université de BORDEAUX

Année 2018-2019 — Session de décembre 2018
Étape : INF501-MI501 — UE : 4TIN501U
Épreuve de : Algorithmique des graphes
Date : 14 décembre 2018 – 11h30 — **Durée :** 1h30
Épreuve de : F. Kardoš — **Nb. pages : 6**

Aucun document autorisé. Une attention toute particulière sera portée à la présentation et à la rédaction de la copie.

Dans tous les exercices, on désignera par $V(G)$ et $E(G)$ respectivement l'ensemble des sommets et l'ensemble des arêtes (resp. des arcs) d'un graphe non orienté (resp. orienté) G. Les variables n et m désigneront respectivement le nombre de sommets et d'arêtes (resp. d'arcs). Par défaut, dans toutes les structures de données les sommets sont stockés et parcourus dans un ordre alphabétique.

1 Calcul d'une arête la plus lourde

Soit G un graphe non orienté simple. On définit le *poids* d'une arête $e = uv$ comme la somme des degrés de ses extrémités, c-à-d

$$w(uv) = deg(u) + deg(v).$$

Pour chacune des représentations de graphes suivantes – matrice d'adjacence, listes d'adjacence – proposer un algorithme qui retourne une arête de poids maximal dans un graphe donné.

Étudier la complexité des deux algorithmes.

2 Parcours en profondeur

Soit G un graphe orienté dont la matrice d'adjacence est la suivante :

	A	B	C	D	E	F	G	H
A	0	1	0	1	0	1	0	0
B	0	0	1	0	0	0	1	0
C	0	0	0	1	0	0	1	0
D	0	0	0	0	1	0	0	1
E	1	0	0	0	0	1	0	0
F	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0
H	0	0	0	0	1	0	0	0

2.1) Parcourir le graphe G en profondeur. Dans un tableau, expliciter la date de début et de fin de visite ainsi que le père de chaque sommet.

2.2) Dessiner G. Pour chaque arc de G, déterminer son type.

Rappel. Dans un graphe orienté, tout arc uv peut se classifier dans une des quatre classes suivantes :

- Un arc de liaison est un arc reliant un sommet à un de ses fils.
- Un arc de retour est un arc reliant un sommet à un de ses ancêtres.
- Un arc avant est un arc reliant un sommet à un descendant, mais pas un fils.
- Tout autre arc est un arc transverse.

2.3) Le graphe G est-il fortement connexe ? Justifier.

3 Optimisation de flot

La matrice C ci-dessous à gauche est la matrice d'adjacence d'un réseau de flot G : le premier sommet est la source, le dernier le puits ; une valeur strictement positive représente la capacité de l'arc correspondant. La matrice F ci-dessous à droite décrit un flot f sur G.

$$C = \begin{bmatrix} 0 & 5 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 4 & 0 & 2 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 2 & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad F = \begin{bmatrix} 0 & 2 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 2 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

3.1) Dessiner le graphe G en représentant pour chaque arc le flot actuel et la capacité.

3.2) Le flot actuel est-il optimal ? Justifier. Si ce n'est pas le cas, appliquer l'algorithme de Ford-Fulkerson jusqu'à ce que le flot soit optimal. Pour chaque amélioration par Ford-Fulkerson détailler le chemin augmentant et la valeur d'augmentation. Une fois le flot optimal, expliciter une coupe minimale justifiant l'optimalité du flot.

4 Routes pour des camions très hauts

Modélisons un réseau routier par un graphe non orienté, dans lequel chaque arête porte deux informations : le poids $w(uv)$ d'une arête correspond à la distance à parcourir entre les sommets u et v qui représentent deux villes, puis la hauteur $h(uv)$ d'une arête correspond à la hauteur maximale limitée par la présence de ponts et/ou tunnels sur la route entre u et v.

Un camion de hauteur h_c peut emprunter une route entre s et t seulement si pour toute arête e de cette route, on a $h_c \leq h(e)$.

4.1) En modifiant un algorithme d'exploration de graphes (les algorithmes de parcours en largeur et parcours en profondeur sont rappelés en annexe), proposer un algorithme qui résout la question de décision suivante : étant donné un sommet de départ s et la hauteur d'un camion h_c, quel est l'ensemble des sommets accessibles depuis s par ce camion ?

Préciser bien les lignes de code à ajouter ou à modifier.

4.2) En modifiant l'algorithme de Dijkstra (rappelé en annexe), proposer un algorithme qui résout la question d'optimisation suivante : étant donné un sommet de départ s et la hauteur d'un camion h_c, pour tout sommet v de G calculer la longueur minimale d'une route entre s et v que le camion peut emprunter.

