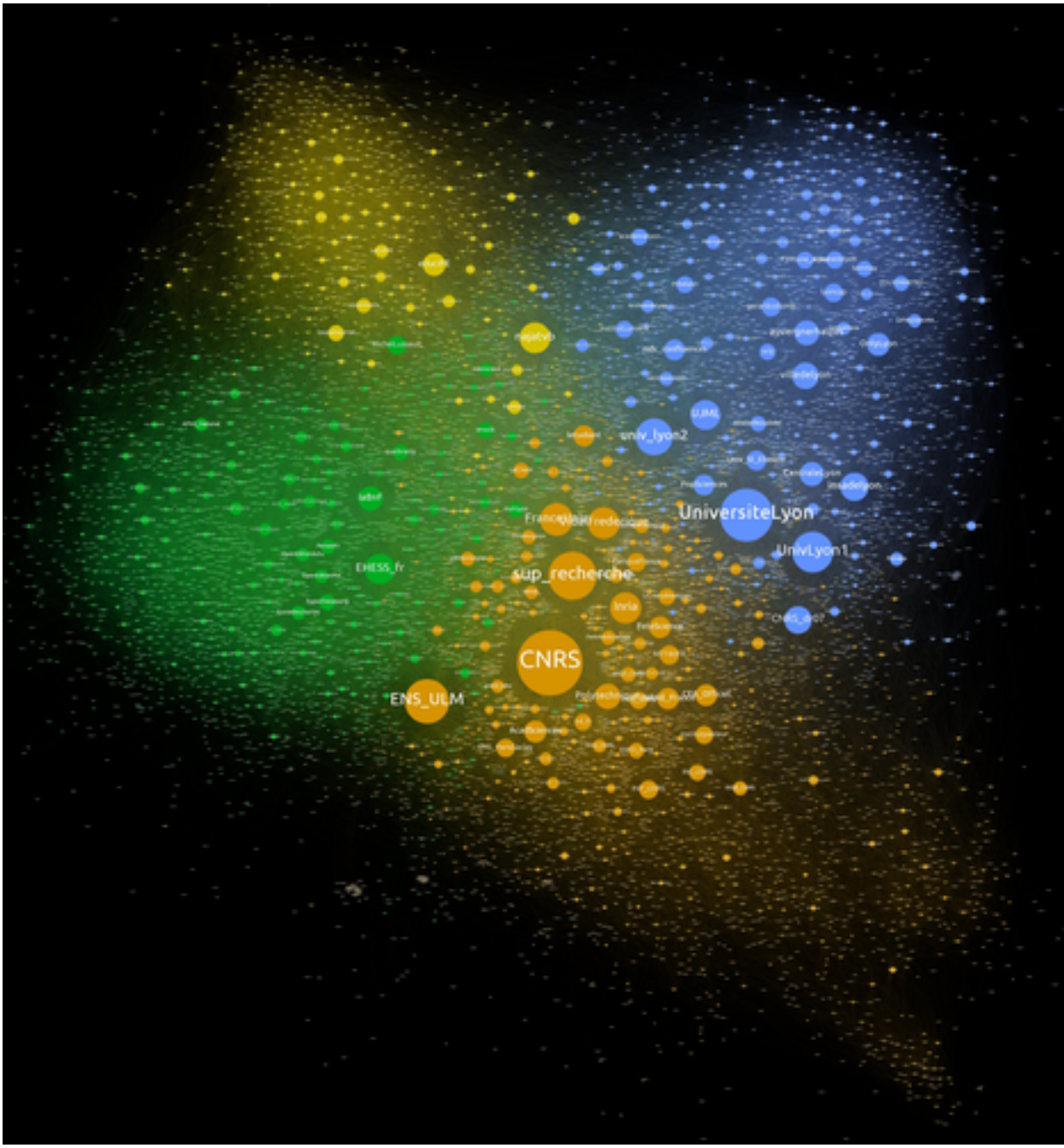
```c

```

■

IV.2 Étude d'un graphe issu d'un réseau social

Dans ce TP on va étudier le graphe **orienté** des followers du compte Twitter `@ENSdeLyon`.

Une représentation graphique de ce graphe est donné dans l'image suivante :



Les sommets sont les comptes donnés par leur identifiant (le `@identifiant` de Twitter) et une arête $x \to y$ indique que le compte x est abonné au compte y.

