Domaine Sciences et Technologies
Portail 1 René Descartes

Aix*Marseille université

**Introduction à l'informatique : TD 7 bis**
**Code UE : S14IN1I1**

Année 2020–21

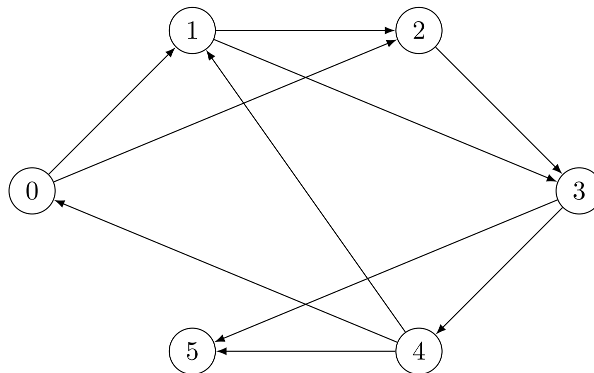
**Parcours de graphes**

**Exercice 1 (Parcours en largeur)** En cours, on a décrit l'algorithme de parcours en largeur dans un graphe orienté $G$ à partir d'un sommet $s$ :

```
fonction parcours_en_largeur(G, s) :
    n := nombre_sommets(G)
    H := graphe_vide(n)          #  graphe sans arcs
    F := ∅                       #  file vide
    couleur := tableau de longueur n rempli de 'blanc'
    couleur[s] := rouge
    enfiler(F, s)
    Tant que F ≠ ∅ faire
        u := défiler(F)
        Pour v de 0 à n−1 faire
            Si (G[u,v] = 1 et couleur[v] = blanc) alors
                couleur[v] := rouge
                H[u,v] := 1
                enfiler(F, v)
            FinSi
        FinPour
    FinTantQue
    retourner(H)                 #  graphe des chemins minimaux
```

Cet algorithme retourne le graphe $H$ qui ne contient que les arcs traversés en parcourant un chemin de $s$ à chacun des autres sommets accessibles.

1. Exécuter l'algorithme de parcours en largeur sur le graphe $G$ représenté ci-dessous à partir du sommet $s = 0$, en montrant toutes les étapes de l'algorithme (avec la coloration des sommets et l'état de la file dans les configurations intermédiaires). Représenter aussi le graphe $H$ retourné par la fonction.




2. Écrire en pseudo-code une fonction `calculer_chemin`$(H, s, t)$ qui prenne en argument le graphe des chemins minimaux $H$ construit par la fonction `parcours_en_largeur`$(G, s)$ et affiche le chemin en $H$ qui commence par le sommet source $s$ et se termine avec le sommet $t$. Pour simplifier, on pourra afficher les sommets du chemin en ordre inverse : par exemple, si le chemin entre $s$ et $t$ est $s \to u \to v \to t$, on pourra afficher « $t$ $v$ $u$ $s$ ».

**Exercice 2 (Plus court chemin)** On a vu en cours un algorithme permettant de calculer des plus courts chemins dans des graphes pondérés. On décrit un graphe (orienté) pondéré à l'aide

d'une matrice (un tableau bi-dimensionnel) d'adjacence $G$ avec autant de lignes et de colonnes qu'il y a de sommets dans le graphe, et dont le coefficient pour $u$ et $v$ deux sommets vaut

$$G[u,v] = \begin{cases} +\infty & \text{s'il n'y a pas d'arcs de } u \text{ à } v \\ \text{poids de } u \to v & \text{sinon} \end{cases}$$

Le poids d'un chemin $u_0 \to u_1 \to u_2 \to \cdots \to u_{n-1} \to u_n$ du graphe (où $u_0 \to u_1$, $u_1 \to u_2$, $\ldots$, $u_{n-1} \to u_n$ sont des arcs du graphe) est égal à la somme

$$G[u_0,u_1] + G[u_1,u_2] + \cdots + G[u_{n-1},u_n] = \sum_{i=0}^{n-1} G[u_i,u_{i+1}]$$

Un plus court chemin de $u$ à $v$ est un chemin menant de $u$ à $v$ dont le poids est minimal parmi tous les chemins possibles.

