Aix*Marseille université

Domaine Sciences et Technologies
Portail 1 René Descartes

Science informatique : TD 2

Année 2021-22

Graphes

Exercice 1

1. Comment sont reliées les valeurs $M_{u,v}$ et $M_{v,u}$ de la matrice d'adjacence M d'un graphe (non orienté), pour tous sommets u et v ?
2. Quelle propriété d'un graphe orienté est représentée par le fait que dans sa matrice d'adjacence M, on a pour tout sommet u, $M_{u,u} = 1$?
3. On peut vérifier qu'une matrice d'adjacence est symétrique à l'aide du pseudo-code suivant :

```
fonction est_symétrique(M):
    n := nombre_sommets(M)  # renvoie le nombre de sommets du graphe
    Pour u de 0 à n−1 faire
        Pour v de 0 à n−1 faire
            Si M[u][v] ≠ M[v][u] alors
                retourner(faux)
            FinSi
        FinPour
    FinPour
    retourner(vrai)
```

   L'algorithme recherche une preuve de non symétrie de la matrice (auquel cas l'algorithme s'arrête en plein milieu en retournant `faux` dès que possible) : s'il n'a pas trouvé de preuve de non symétrie, c'est que la matrice est symétrique et on renvoie donc `vrai`.

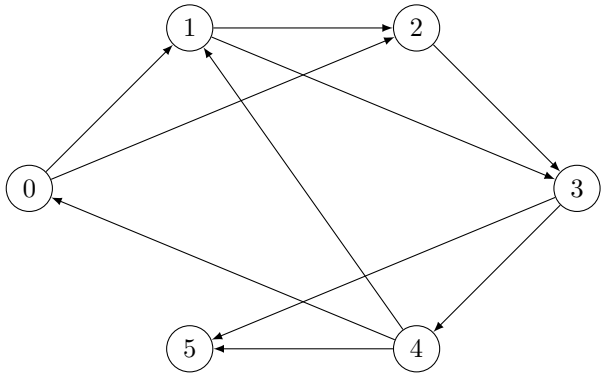
   Noter qu'on fait deux fois trop de travail dans ce code, puisqu'on teste chaque couple de sommets (u, v) deux fois... Comment modifier le code pour faire mieux, en ne testant qu'une seule fois chaque couple ?
4. Écrire un algorithme qui prend en entrée la matrice d'adjacence d'un graphe orienté, et renvoie le nombre d'arcs dans le graphe.
5. Comment modifier votre algorithme pour qu'il compte le nombre d'arêtes d'un graphe non orienté ?

Exercice 2 (Parcours en largeur) L'algorithme de parcours en largeur d'un graphe orienté G permet de visiter tous les sommets accessibles à partir d'un certain sommet s. On peut l'implémenter à l'aide d'une file d'attente des sommets à visiter, ainsi qu'une coloration des sommets visités :

```
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
    insérer(F, s)
    Tant que F ≠ ∅ faire
        u := extraire(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
                insérer(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 H construit par la fonction `parcours_en_largeur`(G, s) et affiche le chemin dans 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 3 (Plus court chemin) Lorsque les graphes sont équipés de poids (sur les arcs), il peut être intéressant de calculer des plus courts chemins entre des paires de sommets. On décrit un graphe (orienté) pondéré à l'aide d'une matrice d'adjacence M 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

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

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

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

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.

Voici une description de l'algorithme de Dijkstra 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
```