Ce graphe est assez conséquent : il comporte 8418 sommets et 305288 arêtes. Il nous permettra ainsi d'étudier en pratique la complexité des différents algorithmes étudiés. On va commencer par lire ce graphe depuis un fichier, ensuite, on en déduira différents graphes associés (sous-graphes, symétrisés par excès ou par défaut...) sur lesquels on pourra appliquer les algorithmes demandés. Une table de résultat est fourni en fin de TP pour vérifier vos résultats.

IV.2.i Définition et lecture du graphe

Le graphe est donné dans le fichier ENSdeLyon.graph. Il s'agit d'un fichier texte ayant la structure suivante :

- un entier `n_sommets` sur une ligne
- un entiers `n_aretes` sur une ligne
- `n_sommets` lignes contenant une chaîne de caractère représentant l'identifiant d'un sommet
- `n_aretes` couple de lignes comportant sur la première un entier `src` et sur la seconde un entier `tgt` indiquant une arête `src -> tgt`.

Question IV.15 Expliquer pourquoi cela ne semble pas être une bonne idée de représenter ce graphe par une matrice d'adjacence. ■

On va utiliser le type suivant permettant de représenter le graphe par listes d'adjacence :

```OCaml
type graphe = {
    sommets : string array;
    aretes : int list array
}
```

Pour lire le graphe depuis le fichier, le plus simple est de le rediriger sur l'entrée standard (*Rappel* `./monprogramme < monfichier`) et d'utiliser les deux fonctions suivantes :

- `read_int : unit -> int` lit une ligne composée d'un entier et renvoie sa valeur.
- `read_line : unit -> string` lit une ligne et la renvoie sans le caractère de saut de ligne, c'est-à-dire, sans le `'\n'`.

Alternativement, on peut lire le graphe depuis un fichier avec :

- `open_in : string -> in_channel` qui crée un descripteur de fichier en lecture pour le nom de fichier passé en paramètre
- `input_line : in_channel -> string` qui lit une ligne dans le descripteur et la renvoie sans le saut de ligne
- `int_of_string : string -> int` qui convertit une chaîne contenant un entier en entier.

Question IV.16 Écrire une fonction `read_graphe : unit -> graphe` qui lit un graphe dans le format précédent sur l'entrée standard et tester que vous arrivez à lire le fichier.

Un graphe minimaliste de 3 sommets et 4 arêtes est donné dans le fichier test.graph afin de vous permettre de tester votre fonction. ■

Démonstration.

```OCaml
let read_graphe () =
    let nb_sommets = read_int () in
    let nb_aretes = read_int () in

    let sommets = Array.init nb_sommets
        (fun _ -> read_line ()) in

    let aretes = Array.make nb_sommets [] in

    for _ = 0 to nb_aretes - 1 do
```

```
        let src = read_int () in
        let tgt = read_int () in

        aretes.(src) <- tgt :: aretes.(src)
    done;
    { sommets=sommets; aretes=aretes }
```

■

Si $G = (S, A)$ est un graphe dont les sommets sont énumérées $S = \{s_0, s_1, \ldots, s_{n-1}\}$, on note, pour $p \leq n$, G_p le sous-graphe induit par $\{s_0, \ldots, s_{p-1}\}$.

Question IV.17 Écrire une fonction `restriction : graphe -> int -> graphe` tel que `restriction g p` où `g` est la représentation d'un graphe G renvoie la représentation du graphe G_p.

■

Démonstration.

On adopte une approche proche de la fonction précédente : on itère sur les arêtes du graphe initial et on ne sélectionne que celles qui sont compatibles avec la restriction.

OCaml
```
let restriction g p =
    let sommets = Array.sub g.sommets 0 p in
    let aretes = Array.make p [] in
    for i = 0 to p - 1 do
        List.iter (fun j ->
            if j < p then aretes.(i) <- j :: aretes.(i))
            g.aretes.(i)
    done;
    { sommets = sommets; aretes = aretes }
```

Notons ici qu'on aurait pu avoir une approche plus fonctionnelle pour sélectionner les bonnes arêtes :

OCaml
```
let restriction g p =
    let sommets = Array.sub g.sommets 0 p in
    let aretes = Array.map (List.filter ((>)p))
        (Array.sub g.aretes 0 p) in
    { sommets = sommets; aretes = aretes }
```

■

Si $G = (S, A)$ est un graphe orienté, on a vu au paragraphe Graphes non orientés les graphes non orientés par défaut et par excès, G^- et G^+ qui lui sont associés.

