Graphes, Feuille de TD N^o 2 : Parcours

- *2020 :*
  Séance 2 :exos 1 à 5, 8

Rappel : parcours en profondeur de tout le graphe :

```
DFS(G) :
pour tous les sommets s, faire c[s] <- bleu
Initialisation()
pour tous les sommets s, faire
     si c[s] = bleu  alors  Depart()
                            Rec_DFS(s, ArgDuMain)
                            Arrivee()
Conclusion()

Rec_DFS (t, AutresArguments) :  /* t est suppose etre bleu */
c[t] <- blanc
Prefixe()
pour tous les successeurs/voisins u de t
       faire selon c[u] :  bleu :  PrefixeExt()    // variante de placement de Prefixe
                                   Rec_DFS(u, ArgDuRec)
                                   SuffixeExt()   // variante de placement de Suffixe
                           blanc : Blanc()
                           rouge : Rouge()
c[t] <- rouge
Suffixe()
```

Rappels :

BFS
pour tout sommet s faire couleur[s] ← bleu
Initialisation()
pour tous les sommets s faire
si couleur[s] = bleu
alors depart()
File < − vide
ajouter s a File
c[s] < − blanc
tant que File non vide
sortir u de la FIle
sortie()
pour tous les successeurs v de u
selon c[v]
bleu : ajouter v a la File
c[v] < − blanc
entree()
blanc : Blanc()
rouge : Rouge()
arrivee()
conclusion()

Entree(), Sortie() ... sont a definir suivant ce qu'on veut faire du BFS. L'algo sera lineaire si blanc,

rouge sont 0(constant), entree(), sortie(), depart, arrivee sont 0(degre), init, conclusion sont O(n+m).

La forme generale d'un DFS (Depth First Search)

```
DFSmain (G graphe)
pour tous les sommets s faire couleur[s] ← bleu
Initialisation()
pour tous les sommets s faire si couleur[s] = bleu
                                alors depart ()
                                     DFSslave(s)
                                     arrivee
conclusion
```

variante un

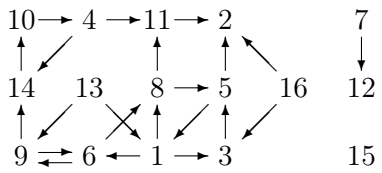
```
DFSslave (s : sommet ; autres arguments)
selon couleur de s :
            bleu :   Prefixe()
                     couleur[s] ← blanc
                     pour tous les successeurs x de s faire DFSslave(x)
                     Suffixe()
                     couleur[s] ← rouge
            blanc :  Blanc()
            rouge :  Rouge()
```

deuxieme variante :

```
DFSslave (s : sommet ; autres arguments)
/* s est bleu */
Prefixe()
couleur[s] ← blanc
pour tous les successeurs x de s
selon couleur[x] :
   bleu     DFSslave(x)
   blanc :  Blanc()
   rouge :  Rouge()
Suffixes()
couleur[s] ← rouge
```

Prefixe, Suffixe ... sont a definir suivant ce qu'on veut faire du DFS. L'algo sera lineaire si blanc, rouge sont 0(constant), prefixe, suffixe, depart, arrivee sont 0(degre), init, conclusion sont O(n+m).

1. Faire un DFS sur le graphe ci-contre. Donnez la forêt couvrante, le type des arcs et les numéros préfixe et suffixe des sommets. les sommets seront ordonnés par leur numéros.



   - *Forêt couvrante :*
     1 → *3* → *5* → *2*
     → *6* → *8* → *11*
     → *9* → *14* → *10* → *4*
     7 → *12*
     13
     15
     16
     On en déduit immédiatement les arêtes de l'arbre.

Arêtes descendantes hors arbres : (1,8)
Arêtes ascendantes : (4,14), (5,1), (9,6)
Arêtes vers la gauche : (4,11), (8,5), (11,2), (13,1), (13,9), (16,2), (16,3)
Ordre préfixe : 1, 3, 5, 2, 6, 8, 11, 9, 14, 10, 4, 7, 12, 13, 15, 16
Ordre suffixe : 2, 5, 3, 11, 8, 4, 10, 14, 9, 6, 1, 12, 7, 13, 15, 16