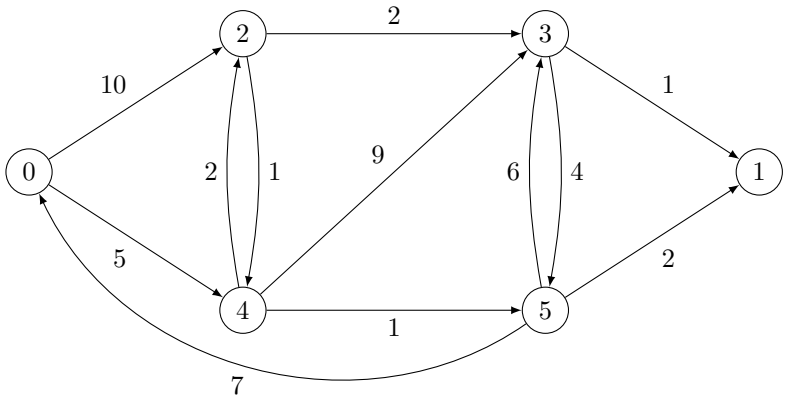
```
        u := extraire_prioritaire(F)
        d := distance[u]
        Pour v de 0 à n−1 faire
            Si (couleur[v] = blanc et G[u][v] ≠ +∞) alors
                 couleur[v] := rouge
                 prédécesseur[v] := u
                 distance[v] := d+G[u][v]
                 insérer_priorité(F, 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) :

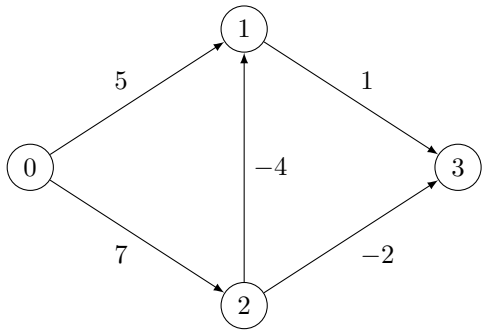
- on peut insérer un nouvel élément u dans la file, en précisant sa priorité $p \in \mathbf{N}$ à l'aide de la fonction `insérer_priorité(F, u, p)` ;
- l'élément prioritaire est celui de **plus petite priorité** : on peut récupérer cet élément à l'aide de la fonction `u := extraire_prioritaire(F)` ;
- 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 :



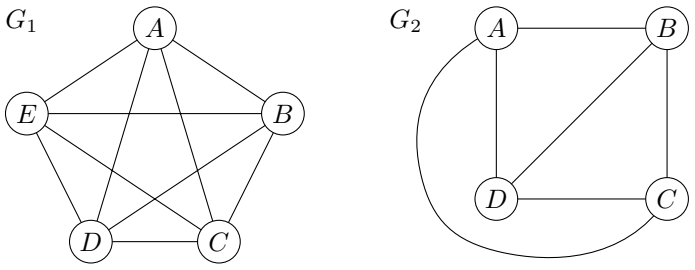
4. Qu'en déduisez-vous sur l'algorithme de Dijkstra ?

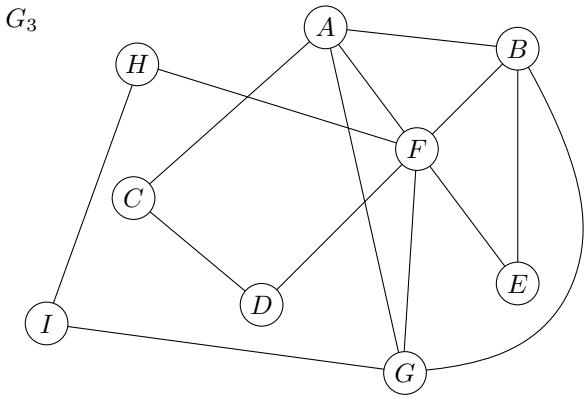
Exercice 4 (Cycles eulériens)

Dans un graphe non orienté, on appelle *cycle eulérien* tout cycle (un chemin qui revient à son point de départ) qui traverse chaque arête du graphe une et une seule fois. La caractérisation suivante existe :

> « Un graphe non orienté connexe admet un cycle eulérien
> si et seulement si chaque sommet est de degré pair. »

1. En utilisant la caractérisation précédente, dites pour chacun des graphes suivants s'ils admettent un cycle eulérien.



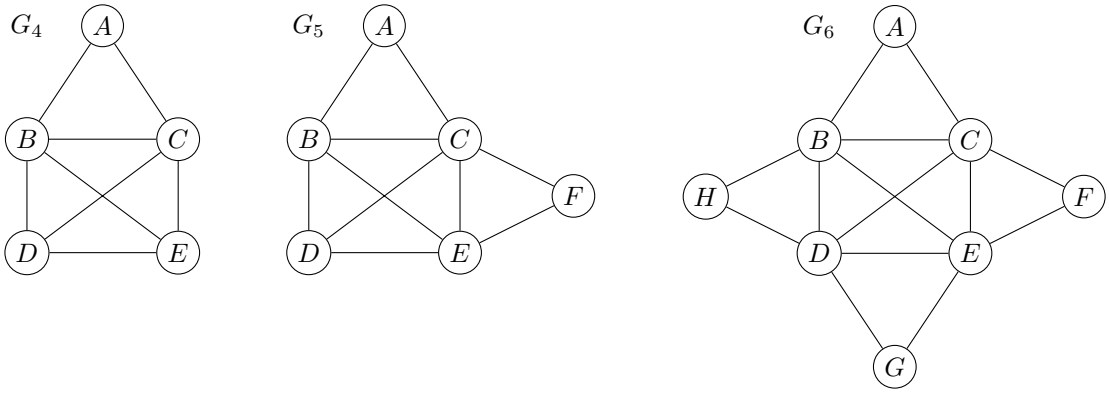


L'algorithme de Hierholzer, décrit de manière informelle ci-dessous, permet de construire un cycle eulérien (s'il en existe un) :

- Choisir n'importe quel sommet initial v
- Suivre un chemin arbitraire d'arêtes jusqu'à retourner à v, obtenant ainsi un cycle c
- **Tant qu**'il y a des sommets u dans le cycle c avec des arêtes qu'on n'a pas encore choisies **faire**
  - Suivre un chemin à partir de u, n'utilisant que des arêtes pas encore choisies, jusqu'à retourner à u, obtenant un cycle c'
  - Prolonger le cycle c par c'

À noter qu'en toute généralité, un graphe eulérien peut admettre plusieurs cycles eulériens différents.

2. Pour les graphes précédents qui admettent un cycle eulérien, trouver un tel cycle en appliquant l'algorithme de Hierholzer.
3. La notion de cycle eulérien peut être étendue : un *chemin eulérien* est un chemin (pas nécessairement un cycle) qui traverse chaque arête du graphe une et une seule fois. Parmi les graphes suivants, quels sont ceux pour lesquels vous pouvez trouver un chemin eulérien ?



