

Corrections des TD 12 à 19

Remarques importantes

- Inutile d'imprimer ces corrigés : ce ne sont pas des documents autorisés lors des examens.
- On ne comprend vraiment qu'en faisant soi-même. Lire la correction d'un exercice sans avoir fait l'exercice est une perte de temps ; c'est même pire, puisque potentiellement on perd une occasion d'apprendre quelque chose.

Correction TD 12 : Complexité

Exercice 1. Dans cet exercice, on utilise la « représentation par défaut » des graphes (tableau d'ensembles d'adjacence), pour laquelle on considère que toutes les opérations de base sont $O(1)$. Attention, on n'a pas accès directement au degré d'un sommet, il faut le calculer.

1. Écrire une fonction `estIsolé(G, s)` qui prend en paramètre un graphe non orienté et un de ses sommets, et indique si c'est un sommet isolé (c'est-à-dire sans arête incidente). Attention, votre fonction doit être en $O(1)$, quel que soit le degré du sommet !

```
fonction estIsolé(G, s):  
    pour chaque arête incidente à s:  
        renvoyer faux  
    renvoyer vrai
```

Il peut être tentant de calculer le degré et de vérifier s'il vaut 0, mais c'est du travail inutile : dès qu'on a trouvé une arête incidente, on peut déjà répondre à la question. La fonction est en $O(1)$, bien qu'il y ait une boucle, parce qu'il ne peut jamais y avoir plus d'une itération.

2. Écrire une fonction `nbSommetsIsolés` qui prend en paramètre un graphe non orienté et renvoie son nombre de sommets isolés (c'est-à-dire sans arête incidente). Donner sa complexité en fonction de $|S|$ (le nombre de sommets du graphe).

```
nb = 0  
pour chaque sommet s de G:  
    si estIsolé(G, s):  
        nb = nb + 1  
renvoyer nb
```

Cet algorithme a une complexité en $O(|S|)$: chaque itération est en $O(1)$.

3. Écrire une fonction `degréMax` qui prend en paramètre un graphe non orienté et renvoie son degré maximal $\Delta = \max_{s \in S} d(s)$. Donner sa complexité en fonction de $|S|$ et Δ .

```
degmax = -∞  
pour chaque sommet s de G:  
    deg = 0  
    pour chaque arête incidente à s:  
        deg = deg + 1  
    si deg > degmax:  
        degmax = deg  
renvoyer degmax
```

L'itération sur le sommet s parcourt toutes ses arêtes incidentes, donc est en $O(d(s))$. Chaque itération sur les sommets est donc en $O(\Delta)$. L'algorithme est donc en $O(|S| \cdot \Delta)$.

4. On applique `degréMax` aux graphes-étoiles (famille S_n). Réécrire la complexité trouvée à la question précédente, mais en fonction de n seulement. Bien sûr $n = |S|$, mais j'utilise n ici pour mieux faire la différence entre ce cas spécifique (la famille S_n) et le cas général.

Pour S_n on a $\Delta = n - 1$ et $|S| = n$, donc la complexité revient à $O(n^2)$.

5. Cette complexité vous semble-t-elle « réaliste » ? En séparant conceptuellement la boucle en deux parties (l'itération sur le sommet universel d'une part, toutes les itérations sur les feuilles d'autre part), donner une complexité plus précise, toujours en fonction de n .

L'itération sur le sommet universel est en $O(\Delta) = O(n)$, toutes les $n - 1$ autres itérations sont en $O(1)$, donc au total cela fait $O(n)$. (Remarque : la complexité ne peut pas être plus basse, puisque l'algo parcourt forcément tous les sommets.)

6. Maintenant que l'on a vu que la complexité trouvée à la question 3 est trop grossière, on va la préciser dans le cas général.

- (a) Soit s un sommet quelconque du graphe. On ne considère que l'itération sur le sommet s : combien de fois au maximum chaque instruction de l'itération est-elle effectuée, en fonction de $d(s)$, le degré de s ? (Attention, il n'est pas question de complexité asymptotique ici : une fois n'est pas coutume, on va compter les instructions.)

Dans l'itération sur le sommet s , chaque instruction est effectuée au maximum une fois, sauf l'incréméntation de `deg`, qui est effectuée $d(s)$ fois.

- (b) On considère maintenant l'ensemble de toutes les itérations : combien de fois au maximum chaque instruction est-elle effectuée, en fonction de $|S|$ et $|A|$? Indice : utiliser la question précédente et la formule de la somme des degrés.

À part l'incréméntation de `deg`, chaque instruction dans la boucle principale est effectuée au maximum une fois par itération, et il y a $|S|$ itérations, donc elle est effectuée au maximum $|S|$ fois. L'incréméntation, quant à elle, est effectuée $d(s)$ fois pour chaque s , donc au total elle est effectuée $\sum_{s \in S} d(s)$ fois, c'est-à-dire $2 \cdot |A|$ fois selon la formule de la somme des degrés.

- (c) En déduire la complexité asymptotique de l'algorithme `degréMax` en fonction de $|S|$ et $|A|$.

Pour résumer, dans l'algorithme, il y a des instructions qui sont répétées $|S|$ fois, et une instruction qui est répétée $2 \cdot |A|$ fois. Concrètement, les instructions dont il est question ne prennent pas forcément toutes le même nombre d'opérations élémentaires, mais en termes de complexité asymptotique ça n'a pas d'importance car ce sont des constantes. Au final le nombre d'opérations élémentaires est $K \times |S| + K' \times |A|$ (où K, K' sont des constantes), ce qui est en $O(|S| + |A|)$.

NB : ce point est clairement délicat. Il est très compréhensible que vous soyez troublé·es par ce résultat (pourquoi une somme, alors que ce sont des boucles imbriquées ?). Cependant, soyez conscient·es que c'est le seul point délicat qu'on aborde dans la complexité des graphes – le fait qu'une boucle sur tous les sommets qui itère sur tous les voisins de chaque sommet est en $O(|S| + |A|)$. Tout le reste du temps, la complexité asymptotique des boucles s'obtient en multipliant la complexité d'une itération par le nombre d'itérations.

Exercice 2 (adapté d'un exercice de Julien Courtiel). Dans cet exercice, on ne considère que des graphes orientés simples dont les sommets sont les entiers de 1 à n . On appelle *graphe miroir* d'un graphe orienté simple, le graphe orienté obtenu en inversant le sens de chaque arc.

1. Écrire une fonction `miroirMatrice` qui prend en paramètre une matrice d'adjacence et qui renvoie la matrice d'adjacence du graphe miroir. Quelle est la complexité de `miroirMatrice` ?

```

n = taille(M)
M' = matrice de taille (n, n) remplie de zéros
pour i de 1 à n:
    pour j de 1 à n:
        si M[i][j] = 1:
            M'[j][i] = 1
renvoyer M'

```

L'algo est en $O(|S|^2)$.

