Informatique

TP 7

Parcours de graphes

Instructions

Les questions marquées d'astérisques ($*$ ou $**$) pourront être traitées après toutes les autres.

1 Introduction

Ce TP concerne l'exploration de graphes en suivant des arêtes consécutives.

Définitions. Soit $G = (V, E)$ un graphe. Une **chaîne** (ou un chemin) dans G est une suite non-vide de sommets

$$v_0, v_1, \ldots, v_k$$

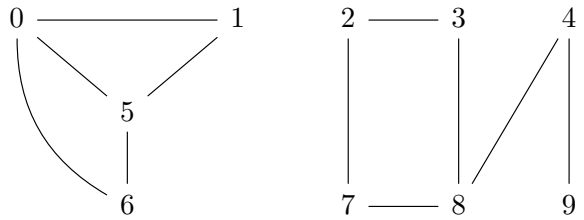
dans laquelle deux sommets consécutifs sont reliés par une arête : pour tout $i < k$, on a $\{v_i, v_{i+1}\} \in E$.

Étant donnés deux sommets $u, v \in V$, une chaîne entre u et v (on dira aussi « chaîne de u à v », ou de manière équivalente « chaîne de v à u ») est une chaîne $v_0, \ldots, v_k$ telle que $v_0 = u$ et $v_k = v$. On dira que v est **accessible** depuis u lorsqu'il existe une chaîne entre u et v.

Une chaîne $v_0, \ldots, v_k$ est de **longueur** k. En d'autres termes, une chaîne de $k + 1$ sommets est de longueur k : on compte le nombre d'arêtes.

Notons qu'une chaîne de longueur 0 n'est constituée que d'un seul sommet. Ainsi, avec ces conventions, pour tout sommet v de G il existe une chaîne (de longueur 0) entre v et lui même.

Exemple 1.1. *Considérons le graphe*



Dans ce graphe,

$$0, 6 \qquad 0, 5, 6 \qquad \textit{et} \qquad 0, 1, 5, 6$$

sont des chaînes entre les sommets 0 *et* 6. *Il n'y a pas de chaîne entre les sommets* 1 *et* 2.

Question 1. *Donner une chaîne entre les sommets* 9 *et* 2 *dans le graphe de l'Exemple 1.1.* Q.1

Parcours de graphes. Soit $G = (V, E)$ un graphe, et soit $u \in V$ un sommet de G. Un parcours de G à partir de u cherche à explorer les sommets de G qui sont accessibles depuis u.

L'idée principale est, à partir d'un sommet v, d'obtenir la liste des voisins de v, et de continuer le parcours à partir de ces voisins. Il existe deux familles principales de parcours de graphes :

— les parcours « en profondeur d'abord »,

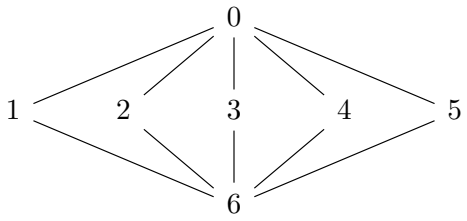
— les parcours « en largeur d'abord ».

Ces familles diffèrent selon l'ordre dans lequel on explore les voisins d'un sommet donné.

Dans tous les cas, un parcours de graphe doit d'une manière ou d'une autre se souvenir des sommets vus au fur et à mesure du parcours. Pour ce faire, dans les parcours d'un graphe $G = (V, E)$ avec $V = \{0, \dots, n-1\}$, on utilisera une liste `visite` de longueur n, initialisée à `False`, de telle sorte que `visite`$[u]$ prend la valeur `True` quand le sommet u est vu lors du parcours.

Le graphe de la question suivante est un exemple intéressant pour comparer les différents parcours de graphes.

Question 2. *Donner une matrice d'adjacence et une liste d'adjacence pour le graphe* Q.2



2 Parcours en « profondeur d'abord »

L'idée d'un parcours en « profondeur d'abord », est, lorsqu'on visite un sommet u, d'obtenir la liste des voisins de u, et de visiter immédiatement chacun des sommets de cette liste. On cherche ainsi à « descendre » dans le graphe dès que c'est possible en suivant la relation d'adjacence.

Par exemple, dans le graphe de la Question 2, en partant du sommet 0, on visite n'importe quel sommet v parmi $1, 2, 3, 4, 5$, pour ensuite visiter le sommet 6 **avant n'importe quel sommet de** $\{1, 2, 3, 4, 5\} \setminus \{v\}$**.**

2.1 Version récursive

Soit $G = (V, E)$ un graphe, et soit $u \in V$ un sommet de G. Un parcours récursif de G à partir de u en « profondeur d'abord » procède comme suit :

— pour chaque voisin v de u, on parcourt récursivement G en « profondeur d'abord » à partir de v.