4. À partir de vos observations, conjecturer une caractérisation des chemins eulériens en terme de degré des sommets du graphe (ressemblant à la caractérisation des cycles eulériens).

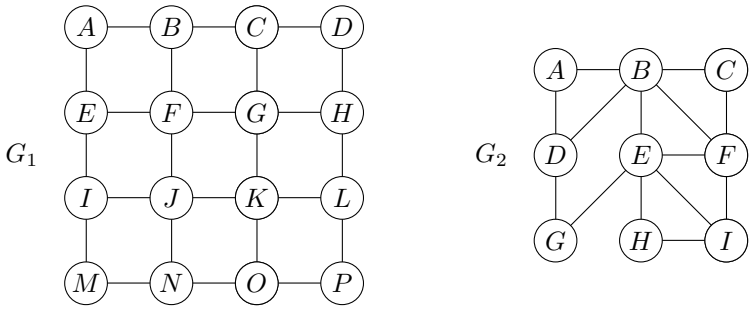
Exercice 5 (Coloration de graphes)

L'algorithme de Welsh-Powell, décrit informellement ci-dessous, permet de colorer les sommets d'un graphe de manière que deux sommets adjacents soient de couleurs différentes. On choisit dans cet exercice de colorer les sommets avec des couleurs qui sont des entiers $0, 1, 2, \ldots$.

— Trier le sommets du graphe par ordre de degré décroissant
— couleur := 0 (*couleur initiale*)
— **Tant qu**'il y a encore des sommets non colorés **faire**
  — Parcourir la liste triée des sommets et colorer en *couleur* les sommets non colorés qui ne sont pas connectés à d'autres sommets de la même couleur
  — couleur := couleur + 1 (*choisir une nouvelle couleur*)

À chaque fois que deux sommets ont le même degré, *on les suppose triés par ordre alphabétique*. Par exemple, si les sommets A, D, B sont tous de degré 3, on suppose qu'ils seront triés dans l'ordre A, B, D.

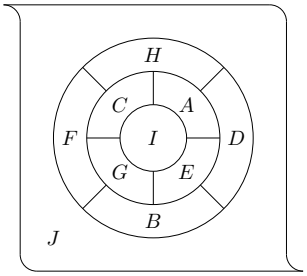
1. Colorier les graphes suivants à l'aide de l'algorithme de Welsh-Powell.



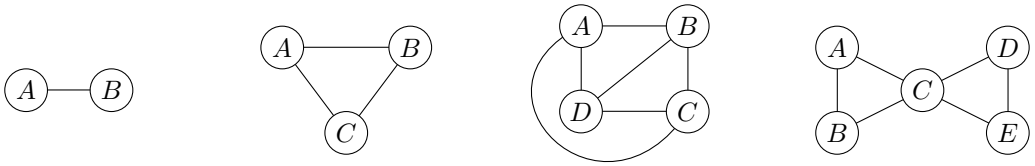
2. Les colorations obtenues sont-elle optimales en terme du nombre de couleurs utilisées ? Si oui, pourquoi ? Si non, trouver une meilleure coloration.

Exercice 6 (Coloration de cartes)

Une carte est un découpage d'une feuille de papier en régions, séparées par des frontières. Par exemple, la carte à droite est découpée en 10 régions. Chaque carte est associée à un *graphe planaire* (un graphe qu'on peut dessiner sur une feuille de papier sans qu'aucune des arêtes n'en croise une autre). Vice versa, chaque graphe planaire est associé à une carte (non nécessairement unique). Chaque région de la carte correspond à un sommet et chaque frontière entre régions à une arête.



1. Dessiner une carte pour chacun des graphes suivants.



2. À l'aide de l'algorithme de Welsh-Powell, colorier la carte donnée en exemple au début de l'exercice. *Noter que la région externe J correspond, elle aussi, à l'un des sommets du graphe associé. (Comme dans l'exercice précédent, si deux sommets ont le même degré, choisir d'abord le plus petit selon l'ordre alphabétique.)*
3. La coloration obtenue est-elle optimale en terme de nombre de couleurs ? Si oui, pourquoi ? Si non, trouver une meilleure coloration.