- Écrire une fonction `miroirDico` qui prend en paramètre un graphe représenté comme un dictionnaire (table de hachage) dont les clefs sont les sommets et les valeurs sont les ensembles de successeurs. Quelle est la complexité de votre algorithme, en fonction de $|S|$?

```

D' = dictionnaire vide
pour chaque clef s de D:
    D'[s] = ensemble vide
pour chaque clef s de D:
    pour chaque s' dans D[s]:
        ajouter s à l'ensemble D'[s']
renvoyer D'

```

La première boucle est clairement en $O(|S|)$. Pour la deuxième, la boucle intérieure fait $d^+(s)$ itérations, donc l'instruction d'ajout est effectuée $\sum_{s \in S} d^+(s) = |A|$ par la formule des poignées de mains. La complexité de la fonction est donc $O(|S| + |A|)$.

- Quelle est la complexité de la 2e version en fonction seulement de $|S|$, si le graphe a peu d'arcs (à peu près autant que de sommets)? Et s'il en a beaucoup (par exemple, si tous les arcs possibles sont présents)? Quel est l'algo le plus efficace dans chacun de ces cas?

Si le graphe a peu d'arcs, $|A| \in O(|S|)$, donc la complexité de `miroirDico` est $O(|S|)$. Si tous les arcs sont présents dans le graphe, il y en a un nombre $O(|S|^2)$, donc la complexité de `miroirDico` est $O(|S|^2)$. Dans le premier cas, `miroirDico` est meilleur, dans le second cas les deux versions sont équivalentes d'un point de vue complexité théorique.

Exercice 3.

- Écrire une fonction qui prend en paramètre un graphe simple non orienté G et qui renvoie une clique maximale (pour l'inclusion) de G . Indice : il pourrait être une bonne idée de faire d'abord une fonction `voisinDeTous(G, s, C)`, qui renvoie vrai ssi le sommet s donné est voisin de tous les sommets de l'ensemble C donné.

```

fonction voisinDeTous(G, s, C):
    pour chaque sommet s' de C:
        si s et s' ne sont pas voisins dans G:
            renvoyer faux
    renvoyer vrai

fonction cliqueMaximale(G):
    C = ensemble vide
    pour chaque sommet s de G:
        si voisinDeTous(G, s, C):

```

```

    C = C + {s}
    renvoyer C

(autre version :
  C = ensemble vide
  candidats = tous les sommets
  tant qu'il reste des candidats :
    s = un candidat
    ajouter s à C
    pour chaque candidat :
      s'il n'est pas voisin de s :
        le retirer
  )

```

2. Donner la complexité au pire cas de votre fonction.

Il est facile de voir que la complexité au pire cas de `voisinDeTous` est proportionnelle à $|C|$, donc en $O(|S|)$. Dans `cliqueMaximale`, on appelle `voisinDeTous` pour chaque sommet, on obtient donc une complexité en $O(|S|^2)$.

Cette borne supérieure est-elle trop lâche ? On peut essayer de préciser, en réfléchissant au pire des cas, qui arrive lorsque G est un graphe complet. Dans ce cas, à chaque itération dans `cliqueMaximale` on ajoute le sommet considéré dans C . La complexité de la boucle est donc proportionnelle à $\sum_{i=1}^{|S|} (i-1)$, ce qui est bien $O(|S|^2)$.

3. On veut maintenant écrire une fonction qui renvoie un *stable* maximal (pour l'inclusion). Il y a deux solutions :

- adapter l'algorithme de recherche de clique, en utilisant une fonction `voisinDAucun` à la place de `voisinDeTous`
- construire le complémentaire du graphe d'entrée (le graphe dont l'ensemble d'arêtes est exactement le complémentaire de l'ensemble d'arêtes du graphe d'entrée) puis lancer `cliqueMaximale` dessus (un stable étant toujours une clique du complémentaire).

Quelle est la solution la plus efficace des deux du point de vue de la complexité algorithmique ? En pratique, que feriez-vous ?

a priori en termes de complexité c'est pareil. En pratique, la première solution est certainement plus efficace, mais ce n'est pas absurde de préférer la 2e pour éviter de « dupliquer » la logique dans des fonctions extrêmement proches.

Exercice 4. Cet exercice traite de graphes orientés simples. On supposera toujours que les graphes ont au moins un sommet et un arc.

1. On va commencer par déterminer la complexité au pire cas de l'algo suivant :

```

fonction existeCheminOrientéLongueurN(G, s1, s2, N):
  si N = 0:
    renvoyer (s1 = s2)
  pour chaque voisin v de s1:
    si existeCheminOrientéLongueurN(G, v, s2, N-1):
      renvoyer vrai
  renvoyer faux

```

- (a) On note $c(N)$ la complexité au pire cas de l'algo suivant appelé avec le paramètre $N > 1$, et D le degré sortant maximal du graphe. Exprimer $c(N)$ en fonction de D et de $c(N-1)$.

$c(N) = O(1) + D \times (O(1) + c(N-1))$. On peut simplifier puisque $D \geq 1$:
 $c(N) = O(D) + D \times c(N-1)$.

- (b) Exprimer $c(0)$, $c(1)$, puis $c(2)$ en fonction de D . En déduire la formule générale de $c(N)$ en fonction de D et N .

— le cas de base est $c(0) = O(1)$
 — $c(1) = O(D) + D \times (O(1)) = O(D)$
 — $c(2) = O(D) + D \times (O(D)) = O(D^2)$
 — etc., jusqu'à $c(N) = O(D^N)$.

- (c) On ne souhaite pas s'encombrer de D dans la suite de l'exercice. Exprimer la complexité de `existeCheminOrientéLongueurN` en fonction de N et $|A|$ (ce sera moins précis, mais tant pis).

On sait que D est $O(|A|)$, donc la réponse est $O(|A|^N)$.

2. Déterminer la complexité au pire cas de l'algo suivant, en fonction de la taille du graphe $|A|$.

```
fonction distanceOrientée(G, s1, s2):
  pour i variant de 0 à taille(G):
    si existeCheminOrientéLongueurN(G, s1, s2, i):
      renvoyer i
  renvoyer +oo
```

À chaque itération, on fait $O(|A|^i)$ opérations. Dans le pire des cas on va parcourir toute la boucle, on fera donc $\sum_{i=1}^{|A|} O(|A|^i)$ opérations, ce qui fait $O(|A|^{|A|})$ (rappel : $\sum_{i=1}^n x^i = \frac{x^{n+1}-x}{x-1}$, ce qui est $O(x^n)$), et donc $O(|A|^{|A|})$. C'est une complexité monstrueuse (pire que factorielle).

3. Écrire une fonction indiquant si un graphe orienté simple donné en paramètre est fortement connexe, en utilisant la fonction `distanceOrientée`, et donner sa complexité au pire cas en fonction de $|A|$ et $|S|$.

```
fonction estFortementConnexe(G):
  pour chaque sommet s de G:
    pour chaque sommet s' de G:
      si distanceOrientée(G, s, s') = +oo:
        renvoyer faux
  renvoyer vrai
```

La complexité est $O(|S|^2 \times |A|^{|A|})$.

4. Exprimer la complexité en fonction de $|S|$ seulement, dans le cas où il y a peu d'arcs et dans le cas où il y a beaucoup d'arcs.

S'il y a peu d'arcs, $|A| \in O(|S|)$ donc la complexité de `estFortementConnexe` est $O(|S|^{|S|+2})$.
 S'il y a beaucoup d'arcs, $|A| \in O(|S|^2)$, donc la complexité devient $O(|S|^2 \times (|S|^2)^{|S|^2}) = O(|S|^{2|S|^2+2})$.