Question 3. *Pour chacune des deux représentations de graphes (par listes d'adjacence et par matrices d'adjacence), écrire une fonction* Python **récursive** *qui implémente le parcours en « profondeur d'abord » décrit ci-dessus.* Q.3

Ces fonctions doivent prendre en arguments un graphe G et deux sommets u, v de G. Elles doivent renvoyer `True` *s'il existe une chaîne entre u et v dans G, et renvoyer* `False` *sinon.*

Pour chacune de ces deux fonctions, donner le nombre d'opérations en fonction du nombre de sommets et du nombre d'arêtes de G.

Ordre de visite. Afin d'expliciter l'ordre d'exploration des sommets lors d'un parcours en « profondeur d'abord », nous allons légèrement modifier nos fonctions de la Question 3.

Question 4. *Modifiez vos fonctions de la Question 3 de manière à obtenir des fonctions qui prennent en arguments un graphe $G = (V, E)$ et un sommet $u \in V$ de ce graphe. Ces fonctions doivent renvoyer une liste* `rang` *qui à chaque sommet de G associe son rang d'apparition dans le parcours en « profondeur d'abord » de G à partir de u, avec* `rang`$[u] = 0$ *et* `rang`$[v] =$ `None` *si le sommet v n'est pas accessible depuis u.*

Donner l'ordre de parcours ainsi obtenu sur le graphe de la Question 2 (à partir du sommet 0) et sur le graphe de l'Exemple 1.1 (à partir du sommet 2). Q.4

2.2 Version itérative

L'implémentation **itérative** du parcours en « profondeur d'abord » est un modèle pour de nombreux autres parcours de graphes. Une (bonne) implémentation itérative du parcours en « profondeur d'abord » doit effectuer un nombre d'opérations linéaire en la taille du graphe (rappelons que la taille d'un graphe $G = (V, E)$ est $|V| + |E|$).

Nous allons utiliser une structure de données additionnelle `stack`. À chaque itération du parcours, nous allons mettre dans `stack` (certains) des voisins du sommet courant. Afin de garder l'ordre d'exploration d'un parours en « profondeur d'abord », il faut que les sommets ajoutés en dernier dans `stack` soient inspectés en premier. La structure de données adaptée à ce comportement est celle de **pile** (« stack » en Anglais). Les listes Python sont équipées d'opérations adéquates pour les utiliser comme des piles. [1]

Exemple 2.1. *Dans la console* Python *:*

```
>>> x = list(range(5))
>>> x
[0, 1, 2, 3, 4]
>>> x.pop()
4
>>> x
[0, 1, 2, 3]
>>> x.append(10)
>>> x
[0, 1, 2, 3, 10]
>>> x.pop()
10
>>> x
[0, 1, 2, 3]
```

Question 5. *Écrire une fonction* Python *qui prend arguments un graphe G et deux sommets u, v de G, et qui implémente le parcours en « profondeur d'abord »* **itératif**. *Le graphe G est représenté par une* **liste d'adjacence**. *La fonction doit renvoyer* `True` *s'il existe une chaîne entre u et v dans G, et renvoyer* `False` *sinon.* **Le nombre d'opérations doit être linéaire en la taille de G.** Q.5

Question 6. *En effectuant des modifications analogues à celles de la Question 4, vérifiez sur l'exemple de la Question 2 que votre parcours est bien en « profondeur d'abord ».* Q.6

1. Voir https://docs.python.org/3/tutorial/datastructures.html#using-lists-as-stacks.

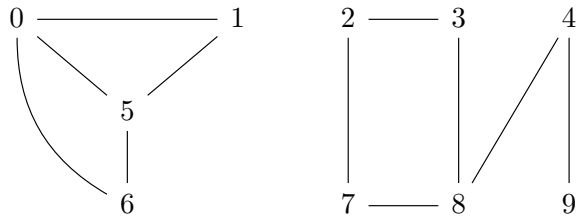
3 Parcours en « largeur d'abord »*

Les parcours en « largeur d'abord » sont un peu plus subtils que les parcours en « profondeur d'abord ». Nous ne verrons qu'une version itérative.

3.1 Préliminaires

Définition. Soit $G = (V, E)$ un graphe (simple). La **distance** $\delta(u, v)$ entre deux sommets $u, v \in V$ de G est, lorsqu'elle existe, la longueur du plus court chemin entre u et v. Notons que pour tout sommet u de G, on a $\delta(u, u) = 0$.

