Numérique et science informatique
Classe de Terminale

Lycée Hoche

année scolaire 2025-2026

Contents

1 Algorithmes de parcours d'un graphe

Algorithme 1 : Parcours en largeur d'un graphe G d'origine s

```
1  pour chaque somment dans G faire
2  |   sommet.couleur ⟵'blanc'
3  |   sommet.parent ⟵ None
4  |   sommet.d ⟵ infini
5  fin
6  s.couleur ⟵ 'gris'
7  f est une file vide
8  f.enfiler(s)
9  tant que f est non vide faire
10 |   u ⟵ f.defiler()
11 |   pour  chaque voisin v dans G.Adj[u] faire
12 |   |   si v.couleur=='blanc' alors
13 |   |   |   v.couleur ⟵ 'gris'
14 |   |   |   v.parent ⟵ u
15 |   |   |   v.d ⟵ u.d + 1
16 |   |   |   f.enfiler(v)
17 |   |   fin
18 |   |   u.couleur ⟵ 'noir'
19 |   fin
20 fin
```

Arbre de parcours en largeur

Le parcours en largeur construit un arbre de parcours en largeur:
Cet arbre a pour racine le sommet s choisi pour origine du graphe .
A chaque découverte d'un sommet blanc v, ligne 12 , dans la liste d'adjacence du sommet défilé u , le sommet v et l'arête (ou l'arc) (u, v) sont ajoutés à l'arbre.(on dit que u est le parent ou le prédécesseur de v dans l'arbre de parcours).
Comme un sommet est découvert au plus une fois , il possède un seul prédécesseur.

Complexité du parcours en Largeur:

L'initialisation dans la boucle `Pour`,lignes 1-5 , requiert un temps en O|S|.

Les opérations de défilement et d'enfilement requièrent un temps constant en O(1).
Puisque chaque sommet est enfilé (puis défilé) une seule fois , au total les opérations dues à la file est en O(|S|).
Pour chaque sommet u défilé , ligne 10 , La boucle `Pour` des lignes 11-17 requiert un temps en O(|Adj[u]|).
Au total le temps consacré au balayage des listes d'adjacences est en O(|A|).

Finalement , on a une complexité en $O(|S| + |A|)$

Distances de plus court chemin:

Pour chaque sommets u, v , on désigne par $\delta(u, v)$ le nombre minimum d'arcs d'un chemin reliant u à v , ou ∞ s'il n'existe pas de chemin de u à v.

Théorème: Le parcours en Largeur appliqué à partir d'un sommet origine s d'un graphe , calcule les distances de plus court chemin $\delta(s, v)$ pour tous les sommets v (autres que s) du graphe.

Exercice 1

Ecrire une fonction `distances(G,s)` qui renvoie la liste des distances d'un sommet origine s à chacun des autres sommets du graphe.

Exercice 2

Ecrire une fonction `Plus_court_chemin(G,u,v)` qui affiche le chemin entre deux sommets u et v d'un graphe.
On donnera une version itérative puis une version récursive.

On donne ci-dessous une version récursive de l'algorithme de parcours en profondeur

Algorithme 2 : Parcours en profondeur d'un graphe G -Version récursive

```
   Data : Un graphe G
   Result : pas de sortie
   /* G.S désigne l'ensemble des sommets du graphe                    */
 1 date ⟵ 0
 2 pour chaque somment u dans G.S faire
 3 |   u.couleur ⟵ 'blanc'
 4 |   u.parent ⟵ None
 5 |   u.debut ⟵ 0
 6 |   u.fin ⟵ 0
 7 fin
 8 pour chaque sommet u ∈ G.S faire
 9 |   si u.couleur ⟵ 'blanc' alors
10 |   |   Visiter-PP(G,u)
11 |   fin
12 fin
```

```
 1 VISITER-PP(G,u)
 2 date ⟵ date +1
 3 u.debut ⟵ date
 4 u.couleur ⟵ 'gris'
   /* On explore les sommets voisins de u                             */
 5 pour  chaque voisin v dans G.Adj[u] faire
 6 |   si v.couleur == 'blanc' alors
 7 |   |   v.parent ⟵ u
 8 |   |   Visiter-PP(G,v)
 9 |   fin
10 fin
11 u.couleur ⟵ 'noir'
12 date ⟵ date+1
13 u.fin ⟵ date
```

Complexité du parcours en profondeur:

La boucle `Pour` des lignes 1-5 requiert un temps en $O(|S|)$.
L'instruction de la ligne 10 est exécutée une fois à chaque étape (une par sommet u).
Pour chaque sommet u , la boucle `Pour` des lignes 11-16 est exécutée |Adj[u]| fois .
Or $\sum |Adj[u]| = |A|$ (nombre d'arêtrs ou arcs) du graphe .

