CentraleSupélec

Algorithmique et complexité

TD 1/7 – Parcours de graphes

Ce TD est l'occasion de mettre en œuvre les premiers algorithmes vus en cours, les parcours de graphes.

Exercice 1 : Parcours en largeur et graphes bipartis

Première partie – Profondeur des sommets et cycle de longueur impaire

Question 1

En vous appuyant sur les algorithmes de parcours de graphe vus en cours, proposez un algorithme qui calcule la distance en nombre d'arêtes entre un sommet du graphe (désigné comme racine) et tous les autres sommets (on parle de *profondeur*).

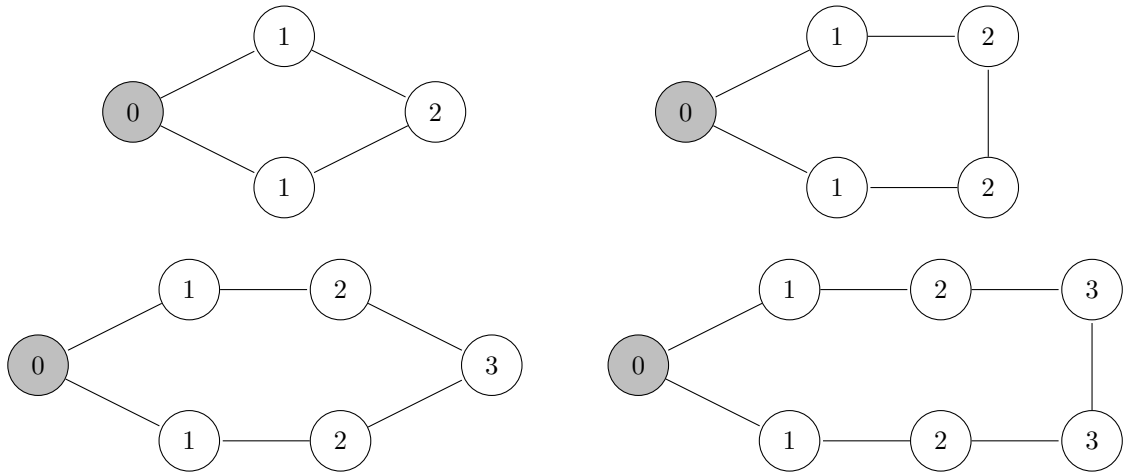
Élements de correction :

Il faut utiliser un parcours en largeur :

```
def BFS_depth(G,s):
    depth = { s: 0 } # dispense de reached
    next = [s]
    while len(next)>0:
      n = pop_begin(next)
      for v in neighbours(n,G):
        if not v in depth:
          add_end(v,next)
          depth[v] = depth[n] + 1
    return depth
```

Question 2

Considérons les cycles suivants de longueur paires et impaires pour lesquels nous avons calculé les profondeurs :



Que remarquez vous (sur les profondeurs) dans les cas des graphes avec un cycle **impair** (en nombre d'arêtes) ? Est-ce une propriété caractéristique ? Énoncez le cas général et donnez une preuve !

Élements de correction :

Propriété caractéristique : Un graphe contient un cycle C avec un nombre impair d'arêtes ssi :

$$\exists (x, y) \in E. \quad depth(x) = depth(y)$$

Preuve :

toutes les arêtes relient des sommets de profondeurs "voisines" :

$$\forall (x, y) \in E. \qquad |depth(x) - depth(y)| \leq 1$$

- $\Rightarrow$ par contraposée, on suppose que $\forall (x, y) \in C.\ depth(x) \neq depth(y)$, alors $\forall (x, y) \in C.\ depth(x) = depth(y) \pm 1$, on aura le long du cycle un nœud de profondeur paire, suivi d'un nœud de profondeur impaire et on alternera ainsi de suite et on ne pourra pas fermer un cycle de taille impaire !
- $\Leftarrow$ si il existe une arête $(x, y) \in E$ pour laquelle $depth(x) = depth(y)$. On considère l'arbre de parcours qui a permis d'annoter les profondeurs. Dans cet arbre x et y ont un premier ancêtre z en commun (éventuellement la racine) à partir duquel on peut former un cycle impair de taille $2(depth(x) - depth(z)) + 1$ en ajoutant l'arête (x, y) au sous-arbre de z.