Exemple 3.1. *Considérons le graphe de l'Exemple 1.1.*

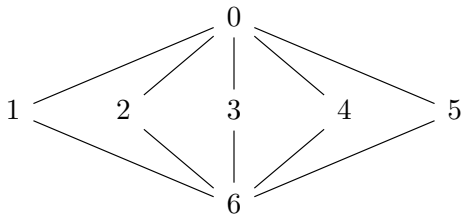


Dans ce graphe, la distance entre les sommets 1 *et* 6 *est* 2*. La disance entre les sommets* 2 *et* 4 *est* 3*. La distance entre les sommets* 1 *est* 2 *n'est pas définie.*

Question 7. *Donner la distance entre les sommets* 0 *et* 6 *dans le graphe ci-dessus.* Q.7

Idée du parcours en « largeur d'abord ». L'idée du parcours en « largeur d'abord » est, lorsqu'on visite un sommet u, de visiter **tous** les voisins de u, **avant** de visiter un nouveau voisin d'un voisin de u. Autrement dit, lorsqu'on parcourt un graphe en « largeur d'abord » à partir d'un sommet u, on visite d'abord tous les sommets à distance 1 de u, puis tous les sommets à distance 2 de u, etc. Ainsi, on « descend » dans graphe par « strates » de plus en plus profondes.

Exemple 3.2. *Considérons le graphe la Question 2.*



Un parcours en « largeur d'abord » de ce graphe à partir du sommet 0 *doit visiter* **tous** *les sommets parmi* $1, 2, 3, 4, 5$ *(dans un ordre quelconque)* **avant** *de visiter le sommet* 6*.*

Les « files » en Python. L'implémentation itérative du parcours en « largeur d'abord » est très similaire à l'implémentation itérative du parcous en « profondeur d'abord ». Il y a cependant une différence fondamentale. Au lieu d'utiliser une structure additionnelle `stack`, nous allons utiliser une structure additionnelle `file`. De manière similaire au parcous en « profondeur d'abord », à chaque itération nous allons mettre dans `file` certains des voisins du sommet courant. Cependant, l'ordre de visite du parcours en « largeur d'abord » impose que les sommets ajoutés en premier dans `file` en soient retirés en premier. En effet, on doit continuer autant que possible à une profondeur donnée avant de descendre plus profond.

La structure de données adaptée à ce comportement est celle de **file** (pour « file d'attente »). Ces « files » sont aussi appelées listes **fifo** (pour « first in, first out » en Anglais). Les listes Python sont équipées d'opérations qui permettent de les utiliser comme des files, mais avec un coût qui en réalité n'est pas adapté à cette utilisation. [2]

Heureusement, le module `deque` implémente des listes adéquates pour être utilisés comme files. [3]

Exemple 3.3. *Dans la console* Python *:*

```
>>> from collections import deque
>>> x = deque()
>>> x
deque([])
>>> x.append(10)
>>> x.append(4)
>>> x.append(0)
>>> x
deque([10, 4, 0])
>>> x.popleft()
10
>>> x
deque([4, 0])
>>> x.append(5)
>>> x.popleft()
4
>>> x
deque([0, 5])
```

3.2 Implémentation Python

Question 8. *Écrire une fonction* Python *qui prend argument un graphe G et deux sommets u, v de G, et qui implémente le parcours de G en « largeur d'abord ». Le graphe G est représenté par une* **liste d'adjacence**. *La fonction doit renvoyer* `True` *s'il existe une chaîne entre u et v dans G, et renvoyer* `False` *sinon.* **Le nombre d'opérations doit être linéaire en la taille de G.** Q.8

Ordre de visite. Afin d'expliciter l'ordre d'exploration des sommets lors d'un parcours en « largeur d'abord », nous allons légèrement modifier nos fonctions de la Question 8.

Question 9. *Modifiez votre fonction de la Question 8 de manière à obtenir une fonction qui prend en arguments un graphe $G = (V, E)$ et un sommet $u \in V$ de ce graphe. Cette fonction doit renvoyer une liste* `rang` *qui à chaque sommet de G associe son rang d'apparition dans le parcours en « largeur d'abord » de G à partir de u, avec* `rang`$[u] = 0$ *et* `rang`$[v] =$ `None` *si le sommet v n'est pas accessible depuis u.* Q.9

Donner l'ordre de parcours ainsi obtenu sur le graphe de l'Exemple 3.2 (à partir du sommet 0) et sur le graphe de l'Exemple 1.1 (à partir du sommet 2).

2. Ajouter ou retirer un élément ailleurs qu'à la fin d'une liste Python a un coût proportionnel à la longueur de la liste, voir https://wiki.python.org/moin/TimeComplexity.

3. Voir https://docs.python.org/3/tutorial/datastructures.html#using-lists-as-queues.

3.3 Plus courts chemins

Une propriété fondamentale du parcours en « largeur d'abord » est qu'il permet de calculer efficacement la distance entre deux sommets.

*Q.10

Question* 10. *Écrire une fonction* Python *qui prend argument un graphe G et deux sommets u, v de G. Cette fonction doit renvoyer la distance entre u et v si elle est définie, et renvoyer* `None` *sinon.* **Le nombre d'opérations doit être linéaire en la taille de G.**