5. Que devient la complexité au pire cas de votre fonction si on dispose d'une version efficace, en $O(|A|)$, de la fonction `existeCheminOrientéLongueurN`? L'exprimer en fonction de $|S|$ et $|A|$, puis en fonction de $|S|$ seulement dans les cas où il y a peu d'arcs et beaucoup d'arcs.

La fonction distance devient alors de complexité $O(|A|^2)$, donc la dernière fonction est en $O(|S|^2 \times |A|^2)$, ce qui est très mauvais mais monstrueusement meilleur que la précédente !

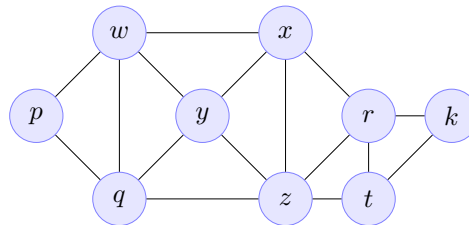
S'il y a peu d'arcs la complexité est en $O(|S|^4)$, s'il y en a beaucoup c'est $O(|S|^2 \times (|S|^2)^2) = O(|S|^6)$.

Exercice 5. Écrire un algorithme qui calcule le diamètre d'un graphe non orienté et donner sa complexité au pire cas, en utilisant l'algorithme de distance de l'exercice 4.

Exercice 6. Écrire une fonction récursive `nbComposantesConnexes(G, E)` qui renvoie le nombre de composantes connexes du sous-graphe de G induit par l'ensemble de sommets E . Le deuxième paramètre permet de simplifier la récursion, et déterminer sa complexité au pire cas.

Correction TD 13 : Parcours de graphes

Exercice 1. On considère le graphe non orienté connexe suivant :



1. Soit l'algorithme suivant, qui construit un arbre couvrant d'un graphe non orienté connexe :

```

fonction arbreCouvrant(G, s):
    T = creerGraphe()
    ajouterSommet(T, s)
    arbreCouvrantRec(G, s, T)
    renvoyer T

fonction arbreCouvrantRec(G, s, T):
    colorer s
    pour chaque successeur v de s:
        si v est blanc:
            ajouterSommet(T, v)
            ajouterArête(T, {s,v})
            arbreCouvrantRec(G, v, T)
    
```

(a) De quel type de parcours s'agit-il ?

parcours en profondeur (DFS)

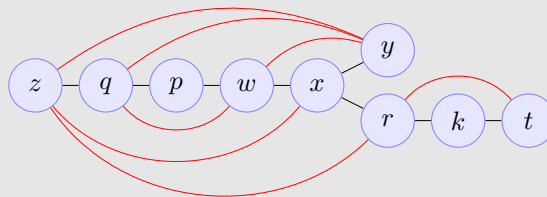
(b) Que représente le fait qu'un sommet soit coloré ?

- Les sommets colorés sont ceux qui ont été découverts
- Les sommets colorés sont ceux qui ont été visités

On colore un sommet quand on commence sa visite, donc la 2e réponse est correcte. Mais il est intéressant de noter que dans la version récursive de DFS, on visite immédiatement les sommets découverts, donc la 1e réponse n'est pas vraiment fausse non plus.

(c) Exécuter l'algorithme sur le graphe donné, en partant du sommet z , en dessinant progressivement l'arbre couvrant construit, en prenant comme ordre de priorité sur les successeurs l'ordre alphabétique.

Au final on obtient l'arbre suivant (en rouge les arêtes de saut demandées à la



question suivante) :

- (d) Ajouter à l'arbre les arêtes de saut en rouge et les arêtes transverses en vert.

Voir question précédente. Il n'y a pas d'arêtes vertes car il n'y a pas d'arête transverses avec un DFS.

2. On considère maintenant l'algorithme suivant, qui construit également un arbre couvrant pour un graphe non orienté connexe, mais calcule en même temps la distance de chaque sommet au sommet de départ.

```
fonction arbreCouvrantEtDistance(G, s):
    T = creerGraphe()
    dist = dictionnaire vide
    f = file vide

    colorer s
    enfiler s dans f
    ajouterSommet(T, s)
    dist[s] = 0

    tant que f est non vide:
        u = défiler f
        pour chaque successeur v de u:
            si v est blanc:
                ajouterSommet(T, v)
                ajouterArête(T, {u,v})
                dist[v] = dist[u] + 1
                colorer v
                enfiler v dans f

    renvoyer T et dist
```

- (a) De quel type de parcours s'agit-il ?

parcours en largeur (BFS)

- (b) Que représente le fait qu'un sommet soit coloré ?
- Les sommets colorés sont ceux qui ont été découverts
 - Les sommets colorés sont ceux qui ont été visités

On colore un sommet quand on l'ajoute à la file, donc la première réponse est correcte.

- (c) Exécuter l'algorithme sur le graphe donné, en dessinant progressivement l'arbre couvrant construit tout en remplissant progressivement le dictionnaire des distances, et décrire l'état de la file à chaque fois qu'elle change, et ce pour un des deux ordres de priorité considérés.

Ordre alphabétique

```
État de la file au début : [z]
(NB: j'enfile à droite, je défile à gauche)
Sommets colorés au début : z
on défile, sommet courant z, f=[]
f=[q], sommets colorés : z, q
f=[q,r], sommets colorés : z, q, r
f=[q,r,t], sommets colorés : z, q, r, t
f=[q,r,t,x], sommets colorés : z, q, r, t, x
f=[q,r,t,x,y], sommets colorés : z, q, r, t, x, y
on défile, sommet courant q, f=[r,t,x,y]
f=[r,t,x,y,p], sommets colorés : z, q, r, t, x, y, p
f=[r,t,x,y,p,w], sommets colorés : z, q, r, t, x, y, p, w
on défile, sommet courant r, f=[t,x,y,p,w]
f=[t,x,y,p,w,k], sommets colorés : tous
```

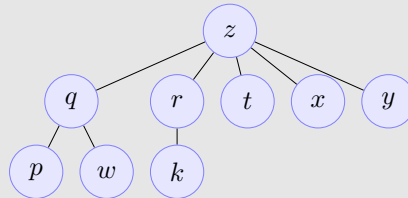
```

on défile , sommet courant t , f=[x,y,p,w,k]
on défile , sommet courant x , f=[y,p,w,k]
on défile , sommet courant y , f=[p,w,k]
on défile , sommet courant p , f=[w,k]
on défile , sommet courant w , f=[k]
on défile , sommet courant k , f=[]

```

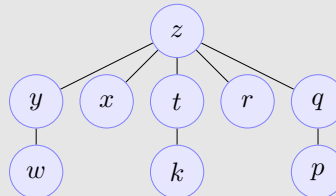
Tableau des distances :

<i>k</i>	<i>p</i>	<i>q</i>	<i>r</i>	<i>t</i>	<i>w</i>	<i>x</i>	<i>y</i>	<i>z</i>
2	2	1	1	1	2	1	1	0



On obtient l'arbre suivant :

Ordre alphabétique inverse



On obtient l'arbre suivant :

- (d) Ajouter à l'arbre les arêtes de saut en rouge et les arêtes transverses en vert.