2. Utilisez un DFS pour décider si un graphe non-orienté est biparti.

- *Un graphe $G = (V, E)$ est biparti si et seulement si il existe une partition de V en deux sous-ensembles V_1 et V_2 tels que pour toute arête $\{u, v\}$ de E, on n'a ni $\{u, v\} \subseteq V_1$ ni $\{u, v\} \subseteq V_2$ (si on associe une couleur a V_1 et une couleur a V_2, deux sommets adjacents ne peuvent pas être dans le même sous-ensemble).*
  On vérifie d'abord quelques propriétés simples :
  1) Un graphe est biparti ssi toutes ses composantes connexes sont biparties (i.e si on sait déterminer si un graphe connexe est bipartie, alors on sait le faire pour un graphe quelconque).
  2)Un arbre est biparti. Pour construire une bipartition, choisissons un sommet r dans V et definissons V_1 comme l'ensemble des sommets e distance impaire de r et V_2 comme l'ensemble des sommets a distance paire de r.
  3) Si on choisit un sommet s d'un graphe connexe biparti et qu'on le place dans V_1 (quitte à échanger V_1 et V_2, alors V_1 est l'ensemble des points à distance paire de s et V_2 l'ensemble des points à distance impaire. De plus, tout chemin entre s et un sommet de V_1 est de longueur paire et tout chemin entre s et un sommet de V_2 est de longueur impaire. (on le voit en observant un chemin quelconque dans le graphe : chaque arête parcourue ajoute 1 à la longueur du chemin et fait changer d'ensemble V_i)
  Algorithme :

```
Bipartir(G):
    Pour tout s sommet de G:
        Ensembles(s) <- indéfini;
    Fin pour;
    test <- true;
    Pour tout s sommet de G:
        Si Ensembles(s) = indéfini:
            DFS_bipartir(G,s,1,test);
        Fin si;
    Fin pour;
    Renvoyer test;

DFS_bipartir(G,s,ens,test):
    Ensembles(s) <- ens;
    Pour tout t voisin de s:
        Si Ensemble(t) = indéfini:   /* ici on met t dans l'ensemble opposé à s */
            DFS_bipartir(G,t,(3-ens),test);
        Sinon si Ensembles(t) = ens: /* sinon si t est dans l'ensemble de s */
            test <- false;           /* c'est que G n'est pas biparti */
        Fin si;                      /* dans le dernier cas tout va bien */
                                     /* on ne fait rien */
```

- *On traite séparément l'une après l'autre chaque composante connexe de G puisque dans un graphe non orienté, un DFS lancé en s parcourt toute la composante connexe de s.*
  Le premier sommet de chaque composante connexe est arbitrairement placé dans l'ensemble V_1 (i.e $Ensembles(s) = 1$). Puis, au fur et à mesure qu'on parcourt le graphe, on met le sommet suivant dans l'ensemble opposé à celui de son prédécesseur dans le graphe de parcours (il ne pourrait évidemment pas être dans le même ensemble ; l'ensemble opposé est donné par $3 - Ensembles(s)$). On vérifie ensuite que toute arête dont les deux sommets sont actuellement coloriés est bien entre les deux ensembles V_1 et V_2.
  À la fin du parcours, soit G n'est pas biparti, donc il y a une arête entre V_1 et V_2, qu'on a découverte en coloriant sa deuxième extrémité à être parcourue, et `test` vaut `false`, soit G est biparti donc on n'en a trouvé aucune et `test` vaut toujours `true`.

C'est un DFS, qui tourne en en $\Theta(n+m)$.
Note : on peut aussi faire un BFS (à faire).

3. Donnez un algorithme linéaire qui détecte si un graphe G orienté possède un cycle, et dans l'affirmative, qui en rend un.
   - *Les sommets blancs sont ceux qui sont en cours de traitement, ce sont les sommets ouverts mais pas encore fermés. Il y a un chemin, qui est chemin suivi par l'exploration qui va du sommet de départ vers le sommet courant et qui passe exactement par les sommets blancs. Donc quand on traite un sommet et qu'on lui trouve un successeur blanc, alors il y a un cycle, car il y a un chemin, que l'on peut retrouver grâce a la pile de ce sommet blanc vers le sommet courant. Réciproquement, si on ne vois jamais de sommet blanc , alors on considère la numérotation postfixe (on attribue un numéro à chaque sommet lors de sa mise en rouge, avec un compteur qui s'incrémente à chaque affectation, cf tri topologique), et on montre que pour toute arête $x \to y$ on a $numero(x) > numero(y)$, et donc on obtient un tri topologique du graphe, qui ne peut donc pas avoir de cycle.*
     Ainsi, l'algorithme demandé est l'algorithme classique du DFS, avec :
     — *au début : on crée une pile, initialement vide ;*
     — *code préfixe : empiler le sommet ;*
     — *code suffixe : dépiler le sommet ;*
     — *si un voisin s est blanc : dépiler la pile jusqu'à retrouver s, le cycle est formé des sommets dépilés jusqu'à s.*
     Si l'algorithme termine sans avoir renvoyé de liste de sommets, c'est que le graphe est acyclique.