Question IV.18 Écrire des fonctions `par_defaut : graphe -> graphe` et `par_exces : graphe -> graphe` qui, étant donné un graphe G, renvoie les graphes G^- et G^+ représentés en tant que graphes orientés symétriques.

■

Démonstration.

On adopte encore l'approche de création par remplissage.

```OCaml
let defaut g =
    let n = Array.length g.sommets in
    let sommets = Array.copy g.sommets in
    let aretes = Array.make n [] in
    for i = 0 to n - 1 do
        List.iter (fun j ->
            if List.mem i g.aretes.(j)
            then aretes.(i) <- j :: aretes.(i))
            g.aretes.(i)
    done;
    { sommets = sommets; aretes = aretes }

let exces g =
    let n = Array.length g.sommets in
    let sommets = Array.copy g.sommets in
    (* on recopie les arêtes existantes *)
    let aretes = Array.copy g.aretes in
    for i = 0 to n - 1 do
        List.iter (fun j ->
            (* on rajoute les retours absents *)
            if not (List.mem i g.aretes.(j))
            then aretes.(j) <- i :: aretes.(j))
            g.aretes.(i)
    done;
    { sommets = sommets; aretes = aretes }
```

■

Si $G = (S, A)$ est un graphe orienté, on note $rev(G) = (S, A')$ son miroir qui vérifie $(i, j) \in A \iff (j, i) \in A'$, c'est-à-dire qui renverse toutes les arêtes.

Question IV.19 Écrire une fonction `miroir : graphe -> graphe` qui renvoie le miroir d'un graphe.

■

Démonstration.

Cette fonction c'est en fait le graphe par excès sans recopier les arêtes initiales sans retour :

```OCaml
let miroir g =
    let n = Array.length g.sommets in
    let sommets = Array.copy g.sommets in
    let aretes = Array.make n [] in
    for i = 0 to n - 1 do
        List.iter (fun j -> aretes.(j) <- i :: aretes.(j))
            g.aretes.(i)
    done;
    { sommets = sommets; aretes = aretes }
```

■

Dans la suite du sujet on note $\mathcal{G}$ le graphe des followers contenu dans le fichier. On va considérer dans la suite les graphes :

$\mathcal{G}, rev(\mathcal{G}), \mathcal{G}^-, \mathcal{G}^+, \mathcal{G}_{500}, rev(\mathcal{G}_{500}), \mathcal{G}^-_{500}$ et $\mathcal{G}^+_{500}$.

IV.2.ii Statistiques sur les degrés

Question IV.20 Écrire une fonction `stat_degre : graphe -> int * float` qui renvoie un couple (d_{max}, d_{moy}) où d_max est le plus grand des degrés du graphe et d_{moy} est le degré moyen donné par un nombre flottant.

■

Démonstration.

On itère directement sur les sommets :

```
OCaml
let stat_degre g =
    let n = Array.length g.sommets in
    let sum_d, max_d = ref 0, ref 0 in
    for i = 0 to n-1 do
        let d = List.length g.aretes.(i) in
        max_d := max !max_d d;
        sum_d := !sum_d + d
    done;
    !max_d, float_of_int !sum_d /. float_of_int n
```

Notons qu'on peut, ici aussi, écrire une fonction utilisant la bibliothèque standard efficacement :

```
OCaml
let stat_degre g =
    let a = Array.map List.length g.aretes in
    let fold = Array.fold_left in
    let foi = float_of_int in
    let n = Array.length g.aretes in
    fold max 0 a, foi (fold (+) 0 a) /. foi n
```

■

IV.2.iii Parcours en largeur

On va réaliser ici un parcours en largeur qui sera amené à être modifié et enrichi dans les questions suivantes. On vous laisse libre d'enrichir ce parcours en utilisant des fonctionnelles pour les traitements ou de modifier le code du parcours directement.

Pour utiliser une file, on va utiliser le module Queue. Dans le parcours on va calculer la fonction de distance `d` et pour gérer les cas où $d(x) = \infty$, on va la représenter par un `int option array`. Si `d.(x) = None` c'est que `x` est inconnu, on peut donc se servir de ce tableau pour avoir l'état d'un sommet.