Question 3

Proposez un algorithme qui détermine si un graphe contient un cycle impair.

Élements de correction :

On lance l'algo d'étiquetage des profondeurs (`BFS_depth` en question 1) puis pour chaque arête du graphe $(x, y) \in E$, il faut tester si x et y sont de même profondeur.

Deuxième partie – Graphe biparti

Un graphe $G = (V, E)$ est dit biparti s'il existe une partition de son ensemble de sommets en deux sous-ensembles $V_1 \subseteq V$ et $V_2 \subseteq V$ telle que chaque arête de E a une extrémité dans V_1 et l'autre dans V_2.

Question 4

Dans un graphe biparti, peut-t-il exister un cycle avec un nombre impair d'arêtes ? est ce une propriété caractéristique ? Justifiez votre réponse.

Élements de correction :

⇒ Si le graphe est biparti, tout chemin alterne entre un sommet de chaque partition donc pour former un cycle et revenir au sommet de départ, il faut nécessairement un nombre pair d'arêtes. Dans un graphe biparti, tous les cycles sont donc pairs.

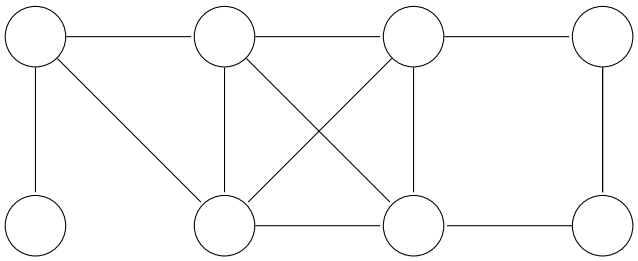
⇐ Il suffit de considérer la partition des sommets de profondeur paire V_1 d'un côté et impaire V_2 de l'autre. Supposons qu'il n'existe pas de cycle impair alors (question 2) :

$$\forall (x, y) \in E. \qquad depth(x) = depth(y) \pm 1$$

Alors toutes les arêtes sont dans $V_1 \times V_2$. Le graphe est donc bien bi-parti.

Question 5

Déduisez un algorithme qui permet de déterminer si un graphe est biparti. Testez votre algorithme sur le graphe suivant. Est-il biparti ? Justifiez votre réponse

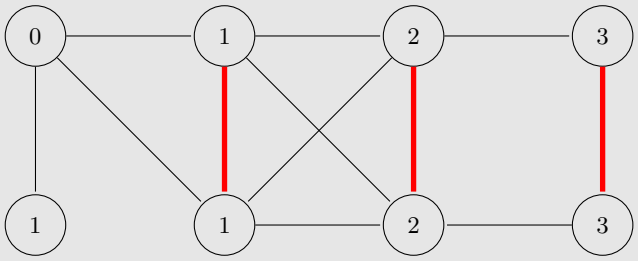


Élements de correction :

Dans les questions précédentes, nous avons montré :

Bi-parti $\Leftrightarrow_{question4}$ Aucun cycle impair $\Leftrightarrow_{question2}$ aucune paire de sommets voisins de même profondeur

Sur le graphe proposé, en partant du sommet en haut à gauche, on obtient les profondeurs suivantes :



Les trois arêtes en gras sont la preuve que le graphe n'est pas biparti.

Exercice 2 : Parcours en profondeur et graphes 2-coloriables

Le coloriage de graphe consiste à attribuer une couleur à chacun de ses sommets de manière à ce que deux sommets reliés par une arête soient de couleurs différentes.

Un graphe 2-coloriable est un graphe pouvant être colorié avec seulement 2 couleurs.

Question 1

Quel est le lien avec l'exercice précédent ? Justifiez votre réponse.

Élements de correction :

Théorème : un graphe est 2-coloriable si et seulement si il est biparti.

$\Longrightarrow$ on partitionne V en $V_1 \cup V_2$ tel que V_1 regroupe tous les noeuds blancs et V_2 tous les noirs ($V_1 \cap V_2 = \emptyset$). Toute arête connecte un noeud blanc (donc dans $V1$) à un noeud noir (dans V_2).

$\Longleftarrow$ il suffit de choisir le blanc pour V_1 et le noir pour V_2 et comme toutes les arêtes sont dans $V_1 \times V_2$, on est bon.