4. Donnez un algorithme linéaire qui détecte si un graphe non-orienté est acyclique.
   - *D'abord, pourquoi ne peut-on pas juste appliquer l'algorithme précédent, en remplaçant les arêtes du graphe non-orienté par des paires d'arêtes orientées ? Parce que dans ce cas toute arête non-orientée devient un cycle de taille 2, donc on trouverait des cycles dès que le graphe possède au moins une arête...*
     Idée : comme les graphes non-orientés acycliques sont les forêts, on cherche à reconnaître les arbres (et il suffit de vérifier que toute composante connexe est un arbre). À quoi ressemblent les voisins d'un sommet d'un arbre lors d'un DFS ?
     Idée 1 : Dans un arbre, soit on vient de commencer le parcours (i.e le sommet est la racine de l'arbre du DFS) et tous ses voisins sont bleus, soit il a un unique voisin blanc qui est son père. Donc si quand on ouvre le sommet x il a deux voisins blancs, c'est qu'on n'est pas dans un arbre (un graphe non orienté sur lequel on fait un DFS n'a pas d'arête vers la gauche, puisque ce serait comme avoir un arc à droite dans un graphe orienté : toutes les arêtes sont dans la forêt ou ascendante/descendante ; dans la question 6 on verra que ça montre qu'à l'ouverture de x, tous ses voisins sont bleus ou blancs mais jamais rouges).
     Cette condition est-elle suffisante ? Si le graphe contient un cycle, dont les sommets sont $s_1, \dots, s_k$, alors il existe i tel que s_i est le dernier sommet du cycle exploré par le DFS. Donc, au moment où s_i est atteint, s_{i-1} et s_{i+1} ont déjà été atteints par le DFS et sont blancs, donc l'algorithme détectera que le graphe n'est pas acyclique.
     Idée 2 : On peut appliquer la même idée que dans la question 3 mais en l'adaptant : pendant le DFS, on retient le père de chaque nouveau sommet exploré. Si lorsqu'on traite le sommet u et qu'on s'intéresse à son voisin v, celui-ci n'est ni bleu ni ni son père, alors c'est que le graphe n'est pas acyclique.
     Idée 3 : On note que G est acyclique ssi toutes ses arêtes sont exactement celles de la forêt du DFS. Dans le cas contraire, il y a au moins une autre arête, qui est donc ascendante/descendante comme vu dans l'idée 1. Cette arête est reconnaissable au fait qu'on voit un sommet rouge, ce qui n'arrive jamais si le graphe est acyclique.
     Pseudo-codes :

```
Acyclique(G):
    test <- true;
    c <- tableau(n,bleu);
    pour tout s sommet de G faire
        si c[s] <- bleu alors DFS(s);
        fin si;
```

```
    fin pour;
    renvoyer test;

/* possibilités pour DFS en fonction de l'idée exploitée : */

DFS1(u):
    c[u] <- blanc;
    cpt <- 0;
    pour tout v voisin de u faire
        si c[v] = blanc alors cpt++;
        fin si;
    fin pour;
    si cpt > 1 alors test <- false;
    fin si;
    pour tout v voisin de u faire
        si c[v] = bleu alors DFS1(v);
        fin si;
    fin pour;
    c[u] <- rouge;

DFS2(u,père):    /* lancer DFS2(s,null) dans Acyclique */
    c[u] <- blanc;
    pour tout v voisin de u faire
        si c[v] = bleu alors DFS2(v,u);
        sinon si c[v] = blanc alors
            si v != père alors test <- false;
            fin si;
        fin si;
    fin pour;
    c[u] <- rouge;

DFS3(u):
    c[u] <- blanc;
    pour tout v voisin de u faire
        si c[v] = bleu alors DFS3(v);
        sinon si c[v] = rouge alors test <- false;
        fin si;
    fin pour;
    c[u] <- rouge;
```

5. Donnez un algo linéaire qui prend en entrée un graphe G orienté avec les sommets numérotés de 1 à n, et qui remplit un tableau `MaxAncetre[sommet] de sommet`, où, pour tout sommet s $MaxAncetre(s) = max\{t \mid$ il y a un chemin de t vers $s\}$
   - *On note que si on lance un DFS du sommet s, on obtient une liste de tous les sommets qui sont accessibles depuis s. Si on lance ensuite – sans avoir réinitialisé les couleurs des sommets – un DFS du sommet t pas encore atteint, on obtient cette fois une liste des sommets accessibles depuis t mais pas depuis s.*
     Ainsi, pour résoudre le problème posé, il suffit de lancer un DFS classique, mais en choisissant les sommets de départ dans l'ordre décroissant, et en associant à chaque sommet le point de départ de la première étape du DFS à l'atteindre.
6. Complétez le code du parcours en profondeur pour remplir le tableau $TypeDArc[arc]$ qui donne le type de tout arc parmi $ArcDeLaForet$, $ArcVersLaGauche$, $ArcAscendant$, $ArcDescendant$.
   - *Quand x prend l'arc $x \to y$ pour voir y, il le trouve*
     — *en bleu si c'est un arc de la foret*
     — *en blanc si c'est un arc ascendant*
     — *en rouge si c'est un arc descendant*

— *en rouge si c'est un arc a gauche.*

On utilise donc ces couleurs pour déterminer la nature de l'arc $x \to y$.