En cours, nous avons exécuté l'algorithme de Dijkstra. En voici une description sous forme de pseudo-code (noter la ressemblance avec le parcours en largeur, même si nous n'avons plus de graphe $H$ mais deux nouveaux tableaux...) :

```
fonction dijkstra(G, s) :
    n := nombre_sommets(G)
    distance := tableau de taille n rempli de +∞
    prédécesseur := tableau de taille n rempli de ⊥
    F := file de priorité vide
    couleur := tableau de longueur n rempli de 'blanc'
    couleur[s] := rouge
    distance[s] := 0
    insérer_priorité(F, s, 0)
    Tant que F ≠ ∅ faire
        u := extraire_prioritaire(F)
        d := distance[u]
        Pour v de 0 à n−1 faire
            Si (couleur[v] = blanc et G[u,v] ≠ +∞) alors
                couleur[v] := rouge
                prédécesseur[v] := u
                distance[v] := d+G[u,v]
                insérer_priorité(F, v, d+G[u,v])
            Sinon Si (d+G[u,v] < distance[v]) alors
                mettre_à_jour_priorité(F, v, d+G[u,v])
                prédécesseur[v] := u
                distance[v] := d+G[u,v]
            FinSi
        FinPour
    FinTantQue
    retourner(prédécesseur)
```

La file $F$ est désormais une file de priorité, c'est-à-dire que chacun de ses éléments est associé à une priorité (un entier positif ici) :

— on peut insérer un nouvel élément $u$ dans la file, en précisant sa priorité $p \in \mathbf{N}$ à l'aide de la fonction `insérer_priorité(F, u, p)` ;

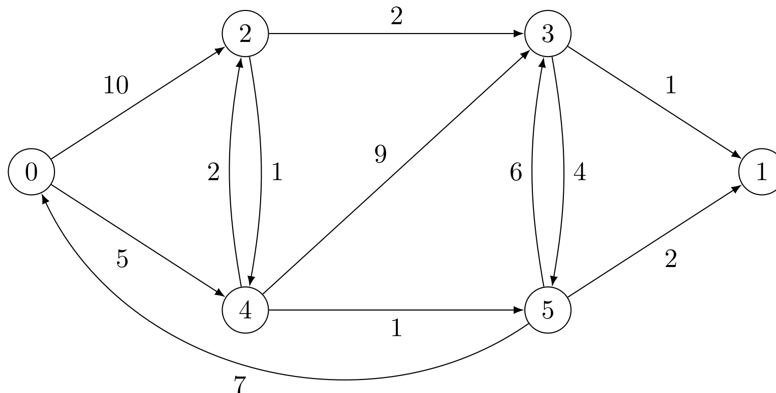
— l'élément prioritaire est celui de **plus petite priorité** : on peut récupérer cet élément à l'aide de la fonction `u := extraire_prioritaire(F)` ;

— on peut mettre à jour la priorité d'un élément $u$ de la file pour lui attribuer la nouvelle priorité $p$ dans $F$, à l'aide de la fonction `mettre_à_jour_priorité(F, u, p)`.

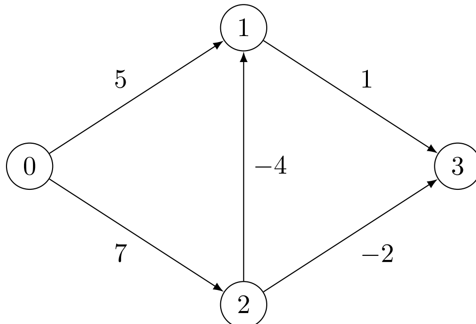
La différence notable entre le parcours en largeur et l'algorithme de Dijkstra réside dans la nécessité de maintenir les estimations des distances (on utilise un tableau `distance` pour cela) et de mettre à jour les arcs rouges : pour cela, plutôt que de maintenir un graphe `H` comme précédemment, on utilise plutôt un tableau `prédécesseur` qui associe à chaque sommet $v$ rouge son prédécesseur dans le graphe `H`, c'est-à-dire l'unique sommet $u$ tel que $(u,v)$ est un arc rouge de

H. Si le sommet est blanc (ou pour la source qui n'a pas de tel prédécesseur), on utilise le symbole $\perp$ pour dénoter l'absence de prédécesseur.

1. Exécuter l'algorithme de Dijkstra sur l'exemple ci-dessous en partant de la source $s = 0$ :




2. En déduire un plus court chemin du sommet 0 au sommet 1.
3. Jusque-là, nous avons étudié uniquement des graphes pondérés avec des poids entiers positifs ou nuls : pourtant, on pourrait imaginer des graphes avec des poids négatifs, par exemple si le poids de l'arc représente des échanges d'argent (vente ou achat de produits). Exécuter l'algorithme de Dijkstra sur l'exemple ci-dessous où plusieurs arcs ont des poids négatifs :




4. Qu'en déduisez-vous sur l'algorithme de Dijkstra ?