Feuille de TD N^o 3

Rappelons le principe du parcours en profondeur avec les calculs des "dates de découverte" et des "dates de fin de traitement" :

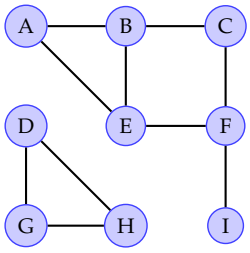
```
visiter_pp(g,s) :
  si dec[s] = -1 :
   dec[s]<-k; k++;
   pour tous les voisin x de s :
      visiter_pp(g,x);
   fin[s]<-k; k++;

visiter(g) :
   k<-0;
   pour chaque sommet s :
      dec[s]<- -1; fin[s]<- -1;
   pour chaque sommet s :
      si dec[s]= -1 :
         visiter_pp(g,s);
```

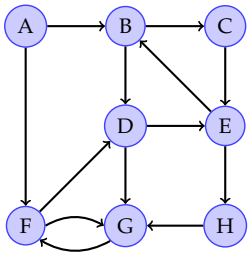
1 Familiarisation avec le parcours en profondeur

Considérons les graphes suivants. Effectuer pour chacun d'eux un parcours en profondeur et mettre en évidence la forêt du parcours. Distinguer les arêtes de la forêt des autres. Donner, à chaque fois, les dates de découverte et de fin de traitement de chacun des sommets.

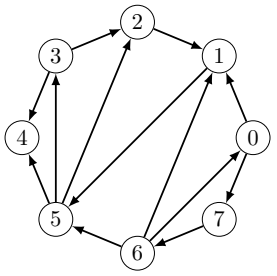
Pour effectuer un parcours en profondeur il faut choisir un ordre sur les voisins. Pour chacun des exemples, faites une version en prenant l'ordre des sommets (ordre sur les entiers, alphabétique) puis l'ordre inverse.



Graphe (a)



Graphe (b)



Prouver le fait suivant : si on considère un graphe non orienté et la forêt de l'un de ses parcours en profondeur alors les arêtes du graphe qui ne sont pas des arêtes de la forêt, sont des arêtes qui relient un sommet du graphe à l'un de ses prédécesseurs (ancêtre) dans l'arborescence. On dit qu'il n'existe pas d'arête transverse.

On en déduit immédiatement la reformulation suivante : si s est un noeud de T ayant pour fils u et v, alors il n'existe aucune arête entre un descendant de u et un descendant de v.

1. Ecrire un programme en Ocaml qui utilise le parcours en profondeur pour déterminer les composantes connexes d'un graphe non orienté représenté par ses listes d'adjacence. On pourra renvoyer un tableau indexé par les sommets qui contient dans la case i le numéro de la composante connexe dans laquelle se trouve le sommet i.
2. Réécrire l'algorithme de parcours en profondeur du cours de manière à renvoyer en sortie la forêt de parcours. On représentera une forêt par la donnée des pères des sommets et donc par un type `foret = int array`. Si `pere : int array` représente une forêt, `pere.(i)` pointe vers l'unique entier qui est le père de `i` ou vaut `-1` si le sommet `i` est une racine.
3. Ecrire un programme Ocaml qui renvoie une liste contenant les sommets d'un graphe orienté représenté par ses listes d'adjacence en ordre topologique.

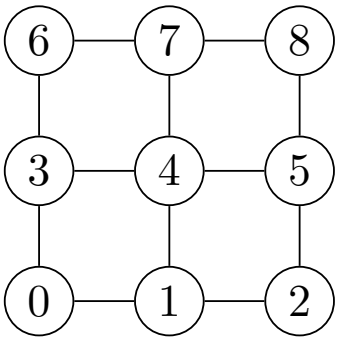
2 Parcours génériques

Considérons les deux stratégies suivantes :

```
Fonction WFS1(G, s_0)
    Mettre s_0 dans le sac.
    Tant que le sac n'est pas vide Faire
        Sortir un élément s du sac.
        Si s n'est pas marqué Alors
            Marquer s.
            Pour tout t voisins de s Faire
                Mettre t dans le sac.
```

```
Fonction WFS2(G, s_0)
    Mettre s_0 dans le sac.
    Marquer s_0.
    Tant que le sac n'est pas vide Faire
        Sortir un élément s du sac.
        Pour tout t voisins de s Faire
            Si t n'est pas marqué Alors
                Mettre t dans le sac.
                Marquer t.
```

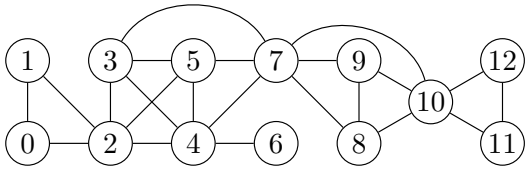
1. On suppose que le sac est une file. Déterminer en justifiant si les données suivantes sont modifiées entre les deux parcours :
   (a) l'ordre de parcours ;
   (b) la complexité temporelle ;
   (c) la complexité spatiale.
2. On suppose que le sac est une pile. En appliquant WFS1 et WFS2 au graphe suivant, déterminer l'ordre de parcours et l'arborescence du parcours. On supposera que les listes d'adjacence sont triées par ordre croissant.