Il faut néanmoins une idée supplémentaire pour distinguer les arcs gauches des arcs descendants.

1^{re} idée : si $x \to y$ va a gauche, alors le numéro préfixe de y est inférieur à celui de x, tandis que si c'est un arc descendant, le numéro de y est supérieur.

2^{me} idée : au moment ou x s'ouvre, le sommet y est encore bleu si c'est un arc descendant, mais est déjà rouge si c'est un arc à gauche

3^{me} idée : utiliser la couleur des prédécesseurs au lieu de la couleur a l'ouverture.

Pour un arc forêt, x voit y en bleu puis y voit x en blanc.

Pour un arc descendant, x voit y en rouge, puis y voit x en blanc.

Pour un arc ascendant, x voit y en blanc, puis y voit x en rouge.

Pour un arc gauche, y voit x en bleu, puis x voit y en rouge.

On fait alors deux boucles dans le DFS, une sur les prédécesseurs et une sur les successeurs.

On en tire les pseudo-codes suivants (tirés de la deuxième variante puisqu'il faut accéder aux deux sommets) :

```
DFS_1(G): /* Idée 1 */
    c <- tableau(n,bleu);
    num <- tableau(n,indéfini);
    type <- tableau(m,indéfini); /* un tableau indicé par les arêtes */
    cpt <- 0;
    pour tout s sommet de G faire
        si c[s] = bleu alors Rec_DFS(s);
    fin pour;
    renvoyer type;

Rec_DFS(u):
    c[s] <- blanc;
    cpt <- cpt+1;
    num[s] <- cpt;
    pour tout v successeur de u faire
        si c[v] = bleu alors
            type[x->y] <- forêt;
            Rec_DFS[v];
        sinon si c[v] = blanc alors
            type[x->y] <- ascendant;
        sinon      /* y est rouge */
            si num[x] > num[y] alors type[x->y] <- gauche;
            sinon type[x->y] <- descendant;
            fin si;
        fin si;
    fin pour;
    c[u] <- rouge;


DFS(G): /* Idée 2 */
    c <- tableau(n,bleu);
    type <- tableau(m,indéfini)
    pour tout s sommet de G faire
        si c[s] = bleu alors Rec_DFS(s);
    renvoyer type;

Rec_DFS(u):
    c[u] <- blanc;
    pour tout v successeur de u faire
        si c[v] = bleu alors type[u->v] <- forêt;
```

```
        sinon si c[v] = blanc alors type[u->v] <- ascendant;
        sinon type[u->v] <- gauche;
        fin si;
    pour tout v successeur de u faire
        si c[v] = bleu alors Rec_DFS;
        sinon si c[v] = rouge alors
            si type[u->v] = forêt alors type[u->v] <- ascendant;
            fin si;
        fin si;    /* si v est blanc on ne fait rien */
    fin pour;
    c[u] <- rouge;


DFS(G): /* Idée 3 */
    c <- tableau(n,bleu);
    type <- tableau(m,descendant)
    pour tout s sommet de G faire
        si c[s] = bleu alors Rec_DFS(s);
    renvoyer type;

Rec_DFS(u):
    c[u] <- blanc;
    pour tout t prédécesseur de u faire
        si c[t] = bleu alors type[t->u] <- gauche;
        fin si;
    fin pour;
    pour tout v successeur de u faire
        si c[v] = bleu alors
            type[u->v] -> forêt;
            Rec_DFS(v);
        sinon si c[v] = blanc alors type[x->y] <- ascendant;
        fin si;      /* on a déjà traité les cas où y est rouge */
    fin pour;
    c[u] <- rouge;
```

