TP 14 : Parcours en profondeur et applications

Télécharger l'annexe sur le site web. Elle contient le code du parcours en profondeur et des exemples de graphes. Dans ce TP, on ne manipulera des graphes que via listes d'adjacence : un graphe sera représenté par une liste `G` de taille n, avec les sommets de G précisément $[\![0, n-1]\!]$. Pour i un tel entier `G[i]` sera la liste (non ordonnée) des sommets tels que l'arc $i \to j$ est présent.

1 Parcours en profondeur du cours

On rappelle ci-dessous le parcours en profondeur d'un graphe, et on donne également son implémentation en Python (disponible dans l'annexe). Tel quel, il ne renvoie rien, mais peut être facilement modifié. Noter que la fonction `pp` est récursive.

Algorithme 1 : Parcours en profondeur complet

Entrée : Un graphe G donné par listes d'adjacence
`deja_vu`$[v] \leftarrow Faux$ pour tout sommet v;
Fonction *pp(u)* **:**
  `deja_vu`$[u] \leftarrow Vrai$;
  pour *tout voisin v de u* **faire**
    si `deja_vu`$[v] = Faux$ **alors**
      pp(v)
pour *tout sommet i du graphe* **faire**
  si `deja_vu`$[i] = Faux$ **alors**
    pp(i)

```
def parcours_profondeur(G):
    """ G graphe donné par listes d'adjacence.
    Parcours le graphe en profondeur.
    Ne renvoie rien """

    n=len(G)
    deja_vu = [False]*n
    def pp(u):
        deja_vu[u]=True
        for v in G[u]:
            if not deja_vu[v]:
                pp(v)
    for i in range(n):
        if not deja_vu[i]:
            pp(i)
```

2 Applications

2.1 Composantes connexes d'un graphe non orienté

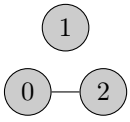

FIGURE 1: Le graphe G_1

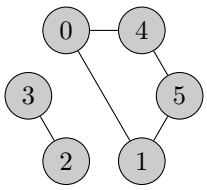

FIGURE 2: Le graphe G_2

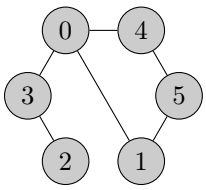

FIGURE 3: Le graphe G_3

Dans un graphe non orienté, chaque « parcours élémentaire » `pp(i)` lancé dans la boucle `for` principale découvre exactement la composante connexe du sommet i.

Exercice 1. Déduire du code du parcours en profondeur, sans regarder le cours, une fonction `liste_cc(G)` calculant la liste des composantes connexes du graphe passé en paramètre, supposé non orienté.

```
>>> liste_cc(G1)
[[0, 2], [1]]
>>> liste_cc(G2)
[[0, 4, 5, 1], [2, 3]]
>>> liste_cc(G3)
[[0, 4, 5, 1, 3, 2]]
```

Exercice 2. Déduire de la fonction précédente une fonction `est_connexe(G)` testant si le graphe passé en paramètre est connexe.

```
>>> est_connexe(G1), est_connexe(G2), est_connexe(G3)
(False, False, True)
```

2.2 Détection de circuit dans un graphe orienté

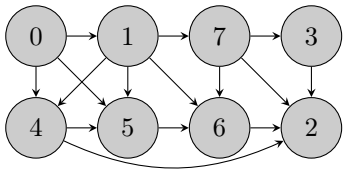

FIGURE 4: Le graphe G_4

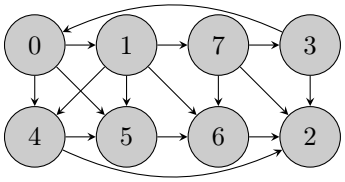

FIGURE 5: Le graphe G_5

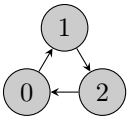

FIGURE 6: Le graphe G_6

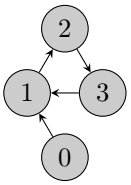

FIGURE 7: Le graphe G_7

Plutôt que d'utiliser une liste de booléens `deja_vu` encodant les états des sommets de manière binaire (découvert / non découvert), on peut choisir d'utiliser trois états :

- blanc pour un sommet non découvert ;
- gris pour un sommet découvert mais dont l'exploration n'est pas terminée (c'est-à-dire un sommet u pour lequel $pp(u)$ est lancé mais ne s'est pas terminé).
- noir pour un sommet dont l'exploration est terminée (un sommet u pour lequel $pp(u)$ a été lancé et son exécution s'est terminée).

