Leçon d'informatique : DFS récursif, application au labyrinthe

S. Benlhajlahsen - PCSI$_1$

Sommaire

Un **parcours de graphe** est une stratégie d'exploration d'un graphe où l'on passe d'un sommet à l'autre, de proche en proche. Cela peut permettre de :

- trouver un chemin de l'entrée vers la sortie dans un labyrinthe,
- trouver le plus court chemin vers une destination par GPS,
- vérifier que tous les les sommets sont accessibles à partir d'un point de départ donné,

On va concentrer sur le **parcours en profondeur**, qu'on appellera DFS (*Depth-First Search*) dans la suite. L'idée est, partant d'un sommet donné, de s'éloigner le plus possible du départ et ne faire demi-tour que quand on arrive à une impasse. Donc on explore **en profondeur** le graphe.

I DFS récursive

I.A Présentation

Prenons un graphe tout simple, celui de la figure 1.

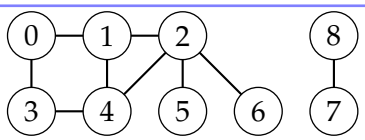

FIGURE 1 – Exemple de graphe.

On part du sommet 0 (`depart`).L'objectif est d'obtenir la liste de tous les sommets accessibles depuis cette position. On constate qu'il y a deux sommets voisins, 1 et 3.

- On sélectionne le sommet 1 (exemple arbitraire), on appelle un assistant et on l'envoie vers le sommet 1. L'objectif étant d'obtenir la liste de tous les sommets accessibles depuis le sommet 1.
- Quand il revient avec sa réponse, on l'envoie vers le sommet 3 avec la même mission.

En l'état, l'assistant va tourner en rond. Quand il sera sur le sommet 1, il va voir le sommet 0 comme un voisin, donc y envoyer un second assistant, qui va recommencer ce que on est déjà en train de faire.
Pour éviter cela, il ne faut pas oublier de « marquer » le sommet comme déjà-vu (un booléen qui passe de `False` à `True`). Ainsi, si un autre assistant plus tard repasse sur le même sommet, il constate qu'il est déjà marqué et ne recommence pas le travail. **C'est la condition d'arrêt de la récursivité**.

I.B Implémentation

Choisissons de représenter le graphe par sa liste d'adjacence. Ainsi, pour celui de la figure 1, cela donne :

```
1 adjacence = [ [1, 3],  [0, 2, 4],  [1, 4, 5, 6], [0, 4],
2     [1, 2, 3], [2], [2], [8], [7] ]
```

Pour « marquer » un sommet déjà visité, on lui attribue un booléen. On va donc gérer une liste de n booléens avec n le nombre de sommets. Initialement, ils sont tous à `False`. Toujours sur notre exemple :

```
n = len(adjacence)
dejavu = [False for i in range(n)]
```

On peut maintenant implémenter le parcours :

```
def dfs(graphe, depart, dejavu):
    if dejavu[depart]==False:
        dejavu[depart] = True
        for voisin in graphe[depart]:
            dfs(graphe, voisin, dejavu)
```

Remarque : C'est un algorithme récursif sans `return`. Dans ce cas, la liste `dejavu` sera une variable globale et sera modifié par l'algorithme. Comme pour les tris, on ne crée pas une deuxième liste à côté. C'est une modification en-place et non externe.

À chaque appel, la fonction reçoit en entrée :

- la liste d'adjacence `graphe`, qui ne change jamais,
- le point de départ `depart`, qui change à chaque appel (ce sont les « assistants » envoyés vers les divers sommets),
- la liste des sommets `dejavu`, qui est ainsi partagée par tous les appels récursifs.

Si on envoie un assistant vers un sommet non visité (ligne 2), alors il va marquer ce sommet en y arrivant (ligne 3), regarder tous les sommets voisins (ligne 4) et y envoyer d'autres assistants (ligne 5).
La fonction ne renvoie rien, mais pour certaines applications on pourra la modifier afin qu'elle renvoie une information utile. Notons tout de même que la liste `dejavu` est modifiée par la fonction, donc son état après exécution contient déjà des informations utiles…

Remarque : Dans l'exemple précédent, si on part de 0, on explore tous les sommets dans l'ordre sauf les sommets 7 et 8. À titre indicatif, on parcourt les sommets dans l'ordre :

$$1 \to 0 \to 2 \to 1 \to 4 \to 1 \to 2 \to 3 \to 0 \to 4 \to 5 \to 2 \to 6 \to 2 \to 4 \to 3$$

I.C Une optimisation qui ne coûte pas cher

La fonction `dfs` ci-dessus montre bien la structure de l'algorithme, mais elle a un défaut évident au niveau de la boucle sur les voisins : si un voisin est déjà dejavu, on appelle quand même `dfs` dessus, provoquant un appel récursif inutile. Donc :

```
def dfs(graphe, depart, dejavu):
    if dejavu[depart]==False:
        dejavu[depart] = True
        for voisin in graphe[depart]:
            if dejavu[voisin]==False:
                dfs(graphe, voisin, dejavu)
```

Sur l'exemple, on passe d'une quinzaine d'appels récursifs dans la première version à exactement 7 (le nombre de sommets de la partie connexe contenant le sommet 0).

