9 – Parcours de graphes

Comme pour les arbres binaires, il est souvent nécessaire de parcourir un graphe pour ”traiter” les sommets.
Traiter les sommets peut servir à trouver des chemins, calculer des distances minimales, etc.
Dans toute cette partie nous supposerons que les graphes sont ***connexes***, c'est-à-dire ”en un seul morceau” : on dit qu'un graphe est connexe lorsqu'il est toujours possible, étant donnés deux sommets A et B, de trouver un chemin reliant A à B.
Dans les algorithmes de parcours de graphes, il faut marquer les sommets afin de se souvenir qu'ils sont déjà visités.

1. Parcours en largeur

Le parcours en largeur d'un graphe connexe à partir d'un sommet S consiste à traiter tous ses voisins, puis les voisins de ces voisins, etc. Il utilise une file.

Algorithme 1 : Parcours en largeur d'un graphe

```
F ← file vide
enfiler S sur F
marquer S
tant que F est non vide faire
    U ← défiler F
    traiter U
    pour tous les V voisins de U faire
        si V n'est pas marqué alors
            enfiler V sur F
            marquer V
```

2. Parcours en profondeur itératif

Le parcours en profondeur d'un graphe connexe s'écrit presque comme le parcours en largeur, en remplaçant la liste pas une pile. Il consiste à aller explorer le graphe le plus loin possible du point de départ d'abord, puis à revenir en arrière ensuite si besoin.

Algorithme 2 : Parcours en profondeur d'un graphe

```
P ← pile vide
empiler S sur P
marquer S
tant que P est non vide faire
    U ← dépiler P
    traiter U
    pour tous les V voisins de U faire
        si V n'est pas marqué alors
            empiler V sur P
            marquer V
```

3. Parcours en profondeur récursif

La récursivité permet d'écrire un algorithme plus court du parcours en profondeur.

Algorithme 3 : Parcours en profondeur récursif d'un graphe à partir d'un sommet S

```
Fonction ParcoursProfondeur (S)
    traiter S
    marquer S
    pour tous les V voisins de S faire
        si V n'est pas marqué alors
            ParcoursProfondeur(V)
        fin
    fin
fin
```