7. Donner le pseudo-code d'une fonction qui prend en entrée un graphe orienté G, supposé être sans cycle, deux sommets x et y et rend le nombre de chemins distincts de x à y.
   - *Le BFS ne marchera pas : il risque de ne pas bien propager les résultats.*
     Exemple : $x->a$, $a->b$, $b->c$, $c->y$, $x->c$.
     Le BFS met (1) en x puis (1) en a et en c puis(1) en b et en y puis la valeur en c devrait être mise à jour et passer à (2) mais comme c est déjà rouge, il n'y a pas propagation et y garde son (1) alors qu'il aurait fallu mettre (2).
     Solution correcte : comme le graphe est acyclique, on sait qu'il possède un tri topologique. Hors si on range les sommets dans cet ordre, d'abord on n'est intéressé que par les sommets entre x et y, et ensuite, on peut calculer $card[s]$ le nombre de chemins de s vers y en partant de y et en remontant dans l'ordre topologique jusqu'à x, avec $card[s] = \sum_{t succ de s} card[t]$ pour tout $s \neq y$ (on compte les chemins de s à y en fonction de leur deuxième sommet t). On initialise seulement par $card[y] = 1$.
     Hors calculer $card[s]$ en remontant l'ordre topologique entre x et y correspond à calculer $card[s]$ dans l'ordre suffixe du DFS partant de x.

```
nb_chemins(G,x,y):
    c <- tableau(n,bleu);
    card <- tableau(n,0);
    card[y] <- 1;
    DFS(x);
    renvoyer card[x];
```

```
DFS(u):
    c[u] <- blanc;
    pour tout v successeur de u faire
        si c[v] = bleu alors DFS(v);
        fin si;
        card[u] <- card[u]+card[v];
    c[u] <- rouge;
```

8. Où se trouvent dans la forêt couvrante les premiers et les derniers sommets numérotés en préfixe et en suffixe ?
   - — *Premier préfixe à la racine du premier arbre ;*
     — *dernier préfixe en bas à droite du dernier arbre ;*
     — *premier suffixe en bas à gauche du premier arbre ;*
     — *dernier suffixe à la racine du dernier arbre.*
9. (maison) Comment sont répartis les sommets d'une même CFC dans la forêt couvrante d'un DFS ?
   - *Soit C une composante connexe de G, s_0 le premier sommet de C atteint par le parcours. Alors comme aucun sommet de C à part s_0 n'est encore dans la forêt du DFS, et qu'ils sont tous accessibles depuis s_0, ils seront tous descendants de s_0 dans la forêt du DFS.*
     En particulier, tous les sommets de C sont dans le même arbre, et ils ont un ancêtre commun qui est lui-même dans C (c'est le premier atteint par le DFS). On appellera ancre de la CFC cet ancêtre commun.
     De plus, soit s_0 comme décrit précédemment et $s \in C$, alors tout sommet t situé entre s_0 et s (qui est son descendant) dans la forêt du DFS appartient à C (en effet, il existe un chemin de s_0 à t, et comme il existe un chemin de t à s (dans l'arbre) et un de s à s_0 (car $s \in C$), il existe un chemin de t à s_0 dans G, donc s_0 et t sont dans la même CFC, donc $t \in C$).
     En revanche, un arbre de la forêt peut contenir plusieurs CFC : voir le graphe $1 \to 2$ parcouru à partir du sommet 1.

   On fait une numérotation postfixe des sommets lors d'un DFS. Soit x le dernier sommet numéroté. Soit y un sommet. Montrez qu'un sommet y est dans la CFC de x ssi il est co-accessible depuis x.
   - *On note que x est la racine du dernier arbre de la forêt couvrante.*
     On a dit que x et y sont dans la même CFC $\Rightarrow$ x et y sont dans le même arbre de la forêt couvrante, i.e y est dans le dernier arbre. De plus, y est dans le dernier arbre $\Rightarrow$ y est accessible depuis x, donc dans ce cas x et y sont dans la même CFC ssi x est accessible depuis y.
     Donc on a bien x et y sont dans la même CFC $\iff$ x est accessible depuis y, i.e y est co-accessible depuis x.

   Comment trouver les sommets de la CFC de x ?
   - *On fait un parcours (en profondeur ou en largeur si on veut) à partir de x mais en remontant les arêtes au lieu de les descendre, c'est-à-dire qu'au lieu de considérer les successeur d'un sommet u on considère ses prédécesseurs. On trouve ainsi tous les sommets co-accessible depuis x ; comme x est dans le dernier arbre, il n'est accessible depuis aucun sommet d'un arbre précédent (sinon il aurait été dans un de ces arbres) donc tous les sommets trouvés sont bien dans son arbre originel, et donc dans la CFC de x.*

   Soit G_x le sous-graphe obtenu en enlevant à G les sommets de la CFC de x.
   Quel liens y-a-t-il entre les CFC de G et celles de G_x ?
   - *Les CFC de G_x sont celles de G moins celle contenant x.*
     En effet, toute paire de sommets u et v dans une même CFC ont leurs chemins entre eux entièrement inclus dans cette CFC (même raisonnement que pourquoi t est dans la CFC de s et s_0 précédemment) donc si u et v étaient dans une même CFC autre que celle de x dans G, alors ils sont toujours dans une même CFC dans G_x.
     Réciproquement, il n'y a pas de chemin dans G_x qui n'existaient pas dans G, donc deux sommets qui sont dans la même CFC dans G_x étaient dans la même CFC dans G.