Question IV.21 Écrire une fonction `bfs : graphe -> int -> int option array * int option array` telle que `bfs g x` renvoie un couple `(d,parent)` où `d.(y)` est la distance minimale de x à y et `parent.(y)` est l'indice du prédécesseur de `y` dans un tel chemin de x à y.

■

Démonstration.

On implémente directement le parcours en largeur vu dans le cours.

```
OCaml
let bfs g x =
    let n = Array.length g.sommets in
    let p = Array.make n None in
```

```
    let d = Array.make n None in
    let a_traiter = Queue.create () in
    Queue.add x a_traiter;
    d.(x) <- Some 0;
    while not (Queue.is_empty a_traiter) do
        let x = Queue.take a_traiter in
        List.iter (fun y ->
                  Queue.add y a_traiter;
                  p.(y) <- Some x;
                  d.(y) <- Some (Option.get d.(x) + 1))
            (List.filter (fun y -> d.(y) = None)
                g.aretes.(x))
    done;
    d, p
```

■

Question IV.22 Écrire une fonction `chemin : graphe -> int option array -> int -> int list` tel que `chemin g parent y` renvoie les sommets présents dans un chemin de x à y.

■

Démonstration.

On utilise ici la récusivité terminale pour remettre le chemin dans l'ordre.

OCaml
```
let chemin g p y =
    let rec aux y l =
        match p.(y) with
        | None -> y :: l
        | Some x -> aux x (y :: l)
    in aux y []
```

■

Question IV.23 Écrire une fonction `affiche_chemin : graphe -> int list -> unit` qui prend un graphe et un chemin donné par la fonction précédente et l'affiche avec le format :

```
compte1 -> compte1 -> ... -> compten
```

■

Démonstration.

OCaml
```
let affiche_chemin g p =
    Printf.printf "%s\n"
        (String.concat " -> "
            (List.map (fun i -> g.sommets.(i)) p))
```

■

Si $x \in S$, on note $\underline{x} = \{\, y \in S \mid x \rightsquigarrow y \,\}$.

Question IV.24 Écrire une fonction `accessibles : graphe -> int -> int` qui calcule le cardinal de $\underline{x}$ étant donné un graphe G et un sommet x donné par son indice.

Démonstration.

```ocaml
let accessibles g x =
    let d, p = bfs g x in
    Array.fold_left (+) 0
        (Array.map
            (fun v -> if v = None then 0 else 1) d)
```

Question IV.25 Écrire une fonction `max_accesibles : graphe -> int * int` qui renvoie un couple $(x, |\underline{x}|)$ où x est un sommet pour lequel $|\underline{x}|$ est maximal.

Démonstration.

```ocaml
let max_accessibles g =
    let m, v = ref 0, ref (accessibles g 0) in
    for i = 1 to Array.length g.sommets - 1 do
        let v' = accessibles g i in
        if v' > !v
        then ( v := v'; m := i )
    done;
    !m, !v
```

IV.2.iv Plus long chemin et diametre

Question IV.26 Écrire une fonction `plus_loin : graphe -> int -> int list` telle que `plus_long_chemin g x` renvoie le chemin de x au sommet y qui lui est le plus éloigné.

Démonstration.

```ocaml
let plus_loin g x =
    let n = Array.length g.sommets in
    let d, p = bfs g x in
    let i = ref 0 in
    while d.(!i) = None do
        incr i
    done;
    let m = ref (!i) in
    for j = !i+1 to n - 1 do
        match d.(j) with
        | Some v -> if v > Option.get d.(!m) then m := j
        | None -> ()
    done;
    let v = Option.get d.(!m) in
```

```
    !m, v, chemin g p !m
```

■

Question IV.27 Écrire une fonction `diametre : graphe -> int list` qui renvoie un chemin réalisant le diamètre d'un graphe.

■

Démonstration.

```ocaml
let diametre g =
    let n = Array.length g.sommets in
    let v, p = ref 0, ref [] in
    for i = 0 to n-1 do
        let j, v', chemin = plus_loin g i in
        if !v < v'
        then begin
            v := v';
            p := chemin
        end
    done;
    !p
```

■

IV.2.v Table de résultats

Attention : s'il faut peu de temps pour obtenir les résultats pour le sous-graphe de 500 sommets, c'est beaucoup plus long sur le graphe complet.