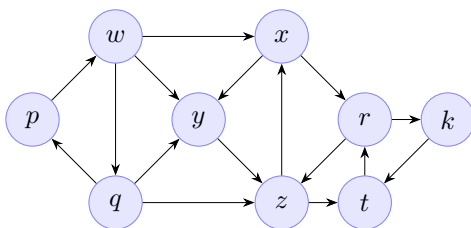
Il n'y aura que des arêtes vertes : pas d'arêtes de saut avec BFS.

- (e) Les arbres obtenus par les deux algos sont-ils similaires ? Quelles sont les caractéristiques qui changent ?

Les arbres sont assez différents. Ceux construits par DFS sont plutôt hauts et de faible arité, tandis que ceux construits par BFS sont plutôt de faible hauteur et de grande arité. (Mais bien sûr cela dépend du graphe : si son degré maximal est faible, les arbres construits par BFS ne pourront pas être très touffus.) Par ailleurs, sur les arbres de DFS il n'y a pas d'arête transverse, tandis que sur les arbres de BFS il n'y a pas d'arête de saut.

Exercice 2.

On considère le graphe orienté ci-dessous, ainsi que l'algorithme suivant, qui prend un graphe orienté et deux sommets s et s' en paramètres et renvoie vrai ssi s' est atteignable depuis s .



```

fonction estAtteignable(G, s, s') :
    trouvé = faux
    parcours(G, s, s')
    renvoyer trouvé

```

```

fonction parcours(G, s, s') :
    si s = s' :
        trouvé = vrai
    colorer s
    pour chaque successeur v de s :
        si v est blanc :
            parcours(G, v, s')

```

1. De quel type de parcours s'agit-il ?

parcours en profondeur (DFS)

2. Appliquer l'algorithme sur le graphe donné pour vérifier si w est atteignable depuis z .

```
{ on visite z ; ses successeurs sont [t,x]
  { on visite t ; ses successeurs sont [r]
    { on visite r ; ses successeurs sont [k]
      { on visite k ; ses successeurs sont [t]
        * t est déjà visité
      } on termine k
    } on termine r
  } on termine t
  { on visite x ; ses successeurs sont [r,y]
    * r est déjà visité
    { on visite y ; ses successeurs sont [z]
      * z est déjà visité
    } on termine y
  } on termine x
} on termine z
```

La fonction renvoie faux puisque le parcours n'atteint jamais w .

3. Faire de même en inversant le sens de l'arête (q, y) .

Tout est pareil jusqu'à la visite de y

```
{ on visite z ; ses successeurs sont [t,x]
  { on visite t
    [...]
  } on termine t
  { on visite x ; ses successeurs sont [r,y]
    * r est déjà visité
    { on visite y ; ses successeurs sont [q,z]
      { on visite q ; ses successeurs sont [p,y]
        { on visite p ; ses successeurs sont [w]
          { on visite w ; ses successeurs sont [x,y]
            on met trouvé à vrai
            * x est déjà visite
            * y est déjà visite
          } on termine w
        } on termine p
        * y est déjà visite
      } on termine q
      * z est déjà visite
    } on termine y
  } on termine x
} on termine z
```

Exercice 3. Écrire un algo qui utilise un parcours pour renvoyer le nombre de composantes connexes d'un graphe non orienté.

```
fonction nbComposantes(G):
  nb = 0
  tant qu'il reste un sommet blanc dans G:
    nb += 1
```

```

    choisir un sommet blanc  $s$  de  $G$ 
    parcourir( $G, s$ )
    renvoyer nb

```

NB : la fonction parcourir peut être DFS ou BFS, l'important est qu'elle colore les sommets rencontrés.

Exercice 4. On considère l'algorithme de parcours suivant, qui prend en paramètre un graphe **orienté** et l'un de ses sommets.

```

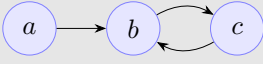
fonction parcourir( $G, s$ ):
    colorer  $s$  en gris
    pour chaque successeur  $v$  de  $s$ :
        si  $v$  est gris:
            afficher "trouvé sommet gris"
            stop
        si  $v$  est noir:
            afficher "trouvé sommet noir"
            stop
        si  $v$  est blanc:
            parcourir( $G, v$ )
    colorer  $s$  en noir

```

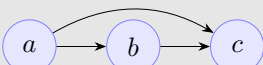
1. De quel type de parcours s'agit-il ?

parcours en profondeur (DFS), formulation récursive

2. Trouver un graphe orienté G et un sommet s tels que l'algorithme appelé sur G et s affiche « trouvé sommet gris ». Indiquer la liste des sommets visités et le sommet gris rencontré.

Un exemple :  Appelé sur le sommet a , on visite a , puis b , puis c , et là on trouve b comme sommet gris.
Ce n'est pas l'exemple le plus simple. Le plus simple est un seul sommet avec une boucle, mais je ne suis pas sûr que ce soit très parlant.

3. Trouver un graphe orienté G et un sommet s tels que l'algorithme appelé sur G et s affiche « trouvé sommet noir ». Indiquer la liste des sommets visités et le sommet noir rencontré.

Un exemple :  Appelé sur le sommet a , on visite a , puis b , puis c , puis on colore c en noir, puis b , et enfin on reprend l'itération sur les successeurs de a et on trouve c comme sommet noir.

4. Quelle est la catégorie de l'arc qui vient d'être suivi quand s'affiche « trouvé sommet gris » ? Et « trouvé sommet noir » ?
5. Démontrer qu'un graphe orienté contient un cycle orienté si et seulement si les arborescences couvrantes construites par DFS ont un arc de saut en arrière.

(\Rightarrow) Idée de la preuve : soit u le premier sommet visité du cycle, et soit v le sommet qui le précède sur le cycle. L'algo visite u avant v , donc l'arc (v, u) n'est pas un arc de liaison. Ça ne peut pas être un arc transverse, car DFS n'en génère pas (d'après le cours, mais pas prouvé). C'est donc un arc de saut, plus précisément de saut en arrière (u ne peut pas être descendant de v dans l'arborescence couvrante, puisqu'il a été visité avant).

(\Leftarrow) S'il y a un arc de saut en arrière, cela signifie que le graphe contient un arc (v, u) où v est un descendant de u dans l'arborescence. Il y a un chemin orienté de u à v

(puisque v est un descendant de u), on obtient un cycle orienté en ajoutant l'arc (v, u) à la fin.

6. Écrire une fonction `contientCycle(G)` qui prend en paramètre un graphe orienté et renvoie vrai si et seulement s'il contient un cycle orienté. On utilisera une version modifiée de la fonction `parcours`, et on pourra utiliser une variable globale pour noter le résultat, comme dans l'exercice 2. Attention, le graphe donné en entrée n'est pas forcément connexe.

On va utiliser ce qu'on a vu aux questions précédentes : chercher un arc de saut en arrière en regardant les couleurs des sommets. L'idée est la suivante : lors de notre parcours, lorsqu'on regarde les successeurs du sommet courant u , si on en voit un qui est gris (disons v), ça veut dire qu'il est visité mais pas encore « terminé », donc v est un ancêtre de u dans l'arbre du parcours. L'arc (u, v) est donc un arc de saut en arrière.

```
fonction contientCycle(G):
    trouvé = faux
    tant qu'il reste un sommet blanc dans G:
        choisir un sommet blanc s de G
        parcours(G, s)
    renvoyer trouvé

fonction parcours(G, s):
    colorer s en gris
    pour chaque successeur v de s:
        si v est coloré en gris:
            trouvé = vrai
        si v est blanc:
            parcours(G, v)
    colorer s en noir
```

Correction TD 14 : Algorithme de Dijkstra

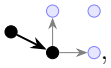
Dans ce TD on étudie l'algorithme de Dijkstra, rappelé ci-dessous.