   La forêt du DFS de G, privée des sommets de la CFC de x, est-elle la forêt d'un DFS de G_x ?
   - *Oui : le parcours de G_x donnera les mêmes premiers arbres, il n'y a que le dernier qui sera modifié. Si on considère ensuite les sommets restant dans l'ordre préfixe du parcours de G, on*

obtiendra les mêmes arbres que ceux donnés en retirant les sommets de la CFC de x au dernier arbre originel.

Soit x' le dernier sommet numéroté qui n'est pas dans la CFC de x. Comment trouver les sommets de la CFC de x' ?

- *Comme ce sommet correspond à la racine du dernier arbre du parcours de G_x décrit précédemment, il suffit de lancer un parcours de G_x en prenant les arêtes à l'envers à partir de x'.*

Donnez un algo linéaire pour trouver les CFC d'un graphe.

- *On pré-traite si nécessaire le parcours de façon à pouvoir accéder aux prédécesseurs d'un sommet en temps constant.*
  On fait d'abord un DFS normal sur G qui range les sommets dans l'ordre suffixe.
  On lance ensuite un parcours en profondeur du graphe en prenant les arêtes à l'envers, qui part des sommets (s'ils sont encore bleus) dans l'ordre inverse de l'ordre suffixe. À chaque nouveau lancement, on obtient alors une des composante fortement connexe du graphe, comme on l'a montré précédemment.
  En fait, puisqu'on veut prendre les sommets dans l'ordre inverse de l'ordre suffixe original, on peut tout simplement utiliser une pile pour les stocker.
  Dans tous les cas, on a effectué deux parcours en profondeur sur g et son inverse, donc on obtient bien un algorithme en temps linéaire.
  Pseudo-code :

```
CFC(G):
    c <- tableau(n,bleu);
    CFC_sommets <- tableau(n,indéfini);
    pile <- vide;
    pour tout s sommet de G faire
        si c[s] = bleu alors P1(s)
        fin si;
    fin pour;
    cpt <- 0;
    tant que pile est non vide faire
        s <- pop(pile);
        si c[s] = bleu alors
            cpt++;
            P2(s);
        fin si;
    fin tant que;
    renvoyer CFC_sommets;

P1(u):
    c[u] <- rouge;    /* ici on n'a pas besoin de distinguer blanc et rouge */
    pour tout v successeur de u faire
        si c[v] = bleu alors P1(v)
    fin pour;
    ajouter u à la file;

P2(u):
    c[u] <- rouge;
    CFC_sommets[u] <- cpt;
    pour tout v prédécesseur de u faire
        si c[v] = bleu alors P2(v);
        fin si;
    fin pour;
```

10. (maison) Donner un algorithme de tri topologique qui n'utilise pas de parcours. Analyser le coût de cet algorithme. Quelles structures de données peut-on utiliser pour faire le travail efficacement ?

- *On va utiliser le fait que tout graphe orienté acyclique admet une source, c'est-à-dire un sommet de degré entrant 0 (c'est vrai car si on voyage en empruntant les arcs à l'envers, soit on trouve un cycle soit on finit par être bloqué, donc on trouve une source), et que cette source peut être le premier élément d'un tri topologique du graphe puisqu'elle n'a aucun prédécesseur.*
  On va donc à chaque étape chercher une source, l'ajouter en bout de l'ordre créé, puis la retirer du graphe.
  Pour conserver une complexité linéaire, il suffit de vérifier que `ajouter` *et* `retirer` *sont en temps constant : par exemple en prenant pour* `E` *une pile ou une file.*

```
Tri_topologique(G):
    TT <- tableau(n,indéfini);
    DE <- tableau(n,0);         /* tableau des degrés entrants */
    pour k de 1 à m faire       /* on calcule chaque degré entrant */
        DE[Succ[k]]++;
    fin pour;
    cpt <- 1;
    E <- vide;                  /* ensemble des sources du graphe */
    pour s de 1 à n faire
        si DE[s] = 0 alors ajouter s à E;
    fin pour;
    tant que E n'est pas vide faire
        retirer x de E;         /* x quelconque dans E */
        TT[x] <- cpt;
        cpt++;
        pour tout y successeur de x faire
            DE[y]--;
            si DE[y] = 0 alors ajouter y à E;
        fin pour;
    fin tant que;
    renvoyer TT;                /* tableau des numéros dans l'ordre topologique*/
```

```
Boucle_BFS(s)
            Depart()
            c[s] <- blanc
            ajouter s a la file
            tant que la file n'est pas vide
                  extraire t de la file
                  c[t] <- rouge
                  Sortie()
                  pour tous les successeurs/voisins u de t faire
                        selon c[u] :
                              bleu  : c[u] <- blanc
                                      ajouter u a la file
                                      Entree()
                              blanc : Blanc()
                              rouge : Rouge()
            Arrivee()
```

