Algorithmique
Les graphes : parcours

Pour connaître les différentes propriétés d'un graphe, il est souvent nécessaire d'examiner exhaustivement les sommets et les arcs ou arêtes de celui-ci. Il est possible de déterminer certains propriétés en examinant simplement chaque sommet et cela dans n'importe quel ordre. Mais la plupart des algorithmes de graphes nécessite de suivre les liaisons : passer de sommets en sommets au travers des relations du graphe. C'est ce que l'on appelle les **parcours de graphes** :

Parcours en profondeur : Ce parcours consiste à suivre, à partir d'un sommet donné, un chemin le plus loin possible. Puis à revenir en arrière pour explorer les chemins ignorés.

Parcours en largeur : Ce parcours se fait par niveaux. On parcourt d'abord tous les sommets se trouvant à une distance de 1 du sommet de départ, puis tous ceux qui se trouvent à une distance de 2 et ainsi de suite...

Parcourir un graphe : les points communs

Dans les parcours, les sommets sont représentés par leur numéro : de 1 à N (l'ordre du graphe).

Les parcours de graphes sont des extensions des parcours des arbres généraux.

Les différences :

— Il est nécessaire de marquer les sommets visités à l'aide d'un ensemble : pour ne pas revenir dessus. Cet ensemble sera représenté ici par un vecteur de booléens, indiquant pour chaque sommets s'il a été visité.

— Le parcours d'un graphe doit être fait à partir d'un sommet donné (il n'y a pas de racine comme dans un arbre). Cela ne garantit pas d'avoir vu tous les sommets. Si on désire un parcours complet du graphe (où tous les sommets seront visités), il faut donc un **algorithme d'appel**, qui relancera ce parcours sur les éventuels sommets non visités.
L'algorithme d'appel sera le même quelque soit le parcours.

```
procedure appel_parcours (graphe G)
variables
    vect_booleens M
    entier s
debut
    pour s ← 1 jusqu'a N faire
        M[s] ← faux
    fin pour
    pour s ← 1 jusqu'a N faire
        si non M[s] alors
            parcours(G, s, M)
        fin si
    fin pour
fin
```

Le parcours d'un graphe peut être représenté graphiquement par une *forêt couvrante* associée au parcours :

— les racines des arbres de la forêt sont les sommets sur lesquels le parcours a été lancé depuis l'algorithme d'appel ;
— les liaisons sont celles empruntées lors du parcours : on parle d'*arcs couvrants* (**même pour les graphes non orientés**).

Il y a autant de forêts couvrantes que d'ordres possibles de rencontre des sommets !

La **complexité** d'un parcours complet dépend de l'implémentation choisie. Soit un graphe de n sommets et m liaisons :
— Avec l'implémentation par matrice d'adjacence, la complexité sera en $O(n^2)$;
— avec une implémentation par listes d'adjacence, elle sera en $O(max(n, m))$.

Parcours profondeur

```
procedure parcours_profondeur(graphe G, entier x, vect_booleens ref M)

variables
    entier    y, i
debut
     /* première rencontre de x = préfixe */
    M[x] ← vrai
    pour i ← 1 jusqu'a d°+(x,G) faire
        y ← i ème-succ-de x dans G
        si non M[y] alors
            parcours_profondeur(G, y, M)
        fin si
    fin pour
     /* dernière rencontre de x = suffixe */
fin
```

En dehors des arcs couvrants (seuls arcs constituant la forêt couvrante), il est utile de classifier les autres liaisons du graphe.

Dans un **graphe orienté**, l'arc (x, y) est, lors du parcours profondeur :

couvrant : arc emprunté lors du parcours (x est le père de y)

avant : y est un descendant (pas direct) de x dans la forêt couvrante associée.

arrière (retour) : y est un ascendant de x dans la forêt couvrante associée

croisé : x et y n'ont aucune relation d'ascendance dans la forêt couvrante associée.

Dans un **graphe non orienté**, les liaisons sont symétriques : celles qui ne sont pas des arcs couvrants lors du parcours sont des *arcs arrières*.

Parcours largeur

```
procedure parcours_largeur(graphe G, entier x, vect_booleens ref M)

 variables
    entier    y, i
    file    f

 debut
    f ←filevide
    f ←enfiler (x, f)
    M[x] ← vrai
    tant que non estvide(f) faire
        x ← premier(f)
        f ← defiler(f)
        pour i ← 1 jusqu'a d°+(x,G) faire
            y ← i ème-succ-de x dans G
            si non M[y] alors
                M[y] ← vrai
                f ← enfiler(y, f)
            fin si
        fin pour
    fin tant que
fin
```