Définitions : rappel

- Un graphe non orienté est dit **connexe** si pour toute paire de sommets distincts x et y, il existe une chaîne reliant x à y.
- On appelle **composante connexe** d'un graphe non orienté un sous-graphe connexe maximal, c'est à dire qui n'est pas strictement contenu dans un autre sous-graphe connexe.

Donc un graphe connexe contient une unique composante connexe.

Lorsque le graphe est non connexe, connaître ses composantes permet de savoir, sans parcours, si deux sommets sont reliés par une chaîne.

1 Cas statique

1.1 Parcours largeur ou profondeur

Déterminer les composantes connexes

Dans la forêt couvrante issue d'un parcours largeur ou profondeur, chaque arbre correspond à une composante connexe.

En pratique, il suffit de remplacer le vecteur de booléens utilisé pour marquer les sommets par un vecteur d'entiers : chaque sommet sera marqué par le numéro de sa composante. Ce numéro changera à chaque retour dans l'algorithme d'appel (à chaque arborescence).

Par exemple, la fonction `composantes(`G`, `cc`)` ci-dessous retourne le nombre de composantes de G, composantes représentées par le vecteur cc.

```
procedure comp_rec(graphe G,
                   entier x, nocc,
                   vect_entiers ref cc)
variables
   entier    y, i
debut
   cc[x] ← nocc
   pour i ← 1 jusqu'a d+(x,G) faire
       y ← i ème-succ-de x dans G
       si cc[y] = 0 alors
           comp_rec(G, y, nocc, cc)
       fin si
   fin pour
fin
```

```
fonction composantes(graphe G, vect_entiers ref cc):entier
variables
   entier s, nocc
debut
   pour s ← 1 jusqu'a N faire
       cc[s] ← 0
   fin pour
   nocc ← 0
   pour s ← 1 jusqu'a N faire
       si cc[s] = 0 alors
           nocc ← nocc + 1
           comp_rec(G, s, nocc,cc)
       fin si
   fin pour
   retourne nocc
fin
```

La fonction `composantes` donnera le même résultat avec un parcours largeur (dans lequel il suffit de marquer également les sommets avec `nocc`).

Déterminer la connexité

Cette fonction peut être utilisée pour vérifier la connexité : il suffit de vérifier qu'il n'y a qu'une seule composante.

Cela dit, il n'est pas nécessaire de faire un parcours complet pour vérifier si un graphe est connexe : il suffit de lancer un parcours (profondeur ou largeur) à partir d'un sommet quelconque et, au retour, vérifier si tous les sommets ont été marqués par cet unique parcours.

1.2 Calcul de la fermeture transitive

Calculer la fermeture transitive d'un graphe sert à déterminer si ce graphe est connexe.

Définition

On appelle *fermeture transitive* d'un graphe $G = < S, A >$, le graphe $G^* = < S, A* >$ tel que pour toute paire de sommets $x, y \in S$, il existe une arête $\{x, y\}$ dans G^* si-et-seulement-si il existe une chaîne entre x et y dans G.

Remarques :
- Un graphe G est connexe si-et-seulement-si sa fermeture transitive G^* est un graphe complet
- Deux sommets x, y sont de la même composante connexe de G si-et-seulement-si il existe une arête $\{x, y\}$ dans la fermeture transitive G^* de G.

Algorithme de Warshall

L'algorithme de warshall construit la fermeture transitive d'un graphe sous la forme d'une matrice d'adjacence.

Cette matrice est initialisée au départ par la matrice d'adjacence du graphe. Elle va ensuite être enrichie à chaque itération par les nouvelles connexions découvertes. À chaque itération i la matrice calculée indique pour toutes les paires de sommets (x, y), s'il existe une chaîne de x à y passant par des sommets inférieurs ou égaux à i :
- soit il existait déjà une chaîne à une itération précédente
- soit il existe une chaîne de x à i et une chaîne de i à y.

```
procedure Warshall(graphe G, mat_booleens ref C)
variables
   entier x, y, i
debut
   C ← /* Matrice d'adjacence du graphe G */

   pour i ← 1 jusqu'a N faire
      pour x ← 1 jusqu'a N faire
         pour y ← 1 jusqu'a N faire
            C[x,y] ← C[x,y] ou (C[x,i] et C[i,y])
         fin pour
      fin pour
   fin pour
fin
```

Pour un graphe d'ordre n avec m arêtes, l'algorithme de Warshall est de complexité n^3. Nettement plus coûteux donc qu'un parcours (profondeur ou largeur) de complexité $O(max(n, m))$ (ou n^2 au pire si le graphe est représenté par une matrice d'adjacence).

Warshall peut être utilisé pour les graphes orientés : dans ce cas `C[x,y]` indique s'il existe un *chemin* $x \rightsquigarrow y$ dans `G`.

2 Cas évolutif

Un graphe non orienté de n sommets peut être connu par la donnée successive de ses arêtes. Les composantes connexes du graphe peuvent alors être obtenues par regroupements successifs des composantes connexes du graphe en évolution.