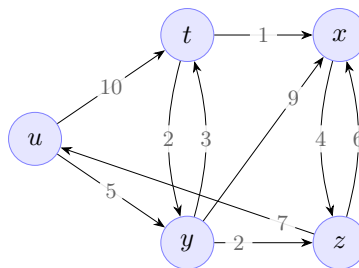
```

fonction Dijkstra(G, source):
    dist, pred = dictionnaires
    nonVisités = {}
    pour chaque sommet s de G:
        dist[s] = +∞
        pred[s] = nil
        ajouter s à nonVisités
    dist[source] = 0

    tant que nonVisités n'est pas vide:
        prendre le sommet s de nonVisités tq dist[s] est minimale
        retirer s de nonVisités

        pour chaque successeur s' de s qui est dans nonVisités:
            alt = dist[s] + poids[s,s']
            si alt < dist[s']:
                dist[s'] = alt
                pred[s'] = s
    renvoyer dist, pred
    
```

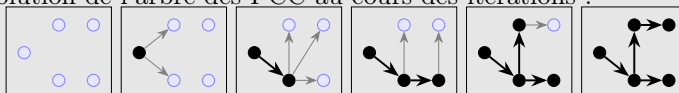
Exercice 1. Appliquer l'algorithme de Dijkstra sur le graphe ci-dessous en prenant comme source le sommet u . Pour chaque itération de la boucle principale, ajouter une ligne au tableau suivant et dessiner schématiquement l'aspect courant de l'arbre de plus courts chemins (par exemple , où les sommets noirs sont les sommets visités, les arcs faisant définitivement partie de l'arbre de PCC sont en noir, et ceux qui peuvent encore changer sont grisés).



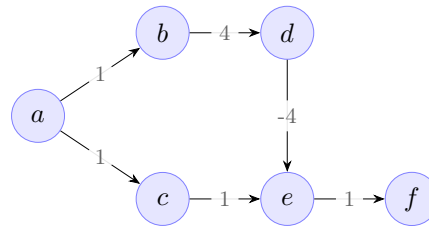
s	nonVisités	dist[x]	dist[y]	dist[z]	dist[t]	dist[u]	pred[x]	pred[y]	pred[z]	pred[t]	pred[u]
(début)	{x, y, z, t, u}	+∞	+∞	+∞	+∞	0	nil	nil	nil	nil	nil
u							

s	nonVisités	dist[x]	dist[y]	dist[z]	dist[t]	dist[u]	pred[x]	pred[y]	pred[z]	pred[t]	pred[u]
(début)	{x, y, z, t, u}	+∞	+∞	+∞	+∞	0	nil	nil	nil	nil	nil
u	{x, y, z, t}	+∞	5	+∞	10	0	nil	u	nil	u	nil
y	{x, z, t}	14	5	7	8	0	y	u	y	y	nil
z	{x, t}	13	5	7	8	0	z	u	y	y	nil
t	{x}	9	5	7	8	0	t	u	y	y	nil
x	{}	9	5	7	8	0	t	u	y	y	nil

Évolution de l'arbre des PCC au cours des itérations :



Exercice 2. Dérouler l'algorithme de Dijkstra sur le graphe suivant, avec a pour sommet source.



La distance calculée pour f est-elle correcte ? Pourquoi ?

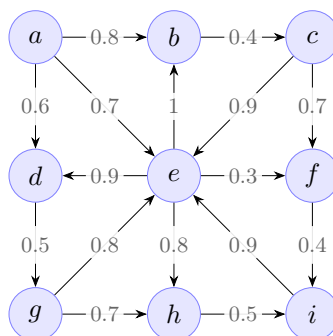
s	(début)	a	b	c	e	f	d
nonVisités	{a, b, c, d, e, f}	{b, c, d, e, f}	{c, d, e, f}	{d, e, f}	{d, f}	{f}	{d}
dist[a]	0	0	0	0	0	0	0
dist[b]	$+\infty$	1	1	1	1	1	1
dist[c]	$+\infty$	1	1	1	1	1	1
dist[d]	$+\infty$	$+\infty$	5	5	5	5	5
dist[e]	$+\infty$	$+\infty$	$+\infty$	2	2	2	2
dist[f]	$+\infty$	$+\infty$	$+\infty$	$+\infty$	3	3	3
pred[a]	nil	nil	nil	nil	nil	nil	nil
pred[b]	nil	a	a	a	a	a	a
pred[c]	nil	a	a	a	a	a	a
pred[d]	nil	nil	b	b	b	b	b
pred[e]	nil	nil	nil	c	c	c	c
pred[f]	nil	nil	nil	nil	e	e	e

À la dernière étape, pour d , on trouve une distance plus basse que celle trouvée précédemment, alors que d est déjà visité. On peut la mettre à jour, mais la distance pour f reste fausse.

Dijkstra ne fonctionne que pour des poids positifs. En effet, une hypothèse fondamentale de Dijkstra est que lorsqu'on visite un sommet s (qu'on le prend comme sommet courant), on sait que $\text{dist}[s]$ est la vraie distance, il ne peut pas exister ailleurs dans le graphe de chemin plus court qui mène à s . S'il y a des poids négatifs, il se peut très bien qu'il y ait quelque part un arc avec un poids très négatif qui va modifier toutes les distances connues quand on l'explorera.

Exercice 3 (Exercice de P. Boizumault). On considère un réseau de télécommunications, composé d'émetteurs/récepteurs pouvant s'envoyer des messages avec une certaine fiabilité, c'est-à-dire une certaine probabilité que la communication ne soit pas interrompue.

On modélise ce problème par le graphe orienté valué suivant, où la valuation d'un arc est une valeur réelle entre 0 et 1 indiquant la probabilité que la communication se passe sans souci.



La question qu'on se pose est la suivante : comment déterminer le chemin le plus fiable pour envoyer un message d'un sommet source (a) vers un sommet cible (i) ?

On cherche un chemin dont le produit des probabilités est maximal. Pour cela, il est possible d'adapter l'algorithme de Dijkstra. Ce dernier est reproduit ci-dessous, avec des TODO aux endroits qu'il faut adapter.