On a vu en cours la propriété suivante :

Proposition 1. *Un graphe orienté $G = (V, E)$ est sans circuit, si et seulement si on ne découvre jamais de sommet gris dans la liste d'adjacence d'un sommet u dans l'appel $pp(u)$.*

Exercice 3. Déduire du code du parcours en profondeur, sans regarder le cours, une fonction `possede_circuit(G)` renvoyant un booléen indiquant si `G` possède un circuit. *Indication :* il suffit de faire renvoyer à la fonction `pp` un booléen. Pour les couleurs, on pourra utiliser 0, 1 et 2 pour blanc, gris et noir.

```
>>> possede_circuit(G4), possede_circuit(G5), possede_circuit(G6), possede_circuit(G8)
(False, True, True, True)
>>> possede_circuit(G1) #un graphe non orienté possédant l'arete {u,v} a le circuit u -> v -> u !
True
```

Exercice 4. Recopier et modifier le code précédent en une fonction `calcule_circuit(G)` renvoyant un couple (booléen, liste).

- Si le graphe G ne possède pas de circuit, le couple sera `(False, [])`.
- Si le graphe G possède pas un circuit, le couple sera `(True, c)`, où `c` est la liste des sommets formant un circuit.

```
>>> calcule_circuit(G1)
(True, [0, 2, 0])
>>> calcule_circuit(G5)
(True, [0, 1, 7, 3, 0])
>>> calcule_circuit(G6)
(True, [0, 1, 2, 0])
>>> calcule_circuit(G4)
(False, [])
>>> calcule_circuit(G7)
(True, [1, 2, 3, 1])
>>> calcule_circuit(G8)
(True, [1, 5, 1])
```

Indication : la fonction `pp` renverra un couple (booléen, liste). Si le booléen est `True`, la liste formera les derniers sommets d'un circuit en construction (en sens inverse). Par exemple pour G_5, le circuit détecté (il est essentiellement unique) est $0 \to 1 \to 7 \to 3 \to 0$. L'appel `pp(3)` (celui où le circuit est détecté) renvoie `(True, [0, 3])`, `pp(7)`

renvoie `(True, [0, 3, 7])`, `pp(1)` renvoie `(True, [0, 3, 7, 1])` et enfin l'appel `pp(0)` lancé dans la boucle renvoie `(True, [0, 3, 7, 1, 0])`. Il suffit alors de renverser le chemin. Pour G_7, l'appel `pp(0)` lance `pp(1)`, qui lance `pp(2)`, qui lance `pp(3)`, qui détecte un circuit $(1 \to 2 \to 3 \to 1)$. Attention à ne pas rajouter 0 : pour tester si un circuit en construction est complet, il suffit que le premier sommet soit égal au dernier.

2.3 Cycle dans un graphe non orienté

Détecter un cycle dans un graphe non orienté est très similaire, sauf qu'il faut ignorer les circuits « triviaux », résultant d'une arête parcourue dans les deux sens. On peut pour cela (comme dans le cours), introduire un paramètre `parent` à la fonction `pp`, pour ignorer l'arc `u` $\to$ `parent`. On propose ici une approche différente, permettant de détecter l'existence d'un cycle mais pas d'en calculer un.

Exercice 5.
1. On admet qu'un graphe $G = (V, E)$ non orienté, *connexe* vérifie $|E| \geq |V| - 1$, et il est *sans cycle* si et seulement si il y a égalité. Généraliser en une inégalité similaire reliant $|V|$, $|E|$ et le nombre k de composantes connexes du graphe, avec égalité si et seulement si le graphe est sans cycle.
2. Écrire une fonction `nombre_aretes(G)` calculant le nombre d'arêtes d'un graphe non orienté (rappel : c'est $\frac{1}{2}\sum_{s\in V} \deg(s)$).
3. En déduire une fonction `possede_cycle(G)` testant si G possède un cycle.

```
>>> possede_cycle(G1), possede_cycle(G2), possede_cycle(G3)
(False, True, True)
```

3 Pour aller plus loin : tri topologique et composantes fortement connexes

On travaille désormais uniquement avec des graphes orientés.

3.1 Graphe sans circuit et tri topologique

On peut montrer (c'est dans la feuille d'exercices) qu'un graphe $G = (V, E)$ orienté sans circuit possède un *tri topologique*, qui est une énumération $v_0, \dots, v_{n-1}$ des sommets de V telle que

$$\forall\, i, j \in [\![0, n-1]\!], \qquad (v_i, v_j) \in E \implies i < j$$

Autrement dit, si on représente le graphe « en ligne », avec les sommets dans l'ordre $v_0, \dots, v_{n-1}$, tous les arcs du graphe vont de la gauche vers la droite.