parcours en largeur depuis s_0 :

```
BFS(G, s0) :
pour tous les sommets s, faire c[s] <- bleu
file <- vide
Initialisation()
Boucle_BFS(s0)
Conclusion()
```

parcours en largeur de tout le graphe :

```
BFS(G) :
pour tous les sommets s, faire c[s] <- bleu
file <- vide
Initialisation()
pour tous les sommets s, faire
      si c[s] = bleu
      alors   Boucle_BFS(s)
Conclusion()
```

11. Utilisez un BFS pour aire un calcul de plus court chemin (les arc étant tous de poids 1) depuis le sommet 13 sur le graphe de l'exo 1. Les sommets seront ordonnés par leur numéros.
    - *Ordre de traitement : 13, 1, 9, 3, 6, 8, 14, 5, 11, 10, 2, 4.*

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*
1	*4*	*2*	*4*	*3*	*2*	∞	*2*	*1*	*3*	*3*	∞	*0*	*2*	∞	∞

12. Utilisez un BFS pour décider si un graphe non-orienté est biparti.
    - *Le procédé est très similaire à celui de la question 2 ; il faut juste placer le sommet dans l'ensemble imposé par son prédecesseur au moment où on le met dans la file.*

```
Bipartir(G):
    ensemble <- tableau(n,indéfini);
    test <- true;
    c <- tableau(n,bleu);
    file <- vide;
    pour tout sommet s de G faire
        si c[s] = bleu alors BFS_bipartir(s,1);
    fin pour;
    renvoyer test;

BFS_bipartir(s):
    c[s] <- blanc;
    ensemble[s] <- 1;
    ajouter s à la file;
    tant que la file n'est pas vide faire
        u <- pop(file);
        c[u] <- rouge;
```

```
        pour tout v voisin de u:
            si c[v] = bleu alors
                ensemble[u] <- 3-ensemble[s]; /* ensemble u opposé à ensemble[s] */
                c[v] <- blanc;
                ajouter v à la file;
            sinon si ensemble[v] = ensemble[u] alors test <- false;
            fin si;
         fin pour;
      fin tant que;
```

13. Donner un algorithme qui prend en entrée un graphe non-orienté G, un sommet s_0 et calcule $CardPCC[x]$ pour chaque sommet x, $CardPCC[x]$ étant le nombre de plus courts chemins (en nombre d'arêtes, le graphe n'est pas valué) de s_0 à x. (Exemple, si G est un 3-cube, et si s_0 et x sont diamétralement opposés, alors $CardPCC[x] = 6$)

- *Ici, le BFS est adapté, puisque soit s un sommet quelconque de G, le nombre de plus courts chemins entre s_0 et s (i.e de chemins de longueur $d(s_0, s)$) est égal à $\sum_{t \text{ voisin de } s \text{ tq } d(s_0,t)=d(s_0,s)-1} CardPCC[t]$ (on compte les chemins de longueur $d(s_0, s)$ entre s et s_0 en fonction de leur première arête). On initialise donc l'algorithme par $CardPCC[s_0] = 1$ et $d[s_0] = 0$, et lorsqu'on arrive à un sommet s à distance $d[s]$ de s_0, tous les sommets à distance $d[s] - 1$ de s_0 ont déjà été traités : on peut donc calculer aisément la somme précédente.*

```
CardinalPCC(G,s0):
    cardPCC <- tableau(n,0);
    couleur <- tableau(n,bleu);
    distance <- tableau(n,infini);
    cardPCC[s0] <- 1;
    distance[s0] <- 0;
    ajouter s0 à la file;
    tant que file est non vide faire
        u <- pop(file);
        couleur[u] <- rouge;
        pour tout v voisin de u faire
            si couleur[v] = bleu alors
                couleur[v] <- blanc;
                ajouter v à la file;
                distance[v] = distance[u]+1;
            sinon si distance[v] = distance[u] - 1 alors
                cardPCC[u] = cardPCC[u] + cardPCC[v];
                fin si;
            fin si;
        fin pour;
    fin tant que;
    renvoyer cardPCC;
```

- *On note qu'ici, c'est quand on atteint u qu'on détermine $CardPCC[u]$ à partir de ceux de ses voisins. On peut choisir de déterminer $CardPCC[u]$ avant de le retirer de la file, au fur et à mesure qu'on traite ses voisins plus proches de s_0.*

14. Dans un graphe non-orienté, les sommets sont coloriés dans l'une des trois couleurs vert, gris, violet.

    Donner un algorithme linéaire qui trouve deux sommets, l'un vert, l'autre violet, les plus proches possibles. Les distances sont en nombre d'arêtes.