NB : cet algorithme n'est correct que si les poids des arcs sont tous compris entre 0 et 1 : chaque fois que l'on ajoute un arc à un chemin, on diminue la fiabilité du chemin — de la même façon que pour Dijkstra classique, où ajouter un arc ne peut que rallonger le chemin (l'algorithme ne fonctionne pas en présence de poids négatifs).

```
fonction DijkstraAdapté(G, source):
    dist, pred = dictionnaires
    nonVisités = {}
    pour chaque sommet s de G:
        dist[s] = TODO
        pred[s] = nil
        ajouter s à nonVisités
    dist[source] = TODO

    tant que nonVisités n'est pas vide:
        prendre le sommet s de nonVisités tq TODO
        retirer s de nonVisités

        pour chaque successeur s' de s qui est dans nonVisités:
            alt = TODO
            si TODO
                dist[s'] = alt
                pred[s'] = s
    renvoyer dist, pred
```

- On initialise tous les $\text{dist}[s]$ à 0 (proba la plus basse possible)
- $\text{dist}[\text{source}] = 1$ (élément neutre de la multiplication)
- on prend le sommet s de nonVisités tq $\text{dist}[s]$ est maximale (la meilleure proba, le sommet pour lequel la communication depuis la source est la plus fiable)
- calcul de la proba du successeur : $\text{alt} = \text{dist}[s] \times \text{poids}[s, s']$
- on met à jour dist si $\text{alt} > \text{dist}[s']$ (on a trouvé un chemin plus fiable)

Correction TD 15 : Bellman-Ford et Floyd-Warshall

On rappelle ci-contre l'algorithme de Bellman-Ford, qui prend en entrée un graphe orienté valué simple et un sommet source, et calcule la distance d'un sommet source à tous les autres sommets ainsi qu'un arbre de plus courts chemins, même en présence de poids négatifs. S'il existe un cycle absorbant, l'algorithme le détecte.

```

fonction BellmanFord(G, source):
    dist, pred = dictionnaires vides
    pour chaque sommet s de G:
        dist[s] = +∞
        pred[s] = nil
    dist[source] = 0

    répéter |S| fois:
        pour chaque arc (s, s') de G:
            alt = dist[s] + poids[s, s']
            si alt < dist[s']:
                dist[s'] = alt
                pred[s'] = s

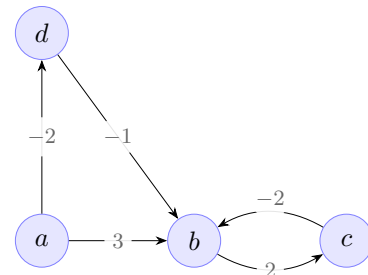
    si dist a changé lors de la dernière itération:
        afficher "trouvé un cycle absorbant"

    renvoyer dist, pred
    
```

Exercice 1 (Algo de Bellman-Ford).

1. Appliquer l'algorithme de Bellman-Ford sur le graphe ci-contre en prenant comme source le sommet a , et en suivant l'ordre suivant pour le parcours des arcs : (a, b) , (b, c) , (c, b) , (d, b) , (a, d) . Pour chaque relaxation d'arc, ajouter une ligne au tableau suivant et dessiner schématiquement l'aspect courant de l'arbre de plus courts chemins.

Attention, il y a plusieurs relaxations d'arcs par itération de la boucle principale! (Une fois que vous êtes à l'aise, vous pouvez omettre les lignes des relaxations qui ne changent rien dans le tableau. Mais il doit toujours y avoir au moins une ligne par itération principale.)



itération	arc	dist[a]	dist[b]	dist[c]	dist[d]	pred[a]	pred[b]	pred[c]	pred[d]
0		0	+∞	+∞	+∞	nil	nil	nil	nil
1	(a, b)			
1	(b, c)				

Après la première itération, je ne mets plus les lignes où rien ne change.

itération	arc	dist[a]	dist[b]	dist[c]	dist[d]	pred[a]	pred[b]	pred[c]	pred[d]
0		0	+∞	+∞	+∞	nil	nil	nil	nil
1	(a, b)	0	3	+∞	+∞	nil	a	nil	nil
1	(b, c)	0	3	5	+∞	nil	a	b	nil
1	(c, b)	0	3	5	+∞	nil	a	b	nil
1	(d, b)	0	3	5	+∞	nil	a	b	nil
1	(a, d)	0	3	5	-2	nil	a	b	a
2	(d, b)	0	-3	5	-2	nil	d	b	a
3	(b, c)	0	-3	-1	-2	nil	d	b	a
4		(pas de modification)				(pas de modification)			

Aucune modification n'a lieu lors de la dernière itération, l'algorithme détecte donc correctement qu'il n'y a pas de cycle absorbant. À noter que pour cet ordre de parcours

des arcs, les $|S| - 1$ (ici 3) itérations sont nécessaires.

2. On considère maintenant le même graphe, sauf que le poids de l'arc (c, b) est -3 . Appliquer l'algorithme de Bellman-Ford de la même façon que précédemment sur le graphe modifié.

itération	arc	dist[a]	dist[b]	dist[c]	dist[d]	pred[a]	pred[b]	pred[c]	pred[d]
0		0	$+\infty$	$+\infty$	$+\infty$	nil	nil	nil	nil
1	(a, b)	0	3	$+\infty$	$+\infty$	nil	a	nil	nil
1	(b, c)	0	3	5	$+\infty$	nil	a	b	nil
1	(c, b)	0	2	5	$+\infty$	nil	c	b	nil
1	(a, d)	0	2	5	-2	nil	c	b	a
2	(b, c)	0	2	4	-2	nil	c	b	a
2	(c, b)	0	1	4	-2	nil	c	b	a
2	(d, b)	0	-3	4	-2	nil	d	b	a
3	(b, c)	0	-3	-1	-2	nil	c	b	a
3	(c, b)	0	-4	-1	-2	nil	c	b	a
4	(b, c)	0	-4	-2	-2	nil	c	b	a
4	(c, b)	0	-5	-2	-2	nil	c	b	a

Le tableau est modifié lors de la dernière itération : l'algorithme détecte bien qu'il y a un cycle absorbant.

Exercice 2 (Dijkstra vs Bellman-Ford).

1. Quelle est la complexité au pire cas (en fonction de $|S|$ et $|A|$) d'une implémentation « naïve » de l'algorithme de Dijkstra, où `nonVisités` est un simple ensemble qu'il faut parcourir pour récupérer le sommet ayant le plus petit `dist` ?

$O(|S|^2 + |A|)$ (chaque arc n'est « visité » qu'une seule fois)

2. La version standard de l'algorithme de Dijkstra utilise une file à priorité basée sur un tas binaire. Récupérer le sommet ayant le plus petit `dist` est seulement en $O(\log|S|)$, mais en contrepartie mettre à jour `dist` n'est plus constant : chaque mise à jour nécessite $O(\log|S|)$ opérations. Quelle est la complexité au pire cas de cette version de Dijkstra (toujours en fonction de $|S|$ et $|A|$) ?

$O((|S| + |A|) \cdot \log|S|)$

3. Quelle est la complexité de l'algorithme de Bellman-Ford (toujours en fonction de $|S|$ et $|A|$) ?

$O(|S| \cdot |A|)$

4. Pour chercher des plus courts chemins dans un graphe qui comporte des poids négatifs, on ne peut pas utiliser Dijkstra. Mais si tous les poids sont positifs, on a le choix. On peut donc se demander lequel des deux algorithmes a la meilleure complexité.

Pour répondre, on va considérer deux cas :

- les graphes « creux », avec peu d'arcs (disons, moins d'arcs que de sommets) ;
- les graphes denses, avec beaucoup d'arcs (disons, de l'ordre d'un arc par paire de sommets).

Donner la complexité de Dijkstra (version standard) et de Bellman-Ford en fonction de $|S|$ suivant la catégorie de graphe. Si tous les poids sont positifs, lequel des deux algorithmes est à privilégier pour les graphes creux ? Et pour les graphes denses ?