Question 2

Nous souhaitons écrire un algorithme, inspiré du parcours en **profondeur** vu en cours, qui prend en entrée un graphe `G` et qui renvoie un couple `(result,color)` où `result` est `True` si le graphe est coloriable, `False` sinon et `color` est un dictionnaire associant une couleur 0 ou 1 à chaque sommet.

Cet algorithme devra **s'arrêter au plus tôt** lorsque le graphe n'est pas 2-coloriable.

Proposez une version **itérative** et une version **récursive**.

Élements de correction :

Algorithme itératif :

```
def coloring2_iter(G, s):
   color = {s: 0}
   next = [s]
   while len(next)>0:
      n = pop_end(next)
      for v in neighbours(n,G):
        if not v in color:
          add_end(v,next)
          color[v] = 1-color[n]
        elif color[v] == color[n]:
          return False, color
   return True, color
```

Et la version récursive :

```
# recursive function called by coloring2
def coloring2_rec(G,n,color):
  for v in neighbours(n,G):
    if not v in color:
      color[v] = 1-color[n]
      if not coloring2_rec(G,v,color):
        return False
    elif color[v] == color[n]:
      return False
  return True

# the calling function for recursive version
def coloring2(G,s):
   color = {s:0}
   colorable = coloring2_rec(G,s,color)
   return colorable, color
```

Question 3

Est ce que votre algorithme est correct, c'est-à-dire :

1. Quand il retourne un coloriage, ce 2-coloriage est-il correct ?
2. Quand il retourne `False`, est-on certain que le graphe n'est pas 2-coloriable ?

Élements de correction :

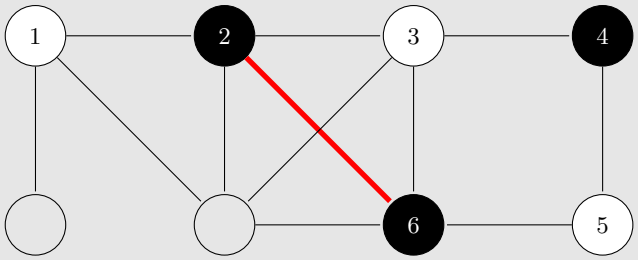
1. Si l'algo trouve un coloriage alors **il a déjà testé toutes les arêtes**
2. Si l'algo ne trouve pas de coloriage, l'algo a trouvé 2 noeuds voisins x et y colorés pareil ! On pourrait imaginer qu'un autre parcours aurait conduit à ne pas colorer ces voisins de la même couleur, et que le graphe est potentiellement coloriable. Il suffit de considérer l'arbre de visite/coloriage réalisé avant l'arrêt de l'algo sur l'arête (x, y). Dans cet arbre, la couleur d'un sommet indique la parité du chemin depuis la racine (la parité de la profondeur dans cet arbre mais pas nécessairement celle du graphe). On raisonne ensuite de manière similaire au raisonnement de la question 2 de l'exercice 1

Question 4

Testez votre algorithme sur le graphe ci-dessus. Est-il 2-coloriable ?

Élements de correction :

On choisit un nœud au hasard (ici en haut à gauche) et on réalise un parcours en profondeur, en annotant les couleurs, on obtient le graphe suivant, les numéros correspondent à l'ordre du parcours :



Ce graphe n'est pas 2-coloriable car on est tombé sur l'arête en gras qui relie 2 sommets de même couleur.

Exercice 3 : Une propriété des DAGs

Prouvez le théoreme suivant :

Dans un graphe orienté sans circuit (c'est à dire un DAG – *Directed Acyclic Graph*) il existe au moins un sommet sans arcs entrants.

Élements de correction :

Soit G un DAG. Supposons que tous ses sommets aient au moins un arc entrant.

1. Soit v un sommet de G. Il a un arc entrant (u, v).
2. On recule, et on considère maintenant le sommet u. Il a un arc entrant (x, u). Si $x = v$, on a un circuit. Sinon, on recule à nouveau.
3. Nous remontons ainsi les sommets jusqu'à croiser un sommet déjà rencontré. Cela signifie la présence d'un circuit, d'où la contradiction.

Cette situation arrive forcément puisque tout sommet a un arc entrant : même si on élimine tous les sommets un par un, arrivé au dernier sommet z, il faudra bien qu'il ait un arc entrant (y, z) avec y un sommet déjà visité !