- $\mathcal{G}_{500}$:

```
degré max 10
degré moyen 0.430000
max_accessibles Mishkalashnikov avec 16 sommets
Diamètre 7 réalisé par :
Isaac__K -> naxonlabs -> faezeh_db -> MooreInst ->
    fath_gabrielle -> hypothesesorg -> ScienceFactor -> savantures
```

- $rev(\mathcal{G}_{500})$:

```
degré max 31
degré moyen 0.430000
max_accessibles savantures avec 76 sommets
Diamètre 7 réalisé par :
savantures -> ScienceFactor -> hypothesesorg ->
    fath_gabrielle -> MooreInst -> faezeh_db -> naxonlabs -> Isaac__K
```

- $\mathcal{G}_{500}^{-}$:

```
degré max 4
degré moyen 0.176000
max_accessibles SeverineWozniak avec 8 sommets
Diamètre 4 réalisé par :
QLMB8mars -> giu_sapio -> louise_tbr -> GroupeImpec -> halfbloodqueenx
```

- $\mathcal{G}^+_{500}$:

```
degré max 32
degré moyen 0.684000
max_accessibles helloselyn avec 98 sommets
Diamètre 12 réalisé par :
TsamiyahL -> FES_AFNEUS -> FlorestanAFNEUS -> FedeAddiction ->
    LS46151053 -> hypothesesorg -> Osec2022 -> ardakaniz ->
    ValRobert974 -> DialloAIbrahim2 -> Defense137 -> KArthemis ->
    SGF_GEOSOC
```

- $\mathcal{G}$:

```
degré max 950
degré moyen 36.266096
max_accessibles Boris_Brana avec 6049 sommets
Diamètre 9 réalisé par :
MonaEmara10 -> SambitPhD -> MIT_CSAIL -> MehdiKaytoue -> gromuald ->
    ECHARDE_ENSL -> cerseilia_ -> dadoyeldado -> Deccefunjoogu -> stoicsalik
```

- $rev(\mathcal{G})$:

```
degré max 3655
degré moyen 36.266096
max_accessibles JustVonBraun avec 7532 sommets
Diamètre 9 réalisé par :
Bonusbasci -> TCebere -> Miruna_Rosca -> h2020prometheus ->
    barENdSonLyon -> INP_CNRS -> ThierryCoulhon -> Phil_Baty ->
    HigherEdFutures -> HEMobilities
```

- $\mathcal{G}^-$:

```
degré max 610
degré moyen 12.069850
max_accessibles augabcoh avec 5352 sommets
Diamètre 10 réalisé par :
LeaLescouzeres -> Gauthier_tls -> MorganeBoulch -> CSNB14 ->
    leo_chapuis -> CCILYONMETRO -> IsabelleHuault -> Phil_Baty ->
    UNIKEhighered -> HigherEdFutures -> HEMobilities
```

- $\mathcal{G}^+$:

```
degré max 3655
degré moyen 60.462343
max_accessibles augabcoh avec 7854 sommets
Diamètre 7 réalisé par :
GabrielMarseres -> caroched -> MarieMoroso -> L3vironaute -> najatvb ->
    LeankonCarotte -> JustVonBraun -> Sardine49160063
```

IV.2.vi Aller plus loin

On propose ici plusieurs pistes de réflexions pour prolonger le TP :

- On a vu des algorithmes de dessin de graphes adaptés à des petits graphes. La présence de l'interaction sommet-sommet semble leur donner une complexité en $O(n^2)$ qui est rédhibitoire ici. Cependant, des sommets éloignés ont peu de chance d'interagir, comment pourrait-on modifier l'algorithme pour ignorer les interactions de répulsions entre sommets éloignés ? On remarque que la distance n'est pas un critère valide car les sommets peuvent être tous être superposés. Une manière de traiter cela efficacement est de découper le plan en région par des droites successives. Allez voir la page Binary Space Partionning et en déduire un algorithme effectif de dessin de graphe adapté.
- Pour estimer l'importance d'un compte, on ne peut pas se fier à son degré. En effet, celui-ci peut être augmenté artificiellement. Une manière fiable de mesurer l'importance est d'imaginer quelqu'un naviguant aléatoirement sur des comptes en suivant des liens d'abonnement et de mesurer la probabilité qu'il se retrouve sur un compte donné. C'est le principe qui est à la base de l'algorithme PageRank utilisé par Google. Implémenter cet algorithme et en déduire les comptes les plus importants dans cet exemple.