4.3) *(Question bonus, à traiter en dernier)*
Considérons maintenant des camions électriques d'une capacité de batterie limitée. On suppose que le camion fait toujours des trajets directs, c'est à dire, sans faire de pause pour recharger la batterie. On suppose aussi qu'à chaque départ, la batterie est pleine. Proposer un algorithme qui résout la question d'optimisation suivante : étant donné un sommet de départ s, un sommet d'arrivée t et la distance d maximale possible à parcourir sans recharger, calculer la hauteur maximale possible d'un camion pour qu'il puisse traverser le réseau de s à t.

Algorithme 1 Parcours en largeur PL(G, s)

```
1: couleur(s) ← GRIS
2: d(s) ← 0
3: pere(s) ← NIL
4: F ← {s}
5: pour tout v ∈ V(G)\s faire
6:    couleur(v) ← BLANC
7:    d(v) ← ∞
8:    pere(v) ← NIL
9: fin pour
10: tant que F non vide faire
11:    v ← tete(F)
12:    pour tout w ∈ Adj(v) faire
13:       si couleur(w) = BLANC alors
14:          couleur(w) ← GRIS
15:          d(w) ← d(v) + 1
16:          pere(w) ← v
17:          Enfiler(F, w)
18:       fin si
19:    fin pour
20:    Defiler(F)
21:    couleur(v) ← NOIR
22: fin tant que
```

Algorithme 2 Parcours en profondeur PP(G)

```
1: pour tout v ∈ V(G) faire
2:    couleur(v) ← BLANC
3:    pere(v) ← NIL
4: fin pour
5: temps ← 0
6: pour tout v ∈ V(G) faire
7:    si couleur(v) = BLANC alors
8:       VisiterPP(v)
9:    fin si
10: fin pour
```

Algorithme 3 VisiterPP(v)

```
1: d(v) ← temps ← temps + 1
2: couleur(v) ← GRIS
3: pour tout w ∈ Adj(v) faire
4:    si couleur(w) = BLANC alors
5:       pere(w) ← v
6:       VisiterPP(w)
7:    fin si
8: fin pour
9: couleur(v) ← NOIR
10: f(v) ← temps ← temps + 1
```

Algorithme 4 Dijkstra(G, w, s)

```
1: pour tout v de V(G) faire
2:    d(v) ← ∞
3:    pere(v) ← NIL
4:    couleur(v) ← BLANC
5: fin pour
6: d(s) ← 0
7: F ← FILE_PRIORITE({s}, d)
8: tant que F ≠ ∅ faire
9:    pivot ← EXTRAIRE_MIN(F)
10:   pour tout e = (pivot, v) arc sortant de pivot faire
11:      si couleur(v) = BLANC alors
12:         si d(v) = ∞ alors
13:            INSERER(F, v)
14:         fin si
15:         si d[v] > d[pivot] + w(e) alors
16:            d[v] ← d[pivot] + w(e)
17:            pere[v] ← pivot
18:         fin si
19:      fin si
20:   fin pour
21:   couleur[pivot] ← NOIR
22: fin tant que
```

Algorithme 5 Ford-Fulkerson(G, s, t, c)

```
1: Marguage()
2: tant que le sommet destination t est marqué faire
3:    Améliorer()
4:    Marquage()
5: fin tant que
```

Algorithme 6 Marquage()

```
1: pour chaque sommet v faire
2:    démarquer(v)
3: fin pour
4: marquer(s)
5: enfiler(s)
6: tant que file non vide faire
7:    u ← tête de la file
8:    pour chaque voisin sortant v de u non marqué faire
9:       si f(uv) < c(uv) alors
10:         marquer(v)
11:         enfiler(v)
12:         π(v) ← uv
13:      fin si
14:   fin pour
15:   pour chaque voisin entrant v de u non marqué faire
16:      si f(vu) > 0 alors
17:         marquer(v)
18:         enfiler(v)
19:         π(v) ← vu
20:      fin si
21:   fin pour
22:   défiler(u)
23: fin tant que
```

Algorithme 7 Améliorer()

```
1: v ← t
2: val ← ∞
3: tant que v ≠ s faire
4:    si π(v) = uv alors
5:       val ← min{val, c(uv) − f(uv)}
6:    fin si
7:    si π(v) = vu alors
8:       val ← min{val, f(uv)}
9:    fin si
10:   v ← u
11: fin tant que
12: v ← t
13: tant que v ≠ s faire
14:   si π(v) = uv alors
15:      f(uv) ← f(uv) + val
16:   fin si
17:   si π(v) = vu alors
18:      f(vu) ← f(vu) − val
19:   fin si
20:   v ← u
21: fin tant que
```