L'ajout d'une arête x — y peut modifier l'ensemble des composantes connexes d'un graphe en réunissant celle de x avec celle de y si elles sont distinctes. On doit alors être capable :
- de dire à quelle composante connexe appartient un sommet donné ;
- de savoir si deux sommets appartiennent ou non à une même composante ;
- de réunir deux composantes connexes en une seule.

Les composantes connexes seront ici définies comme des classes d'équivalence pour la *relation d'équivalence* ***chaîne***, où x ***chaîne*** y est vrai si-et-seulement-si il existe une chaîne entre les deux sommets x et y.

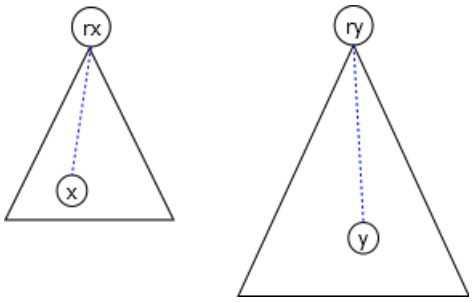
Trouver-réunir

Les classes d'équivalence/composantes connexes sont représentées par une forêt :
- un arbre par composante ;
- la racine de l'arbre est le *représentant* de la composante.

Cette forêt est représentée par un vecteur de pères p :
- si x est une racine, $p[x] = -1$
- sinon $p[x]$ est le père de x dans la forêt.

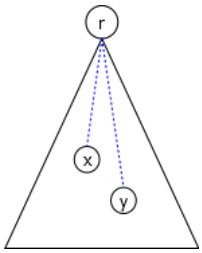
Trouver le représentant d'un sommet x, se fera alors en "remontant" depuis x jusqu'à la racine de l'arbre :

```
fonction trouver(entier x, vecteur p): entier
debut
    tant que p[x] <> -1 faire
        x = p[x]
    fin tant que
    retourne x
fin
```

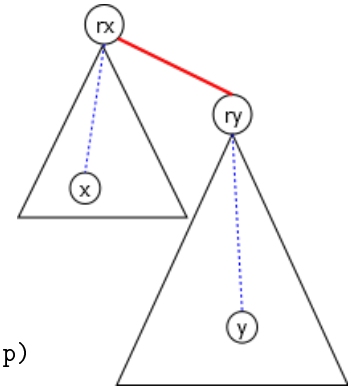


À chaque ajout d'une arête x — y :

Si x et y ont le même représentant, ils sont déjà dans la même composante.



Sinon, il faut **réunir** les deux composantes en accrochant ry à rx (ou l'inverse) : rx devient le père de ry.




```
procedure reunir(entier x, y, vecteur ref p)
debut
    rx = trouver(x, p)
    ry = trouver(y, p)
    si rx <> ry alors
        p[ry] = rx
    fin si
fin
```

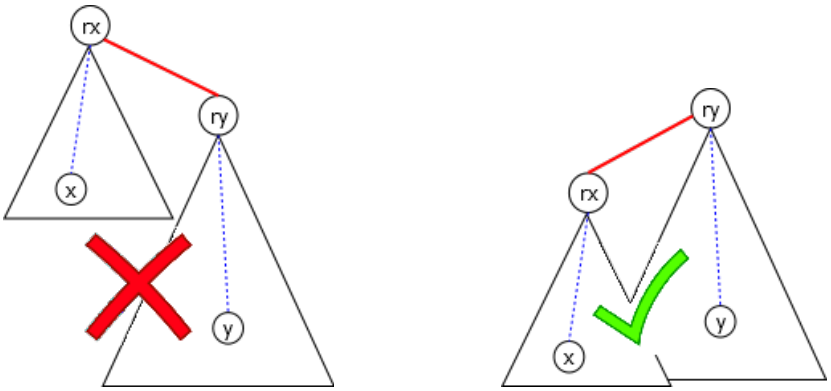
Optimisations

L'opération "coûteuse" est la recherche pour chaque sommet de son représentant : le coût de l'opération est au pire la hauteur de l'arbre (donc $O(n)$ pour un arbre dégénéré). Diminuer la hauteur des arbres permettra donc d'optimiser l'opération **trouver**.

Il existe deux améliorations qui combinées permettent d'obtenir des arbres dont les hauteurs tendent vers 2 : l'*union pondérée* et la *compression des chemins*.

Union pondérée

Au lieu d'accrocher systématiquement l'arbre contenant y à la racine de celui contenant x, nous allons accrocher celui de plus petite taille (contenant le moins d'éléments) à la racine de celui de plus grande, ce qui permet de réduire la hauteur de l'arbre résultat.



Pour cela, pour chaque représentant r_i, $p[r_i] = -n_i$, avec n_i la taille de l'arbre dont r_i est racine. Cela permet de choisir l'arbre le plus petit qui deviendra sous-arbre du plus grand (la nouvelle taille se calcule simplement en sommant les deux tailles initiales).

Compressions des chemins

À chaque appel de la fonction `trouver` pour un sommet x, on accroche directement à la racine de l'arbre contenant x tous les sommets (x compris) se trouvant sur le chemin allant de x à cette racine : ainsi la prochaine recherche du représentant d'un de ces sommets sera quasi-immédiate, puisqu'ils seront tous à une profondeur 1.