- *L'idée est simplement d'effectuer un BFS mais en initialisant la pile avec tous les sommets verts au lieu d'un seul sommet de départ. On vérifie alors aisément qu'on va d'abord traiter les sommets à distance 0 d'un sommet vert (i.e les sommets verts eux-mêmes), puis leurs voisins à distance 1 de l'ensemble vert, puis les voisins des voisins... et le premier sommet violet ainsi rencontré est bien à distance minimale de l'ensemble vert.*
  Il reste ensuite simplement à conserver en mémoire, dans un tableau $Index$, un sommet vert à distance minimale de chaque sommet déjà traité.

```
Distance_ensembles(G,couleur): /* ici couleurs contient vert, gris ou violet */
    c <- tableau(n,bleu);   /* tableau de l'état de considération des sommets */
    index <- tableau(n,indéfini);
    pour tout s sommet de G faire
        si couleur[s] = vert alors
            ajouter s à la file;
            c[s] <- blanc;
            index[s] <- s;
        fin si;
    fin pour;
    tant que la file est non vide faire
        u <- pop(file);
        c[u] <- rouge;
        pour tout v voisin de u faire
            si couleur[v] = violet alors renvoyer (index[u], v)
            fin si;  /* dans ce cas on interrompt l'algo : on a trouvé */
            si c[v] = bleu alors
                c[v] <- blanc;
                index[v] <- index[u];
                ajouter v à la file;
            fin si;
        fin pour;
    fin tant que;
    renvoyer une erreur;  /* si on arrive à cette ligne c'est qu'il n'y a pas */
                          /* de sommet violet accessible depuis un sommet vert */
                          /* donc pas de solution */
```

15. (maison) Plus courts chemins avec poids 0 ou 1 : On a un graphe orienté dont les arcs sont valués par des poids. Pour tout arc, le poids est 0 ou 1. Il y a un sommet source s. On veut determiner pour chaque sommet x la distance de s vers x (valuation du meilleur chemin de s à x).

    (a) Rappeler comment obtenir efficacement les distances quand toutes les valuations sont égales à 1.

    - *Il suffit de faire un parcours en largeur.*

    (b) Donner un algorithme permettant de trouver tous les sommets à distance 0.

    - *On fait un parcours quelconque en ne considérant que les arêtes de poids nul.*

    (c) Décrire comment trouver tous les sommets à distance 1.

    - *On peut commencer par trouver tous les sommets à distance 0 de s, puis tous les sommets à distance 1 de ces sommets (qui sont d'ailleurs les sommets non encore traités à une arête d'un de ces sommets), puis on re-cherche les sommets à distance 0 à partir de chaque sommet à distance 1 déjà trouvé. Ainsi, tout sommet à distance 1 passe par exactement une arête de poids 1, et potentiellement des arêtes de poids 0 avant ou après.*

    (d) Donner un algorithme efficace donnant les distances (expliquer puis donner un pseudo-code). Donner la complexité de votre algorithme.

    - *On fait un parcours en largeur le long des arcs de poids 1, sauf qu'à chaque fois qu'on ajoute un sommet dans la file, on rajoute également tous les sommets qui sont à distance 0 de lui. Pour ça, on prend une fonction auxiliaire qui fait ce qui est décrit en (b) qu'on applique au sommet s à chaque fois qu'on ajoute s dans la file.*
      Il faut également bien garder en mémoire la distance à s_0 à chaque étape de l'algorithme.

```
distances(G,s0):
    c <- tableau(n,bleu);
    d <- tableau(n,infini);
    file <- vide;
    propage(s0,0);
    tant que file est non vide faire
        u <- pop(file);
        pour tout v successeur de u faire
```

```
            si c[v] = bleu alors propage(v,d[u]+1);
                    /* comme on applique propage au moment de l'entrée dans */
                    /* la file, si v est bleu c'est que poids(u,v) = 1 */
            fin si;
        fin pour;
    fin tant que;
    renvoyer d;

propage(u,d):
    c[u] <- rouge;  /* pas de distinction entre blanc et rouge ici /*
    d[u] <- d;
    ajouter u à la file;
    pour tout v successeur de u faire
        si c[v] = bleu et poids(u,v) = 0 alors propage(v,d);
        fin si;
    fin pour;
```

- *On note qu'ici pour* `propage` *on utilise un DFS : le type de parcours n'est pas important et cela permet de ne pas devoir utiliser une deuxième file en parallèle.*