Exercice 6. On peut également montrer que si on effectue un parcours en profondeur du graphe, en stockant les sommets u dans l'ordre où $pp(u)$ termine, on obtient un tri topologique *à l'envers*. En utilisant cette propriété, écrire une fonction `tri_topologique(G)` prenant en entrée un graphe orienté supposé sans circuit, et renvoyant un tri topologique.

```
>>> tri_topologique(G4)
[0, 1, 7, 3, 4, 5, 6, 2]
```

Exercice 7. *À faire en fin de TP si vous avez terminé.* Faire les exercices relatifs au tri topologique dans le TD.

3.2 Composantes fortement connexes d'un graphe orienté

Pour $G = (V, E)$ un graphe orienté, on rappelle que la relation suivante sur V est une relation d'équivalence :

$$u \sim v \quad \iff \quad \text{il existe un chemin de } u \text{ à } v \text{ et un chemin de } v \text{ à } u \text{ dans } G$$

Les classes d'équivalence pour cette relation forment les *composantes fortement connexes* de G. Elles sont plus difficiles à calculer que les composantes connexes d'un graphe non orienté, car un parcours « élémentaire » lancé sur un sommet s découvre la composante fortement connexe de s, mais également les composantes accessibles depuis s. L'algorithme

suivant, dû à Kosaraju, permet le calcul. Il consiste à faire deux parcours en profondeur, l'un sur G, l'un sur ${}^tG$ (le graphe transposé de G, qui est le graphe G où le sens des arcs a été inversé).

Algorithme 2 : Algorithme de Kosaraju pour le calcul de composantes fortement connexes

Entrée : Un graphe G orienté
Sortie : Ses composantes fortement connexes
Effectuer un parcours en profondeur de G, en stockant les sommets dans l'ordre décroissant où $pp(u)$ termine;
Effectuer un parcours en profondeur de ${}^tG$, mais dans la boucle principale du parcours, prendre les sommets en suivant cet ordre. Chaque parcours élémentaire pp lancé dans cette boucle explore découvre exactement une composante fortement connexe.

Exercice 8. *Implémentation de l'algorithme de Kosaraju.*

1. Écrire une fonction `graphe_transpose(G)` calculant le graphe transposé du graphe passé en entrée. Attention à ne pas modifier `G` mais créer un nouveau graphe.

```
>>> G4
[[4, 5, 1], [5, 7, 4, 6], [], [2], [5, 2], [6], [2], [3, 2, 6]]
>>> graphe_transpose(G4)
[[], [0], [3, 4, 6, 7], [7], [0, 1], [0, 1, 4], [1, 5, 7], [1]]
```

2. Implémenter l'algorithme de Kosaraju, sous la forme d'une fonction `liste_cfc(G)`.

```
>>> liste_cfc(G2) #même chose que les composantes connexes pour un graphe non orienté
[[2, 3], [0, 1, 5, 4]]
>>> liste_cfc(G4) #pas de circuit : chaque cfc est réduite à 1 sommet.
[[0], [1], [7], [3], [4], [5], [6], [2]]
>>> liste_cfc(G5)
[[0, 3, 7, 1], [4], [5], [6], [2]]
>>> liste_cfc(G6)
[[0, 2, 1]]
>>> liste_cfc(G8)
[[3, 4], [9], [0], [1, 5, 6, 11, 10], [12, 13], [7], [8, 2]]
```

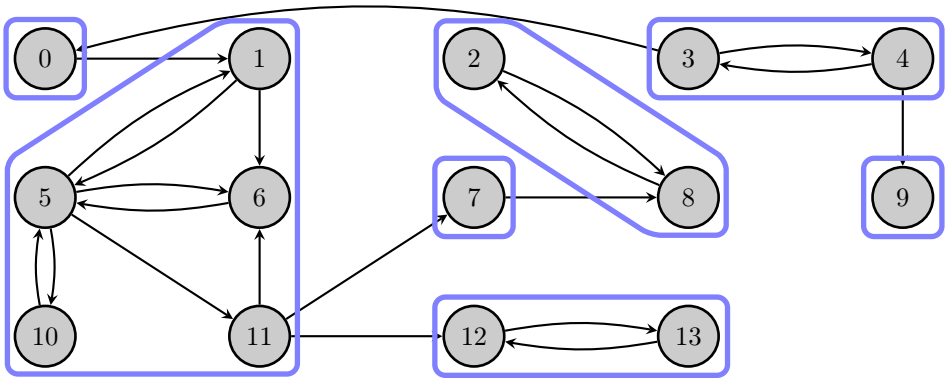

FIGURE 8: Le graphe G_8 et ses composantes fortement connexes