Graphes creux : $|A| \in O(|S|)$ donc Dijkstra en $O(|S| \cdot \log|S|)$ et BF en $O(|S|^2)$: Dijkstra est beaucoup mieux.

Graphes denses : $|A| \in O(|S|^2)$ donc Dijkstra en $O(|S|^2 \cdot \log|S|)$ et BF en $O(|S|^3)$: Dijkstra est également beaucoup mieux.

Exercice 3 (Améliorer Bellman-Ford). L'algorithme de Bellman-Ford fait souvent beaucoup de travail inutile.

1. Proposer des améliorations qui permettent d'éviter des relaxations inutiles et/ou des itérations inutiles.
2. Quelle est la complexité au pire cas de votre algorithme amélioré ? Comparez-la à la complexité au pire cas de l'algorithme de Bellman-Ford basique.

Si le tableau `dist` n'a pas changé lors d'une itération principale, ce n'est pas la peine de continuer.

De plus, pour chaque sommet s , si `dist[s]` n'a pas changé à l'itération précédente, ce n'est pas la peine de relâcher les arcs sortants de s .

On pourrait maintenir par exemple à chaque itération un ensemble des arcs qu'il est utile de relâcher à l'itération suivante (ceux dont on a modifié le `dist` de la source), cela permettrait de faire les deux améliorations en même temps.

Cependant dans les pires cas (par exemple un graphe complet dont tous les arcs sont négatifs) on devra quand même relâcher tous les arcs et faire toutes les itérations, donc la complexité au pire cas ne change pas.

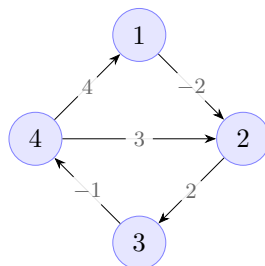
On rappelle à présent l'algorithme de Floyd-Warshall, qui prend en entrée un graphe orienté valué simple et calcule la distance entre tout couple de sommets, même en présence de poids négatifs. On considère que les sommets sont les entiers de 1 à n .

```

fonction FloydWarshall(G):
    dist = matrice (n,n)
    pour i de 1 à n:
        pour j de 1 à n:
            s'il y a un arc (i,j) dans G:
                dist[i,j] = poids[i,j]
            sinon:
                dist[i,j] = +∞
        dist[i,i] = 0
    pour k de 1 à n:
        pour i de 1 à n:
            pour j de 1 à n:
                dist[i,j] =
                    min(dist[i,j], dist[i,k]+dist[k,j])
    renvoyer dist
    
```

Exercice 4 (Algo de Floyd-Warshall).

1. Appliquer l'algorithme de Floyd-Warshall sur le graphe ci-contre. Dessiner l'état de la matrice de distances à chaque itération.



— État initial de la matrice (i correspond à la ligne, j à la colonne) :

dist	1	2	3	4
1	0	-2	+∞	+∞
2	+∞	0	2	+∞
3	+∞	+∞	0	-1
4	4	3	+∞	0

Ce sont les valeurs des plus courts chemins de i à j qui n'utilisent pas de sommet intermédiaire.

- À l'itération $k = 1$, l'algo calcule les valeurs des plus courts chemins de i à j de longueur 1 ou moins qui n'utilisent que le sommet 1 comme sommet intermédiaire :

dist	1	2	3	4
1	0	-2	$+\infty$	$+\infty$
2	$+\infty$	0	2	$+\infty$
3	$+\infty$	$+\infty$	0	-1
4	4	2	$+\infty$	0

Il a trouvé un nouveau plus court chemin de 4 à 2, (4, 1, 2). NB : c'est une façon de parler, car l'algo de base ne calcule pas d'arbre de PCC (on peut l'adapter).

- À l'itération $k = 2$, l'algo essaie le sommet 2 comme sommet intermédiaire, et calcule donc la valeur des plus courts chemins de i à j de longueur 2 ou moins qui n'utilisent que les sommets 1 ou 2 comme sommets intermédiaires :

dist	1	2	3	4
1	0	-2	0	$+\infty$
2	$+\infty$	0	2	$+\infty$
3	$+\infty$	$+\infty$	0	-1
4	4	2	4	0

En s'autorisant à passer par 2, l'algo a découvert un chemin de 1 à 3 et un autre de 4 à 3.

- À l'itération $k = 3$, l'algo essaie le sommet 3 comme sommet intermédiaire, et calcule donc la valeur des plus courts chemins de i à j de longueur 3 ou moins qui n'utilisent que les sommets 1, 2 ou 3 comme sommets intermédiaires :

dist	1	2	3	4
1	0	-2	0	-1
2	$+\infty$	0	2	1
3	$+\infty$	$+\infty$	0	-1
4	4	2	4	0

En s'autorisant à passer par 3, l'algo a découvert un chemin de 2 à 4 et un autre de 1 à 4. (1, 2, 3, 4). NB : l'algo a « trouvé un chemin » avec deux sommets intermédiaires.

- À l'itération $k = 4$, l'algo essaie le sommet 4 comme sommet intermédiaire, et calcule donc la valeur de tous les plus courts chemins de i à j de longueur 4 ou moins, avec n'importe quels sommets intermédiaires :

dist	1	2	3	4
1	0	-2	0	-1
2	5	0	2	1
3	3	1	0	-1
4	4	2	4	0

2. On considère le même graphe, sauf que le poids de l'arc (2, 3) est -3 . Appliquer l'algorithme à nouveau.

TODO

3. Qu'est-ce qui, dans la matrice résultante, nous permet de déduire qu'il y a (au moins) un cycle absorbant ?

Une valeur négative dans la diagonale. Si on a réussi à améliorer la distance de s à s , c'est qu'il y a un chemin de poids négatif qui part de s et revient à s , c'est-à-dire un cycle absorbant. Réciproquement, s'il y a un cycle absorbant, il y en a forcément un de longueur au plus n (en enlevant les sommets doublons). On verra donc forcément

apparaître un ou plusieurs poids négatifs sur la diagonale avant la fin de l'algo.

Exercice 5.

1. Quelle est la complexité temporelle de l'algorithme de Floyd-Warshall ?

$O(|S|^3)$

2. Si on appliquait simplement Bellman-Ford en considérant tour à tour chaque sommet comme sommet source, quelle complexité aurait cet algorithme ? Conclure sur l'utilité de Floyd-Warshall selon que le graphe est creux ou dense (voir question 4 de l'exo 2).

Bellman-Ford répété est en $O(|S|^2 \cdot |A|)$ donc si le graphe est creux ($|A| \in O(|S|)$) ça fait $O(|S|^3)$, donc c'est équivalent à Floyd-Warshall. Cependant si le graphe est dense ($|A| \in O(|S|^2)$) ça fait $O(|S|^4)$, donc Floyd-Warshall est beaucoup plus efficace !

3. Si le graphe n'a pas de poids négatifs, est-ce que se baser sur Dijkstra (en le répétant avec chaque sommet comme sommet source) peut être plus efficace que Floyd-Warshall ? À quelles conditions sur le graphe ?