3 Sommets coupants

Soit $G = (S, A)$ un graphe non orienté connexe. On dit que $s \in S$ est un sommet coupant si le graphe induit par $S \setminus \{s\}$ n'est pas connexe.

1. Déterminer les sommets coupants du graphe suivant :



2. Décrire un algorithme linéaire qui détermine, étant donné G et s, si s est un sommet coupant ou non.
3. En déduire un algorithme naïf qui détermine tous les sommets coupants d'un graphe G donné et déterminer sa complexité.
4. Soit T l'arborescence d'un parcours en profondeur de G depuis un sommet s quelconque.
   (a) Montrer que s est un sommet coupant de G si et seulement si le degré de s dans T est de degré au moins 2.
   (b) Montrer qu'un sommet $t \in S, t \neq s$ est un sommet coupant de G si et seulement si t possède un fils u tel qu'aucun descendant (dans T) de u n'est voisin (dans G) d'un ancêtre strict (dans T) de t.
   (c) En déduire un algorithme linéaire qui détermine tous les sommets coupants d'un graphe G donné. (on pourra calculer, pour chaque sommet u, la date de découverte du sommet accessible depuis l'arbre enraciné en u qui a été découvert en premier)

4 Union Find

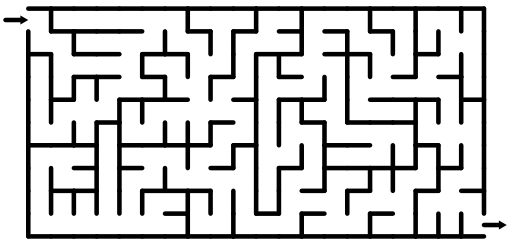
1 Exhiber une suite d'opérations de la structure union find qui à partir de la partition contenant les 8 premiers entiers isolés conduit à la figure présentée sur la première page du cours.

2 Mettre en oeuvre une légère modification de la structure union find pour qu'elle contienne en plus le nombre de classes distinctes contenu dans la partition.

3 Utiliser la structure union-find sur les sommets d'un graphe non orienté connexe pour déterminer s'il est ou non acyclique.

4 On peut utiliser la structure union-find pour construire efficacement un labyrinthe parfait. Un labyrinthe est dit parfait lorsque pour tout couple de cases, il existe un unique chemin élémentaire les reliant.

Voici un exemple :



Pour créer un tel labyrinthe, on crée une structure union-find dont les éléments sont les cases. Deux cases seront dans la même classe ssi elles admettent un chemin élémentaire unique les reliant. Initialement, toutes les cases sont séparées puis on considère toutes les cases adjacentes (verticalement et horizontalement) et pour chaque paire `(c1,c2)`, on compare les calsses de `c1` et `c2`, si elles sont identiques, on ne fait rien et sinon on ouvre la porte les séparant.

1. On va commencer par générer un tableau qui contient toutes les portes possibles. On représente une porte par un triplet `(i,j,d)` où la porte est sur la droite de la case `(i,j)` si $d = H$ et sur le bas si $d = V$. Ecrire une fonction prenant en entrée `n`, la taille du côté du labyrinthe et qui renvoie ce tableau. Il sera nécessaire de déclarer le type : `type dir = H|V`.
2. On va associer à chaque case `(i,j)` un entier entre 0 et $n^2 - 1$ pour pouvoir utiliser la structure union-find. Etablir une bijection simple entre $[|0, n-1|] \times [|0, n-1|]$ et $[|0, n^2-1|]$ qui permette de réaliser cette relation.
3. Mettre en oeuvre la construction du labyrinthe à l'aide d'une stratégie qui parcourt le tableau réalisé à la première question pour faire les unions proposées par la méthode. La fonction renverra deux tableaux de booléens `hor` et `vert` tels que `hor.(i).(j)` sera `true` ssi il y aune porte à droite de la case `(i,j)` et `vert.(i).(j)` sera `true` ssi il y aune porte en bas de la case `(i,j)`.
4. Justifier que le labyrinthe obtenu est bien parfait.
5. La méthode telle qu'elle est décrite va construire un unique labyrinthe qui dépend de l'ordre dans lequel sont placées les portes potentielles dans le tableau créé dans la question 1. Afin de pouvoir générer de multiples labyrinthes, il faudrait au préalable mélanger ces portes. On peut par exemple utiliser le mélange de Knuth dont le principe est le suivant : on parcourt le tableau de la droite vers la gauche et on échange aléatoirement l'élément de la case `i` avec l'élement d'une case choisie aléatoirement entre 0 et i (inclus). Ecrire une fonction qui réalise ce mélange.

   On utilisera la fonction `Random.int` qui prend en entrée un entier k et renvoie uniformément un élément de $[|0, k-1|]$.