Parcours de graphes

Le but est d'explorer les sommets d'un graphe G de proche en proche à partir d'un sommet S de départ. La plupart des algorithmes utilisés sur les graphes sont basés sur un tel parcours. Les n sommets du graphe seront identifiés par des entiers compris entre 0 et $n-1$. Lors du parcours, les sommets auront un des trois états (ou couleurs) suivants :

- non découvert (BLANC)
- découvert (ou à traiter) (GRIS)
- traité (ou marqué) : tous les voisins du sommet ont été découverts (NOIR)

Le marquage des sommets est indispensable pour ne pas tourner en rond indéfiniment dans le graphe s'il contient des cycles : c'est la différence majeure avec les arbres.

1 Parcours en largeur (Breadth First Search)

1.1 Principe

Le parcours en largeur consiste à découvrir d'abord tous les voisins d'un sommet avant de découvrir les voisins de ses voisins, etc. L'algorithme est fondamentalement *impératif* et utilise une *file* de traitement notée `f`, qui contiendra tous les sommets GRIS.

On initialise l'algorithme en plaçant le sommet de départ S (source) dans la file, tous les autres sommets sont BLANCS. A chaque itération, on *défile* pour récupérer le sommet à traiter, on place tous ses voisins non découverts ou traités dans la file `f`, puis on le marque comme traité (il est alors NOIR).

L'algorithme s'arrête quand la file `f` est vide : il ne reste plus aucun sommet à traiter.

1.2 Avec des listes d'adjacence

Le graphe qu'on prendra en exemple dans toute la suite est le suivant :

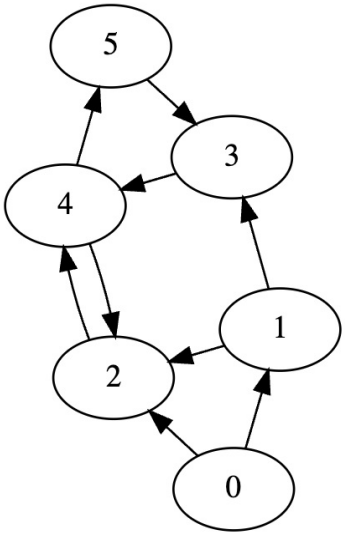

Figure 1. Un exemple de graphe : chaque sommet est identifié par un entier unique.

Définir le dictionnaire `G_adj` (type `dict`) qui va contenir les listes d'adjacence de chaque sommet du graphe. Les clés du dictionnaire seront les entiers identifiant les sommets et les valeurs associées les listes des identifiants de leurs voisins.

Écrire ensuite une fonction `bfs_li(G:dict, S:int)->[int]` qui retourne la liste `t` des sommets traités, et la tester sur l'appel l'exemple précédent avec `0` comme sommet de départ. Vérifier l'ordre de parcours obtenu en faisant tourner l'algorithme à la main (écrire les états de `f` et `t` à chaque appel).

Améliorer la fonction précédente en stockant les états (couleurs) des sommets dans un dictionnaire. En quoi est-ce une amélioration ? Cette fois la fonction *affichera* les sommets traités au fur et à mesure et ne retournera rien.

1.3 Avec une matrice d'adjacence

Définir la matrice d'adjacence `G` du graphe précédent en tant que tableau `numpy` (faire l'import nécessaire) contenant des booléens (`dtype=bool`) :

- Si l'arc (ij) existe, `G[i, j] = True`
- Sinon `G[i, j] = False`

Remarque. Bien sûr on peut créer la matrice soit-même, mais il est aussi possible d'écrire une fonction qui prend en argument un dictionnaire de listes d'adjacence et qui retourne la matrice d'adjacence correspondante (à vous de voir).

Écrire d'abord une fonction `voisins(P:int, G:np.array)-> [int]` qui retourne la liste des voisins (la liste d'adjacence) du sommet `P` dans le graphe `G`. En déduire ensuite une fonction `bfs_mat(G:np.array, S:int)->None` et vérifier qu'elle est correcte par l'appel `bfs_mat(G, 0)`.

2 Parcours en profondeur (Depth First Search)

2.1 Principe

Si on considère le graphe comme un labyrinthe, ce parcours consiste à aller le plus loin possible tant qu'on le peut et à rebrousser chemin dans le cas d'une impasse (un sommet dont tous les voisins sont GRIS ou NOIRS, ou encore un sommet sans voisins).

Ce parcours est fondamentalement *récursif* : on appelle la fonction de parcours sur tous les voisins d'un sommet qui elle-même s'appelle sur les voisins de celui-ci, etc.

2.2 Version récursive

Écrire une fonction `dfs_li` affiche les sommets lorsqu'ils sont traités. Cette fonction définira une fonction `dfs` interne récursive qui implémentera concrètement le parcours. L'imbrication des deux fonctions permet de définir dans `dfs_li` le dictionnaire des couleurs qui sera alors accessible lors de tous les appels de la fonction `dfs` (comme une variable globale à tous les appels).

2.3 Version impérative : utilisation d'une pile

Une fonction récursive utilise de fait la pile d'appel (call stack). On peut simuler ce comportement en écrivant une version impérative du parcours DFS qui utilise une *pile* à la place de la file du parcours BFS. En déduire une deuxième version (impérative) de la fonction précédente. Que remarque-t-on sur l'ordre obtenu pour les sommets traités ? Proposer une explication.