Finalement , on a une complexité en $O(|S| + |A|)$.

Exercice 3

Ecrire une version du parcours en profondeur qui utilise une pile (non récursive).

Algorithme 3 : Parcours en profondeur d'un graphe G avec Pile

```
   Data : Un graphe G , s un sommet origine
   Result : pas de sortie
   /* G.S désigne l'ensemble des sommets du graphe                    */
 1 pour chaque sommet u dans G.S faire
 2 |   u.couleur ⟵ 'blanc'
 3 |   u.parent ⟵ None
 4 fin
 5 P est une Pile vide
 6 s.couleur ⟵ 'gris'
 7 P.empiler(s)
 8 tant que P est non vide faire
 9 |   u ⟵ depiler()
10 |   pour chaque voisin v dans G.Adj[u] faire
11 |   |   si v.couleur == 'blanc' alors
12 |   |   |   v.couleur ⟵ 'gris'
13 |   |   |   v.parent ⟵ u
14 |   |   |   P.empiler(v)
15 |   |   fin
16 |   fin
17 fin
```

Appliquer cet algorithme aux graphes implémentés à l'aide des dictionnaires:

$G7 = \{a : [b, c], b : [a], c : [a, d, e], d : [c], e : [c]\}$
$G9 = \{a : [b, c], b : [a, d, e], c : [a, d], d : [b, c, e], e : [b, d, f, g], f : [e, g], g : [e, f, h], h : [g]\}$

Exercice 4

On rappelle qu'un graphe non orienté est **connexe** s'il existe un chemin entre deux sommets quelconques.

La composante connexe d'un graphe non-orienté G est un sous-graphe G' de G qui est connexe et maximal (c'est-à-dire qu'aucun autre sous-graphe connexe de G ne contient G'). Un graphe est dit connexe si et seulement si , il admet une unique composante connexe.

Dans l'exemple ci-dessous , le graphe a pour composantes connexes [0,2,4] et [1,3]

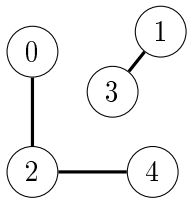


Figure 1

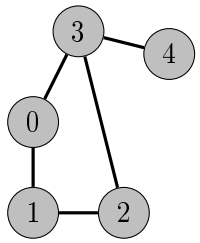
Modifier l'algorithme écrit dans l'exercice 3 , afin qu'il affiche les composantes connexes d'un graphe non orienté.

Exercice 5

Dans cet exercice , on veut réécrire les algorithmes du parcours en largeur et parcours en profondeur mais sans considérer les sommets d'un graphe comme un objet (c'est-à-dire qu'on n'utilise pas la classe Sommets).

Pour cela , on considérera un graphe de n sommets numérotés de 0 à n.

Par exemple pour le graphe G ci-dessous , G sera modélisé par une liste d'adjacences `adj`:



`adj`= [[1,3] , [0,2] , [1,3], [0,2,4] , [3]]

Quelle structure de données peut-on prévoir pour gérer les attributs `couleur` , `parent` et `distance` où `distance` est la distance d'un sommet depuis le sommet origine (en terme de nombres d'arêtes)

Réécrire alors en `Python` les fonctions `parcours_en_largeur` et `parcours_en_profondeur` en utilisant cette modélisation du graphe.

Exercice 6

Pour chercher si un graphe (non orienté) a un cycle , on le parcourt en profondeur et si on trouve au cours de ce parcours un sommet v:

- adjacent au sommet où on se trouve .
- différent du sommet qu'on vient de quitter.
- déjà visité

Il y a cycle.

Ecrire une fonction `detecter_cycle` qui détermine si un graphe non orienté possède un cycle.

Annexe: Utilisation du module `Graphviz` pour l'affichage de graphes

Pour l'affichage d'un graphe , on peut utiliser le module `graphviz`:

```
from graphviz import Digraph #pour les graphes orientés
from graphviz import Graph    #pour les graphes non orientés
```

Pour créer un graphe:

```
graf=Digraph(format='png')
```

Pour créer des noeuds avec des étiquettes:

```
graf=Digraph(format='png')
graf.node('a', 'un')  # 'un' est l'étiquette du noeud 'a' dans
graf.node('b', 'deux') # 'dux' est l'étiquette du noeud 'b'
```

Pour créer une arête entre les noeuds 'a' et 'b':
`graf.edge('a','b')`

pour afficher le graphe `graf`:
`graf.render('test-output/resultat.gv',view=True)`

Pour créer un graphe , on peut également préciser le moteur 'engine'.Par exemple:

```
w = Graph('wide',engine='neato')
```

Et pour l'affichage:

```
w.view()
```