Remarque : Dans cet exemple, on explore les sommets dans l'ordre : $1 \to 2 \to 4 \to 3 \to 5 \to 6$.

II Exemples d'utilisation

II.A Test de connexité

Sur l'exemple, la liste `dejavu` vaut, après exécution :

```
[True, True, True, True, True, True, True, False, False]
```

Les sommets 7 et 8 sont restés à `False`, ce qui veut dire qu'ils ne sont pas accessibles à partir de 0 (ce qui était bien sûr évident à l'œil nu sur ce petit graphe).

À retenir : S'il reste des sommets non marqués après une DFS, cela veut dire que le graphe est non connexe.

```
def est_connexe(graphe):
    dejavu = [False]*len(graphe)
    dfs(graphe, 0, dejavu)
    for v in dejavu:
        if  v==False:
            return False
    return True
```

II.B Trouver la sortie !

Si, à l'issu de la DFS, le sommet de sortie a été marqué, alors c'est qu'il existe un chemin vers elle. Par exemple, modélisons le labyrinthe de la figure 2 par le graphe de la figure 3 où les sommets sont les cases du labyrinthe, deux sommets adjacents étant reliés par une arête s'il n'y a pas un mur entre eux.

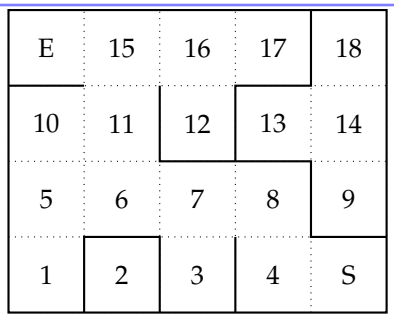

FIGURE 2

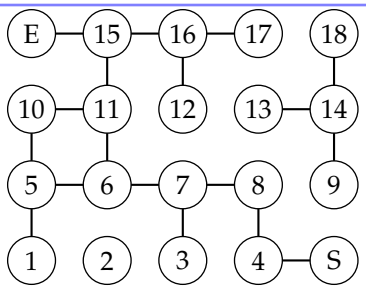

FIGURE 3

FIGURE 4 – À gauche, le labyrinthe. À droite, le graphe associé.

Charge à vous d'écrire la liste d'adjacence, par exemple en numérotant comme 0 l'entrée et 19 la sortie. Après exécution de la fonction `dfs`, le code suivant permet de conclure :

```
entrée = 0
sortie = 19
dfs(labyrinthe, entrée, dejavu)
if dejavu[sortie] == True:
    print("Sortie accessible !")
```

En l'état, la fonction indique que le chemin existe, mais elle n'en donne pas le détail. C'est une question pour plus tard.

Exercice 1 : Écrire à la main le chemin donné par un dfs pour le labyrinthe de la figure 2.

III Variante : avec matrice d'adjacence

L'algorithme ne change pas avec le mode de représentation du graphe, mais les détails de son implémentation oui. Restons sur le graphe de la figure 1. Sa matrice d'adjacence est :

```
adjacence = [
    [0, 1, 0, 1, 0, 0, 0, 0, 0],
    [1, 0, 1, 0, 1, 0, 0, 0, 0],
    [0, 1, 0, 0, 1, 1, 1, 0, 0],
    [1, 0, 0, 0, 1, 0, 0, 0, 0],
    [0, 1, 1, 1, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 1],
    [0, 0, 0, 0, 0, 0, 0, 1, 0]
]
```

On peut garder le mécanisme de marquage avec la liste `dejavu`, donc la seule différence notable est la boucle sur les voisins.

```
def dfs(graphe, depart, dejavu):
    if dejavu[depart]==False:
        dejavu[depart] = True
        for k in range(n):
            if graphe[depart][k] == 1 and dejavu[k]==False:
                dfs(graphe, k, dejavu)
```

Pour un sommet `s`, `graphe[s]` est la s^e ligne de la matrice : c'est donc une liste dont le k^e élément vaut 1 si `k` est un voisin de `s`, 0 sinon. Donc on traite (récursivement) un sommet si c'est réellement un voisin, et s'il est non marqué.
L'utilisation de cette fonction est par ailleurs identique.