Dijkstra répété est en $O((|S| + |A|) \log(|S|))$ donc $O(|S|^2 \log(|S|))$ si le graphe est creux, ce qui est mieux que le $O(|S|^3)$ de Floyd-Warshall. Si le graphe est dense, on a du $O(|S|^3 \log(|S|))$, ce qui est moins bien que Floyd-Warshall.

Correction TD 16 : Coloration de graphes

Exercice 1.

1. Soit G un graphe non orienté, et soit G' un sous-graphe de G . Montrer que $\chi(G) \geq \chi(G')$ (le nombre chromatique d'un sous-graphe est un minorant du nombre chromatique de G).

Notons $n = \chi(G)$. On peut colorer G avec seulement n couleurs de façon à ce que les sommets voisins soient de couleur différente. Mais les sommets qui sont voisins dans G' le sont aussi forcément dans G (puisque G' est un sous-graphe de G). Si on prend la n -coloration de G et qu'on la restreint aux sommets de G' , on obtient donc une coloration de G' avec n couleurs (ou moins). Par définition du nombre chromatique de G' , cela signifie que $n \geq \chi(G')$.

2. Montrer que tout graphe contenant un triangle (c'est-à-dire, qui a un sous-graphe isomorphe à K_3) ne peut pas être coloré en moins de trois couleurs.

Le nombre chromatique de K_3 est 3 : on peut le colorier avec 3 couleurs mais pas moins. Donc tout graphe qui a K_3 comme sous-graphe est forcément de nombre chromatique supérieur ou égal à 3, par le résultat de la question précédente.

3. Adapter la preuve précédente pour montrer que pour tout $n \in \mathbb{N}$, tout graphe contenant K_n ne peut pas être coloré en moins de n couleurs.

Le nombre chromatique de K_n est n : on peut le colorier avec n couleurs mais pas moins, car sinon il y aurait deux sommets de même couleur, or chaque sommet est voisin de tous les autres.

Par conséquent tout graphe qui a K_n comme sous-graphe est forcément de nombre chromatique supérieur ou égal à n .

Rappelons l'algorithme de Welsh-Powell, un algorithme de coloration glouton qui donne souvent de bons résultats, mais n'assure pas que le nombre de couleurs utilisé soit minimal.

```
fonction WelshPowell(G):  
  trier les sommets de G par ordre décroissant de leur degré  
  tant que tous les sommets ne sont pas colorés:  
    choisir une nouvelle couleur K  
    pour chaque sommet s non coloré (dans l'ordre choisi):  
      si aucun voisin de s n'est de couleur K:  
        colorer s avec la couleur K
```

Exercice 2 (Adapté de P. Boizumault). Un lycée doit prévoir les horaires des examens. Il y a 8 épreuves à planifier :

- | | |
|-------------|------------------|
| 1. Français | 5. Philosophie |
| 2. EPS | 6. Mathématiques |
| 3. SVT | 7. SES |
| 4. Physique | 8. Poney |

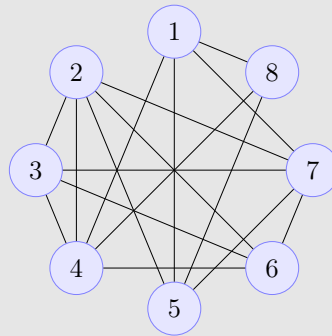
Les paires de cours suivantes ont des élèves communs :

- Français et physique, français et philo, français et SES, français et poney
- EPS et SVT, EPS et physique, EPS et philo, EPS et maths, EPS et SES
- SVT et physique, SVT et maths, SVT et SES
- Physique et maths, physique et poney
- Philo et SES, philo et poney
- Maths et SES

Le lycée souhaite planifier les examens sur un nombre minimal de créneaux (noté nb_{opt}).

1. Modéliser ce problème sous la forme d'un problème de coloration des sommets d'un graphe (à dessiner). Indiquer quels sont les sommets, les arêtes, et à quoi correspond nb_{opt} .

La question revient à chercher le nombre minimal de couleurs permettant de colorier le graphe ci-dessous, dont les sommets sont les épreuves (numérotées de 1 à 8), et une arête relie deux sommets lorsque les deux cours correspondants ont des élèves en commun. Les couleurs correspondent aux créneaux possibles pour les épreuves : deux épreuves ayant la même couleur ne partagent pas d'arête, donc n'ont pas d'élève en commun, et peuvent ainsi être programmées en parallèle sur le même créneau. Le nombre minimal de créneaux nécessaires est donc le nombre chromatique du graphe.



2. En analysant la structure du graphe, proposer un minorant de nb_{opt} .

L'ensemble de sommets $\{2, 3, 6, 7\}$ (ou $\{2, 3, 4, 6\}$) est une clique de taille 4, donc $nb_{opt} \geq 4$.

3. Appliquer l'algorithme de Welsh-Powell sur ce graphe, en détaillant les différentes étapes. On ordonnera les sommets de degré égal **en suivant le numéro du cours correspondant**. Combien de couleurs comporte la solution trouvée par l'algorithme ? Expliquer pourquoi ce nombre est un majorant de nb_{opt} .

On classe les sommets dans l'ordre décroissant de leurs degrés ; à degré égal on prend l'ordre naturel des numéros des cours. Ensuite on choisit une couleur C et on parcourt les sommets. Pour chacun, s'il n'est pas encore coloré et n'est pas adjacent à un sommet de couleur C , on lui attribue la couleur C . On recommence le parcours avec une autre couleur, tant qu'il reste un sommet non coloré.

sommet (degré)	1er parcours	2e parcours	3e parcours	4e parcours
2 (5)	bleu	rouge		
4 (5)	voisin de 2			
7 (5)	voisin de 2			
1 (4)	bleu	vert	vert	
3 (4)	voisin de 2			
5 (4)	voisin de 2,1			
6 (4)	voisin de 2	voisin de 4,7	voisin de 3	jaune
8 (3)	voisin de 1	voisin de 4	voisin de 5	jaune

On trouve une coloration avec 4 couleurs. Comme nb_{opt} est la taille minimum d'une coloration, il est majoré par la taille de n'importe quelle coloration. Ainsi $nb_{opt} \leq 4$.

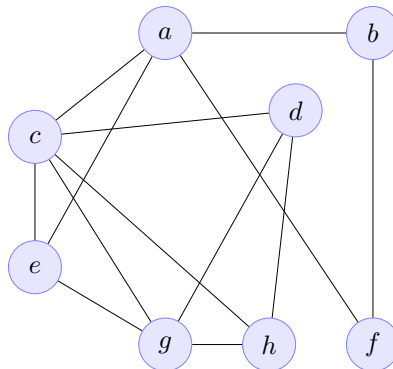
4. Dédurre des deux questions précédentes la valeur de nb_{opt} .

Selon la question 2, il faut au moins 4 couleurs, et selon la question 3, il faut au plus 4 couleurs. Par conséquent $nb_{opt} = 4$.

Exercice 3. Prouver que K_5 n'est pas planaire, en utilisant le théorème des quatre couleurs.

Le théorème des quatre couleurs dit que tout graphe planaire peut être coloré en quatre couleurs (ou moins). On sait que K_5 nécessite cinq couleurs (exercice 1), donc il ne peut pas être planaire (c'est la contraposée du théorème).

Exercice 4. On appelle G le graphe non orienté suivant.

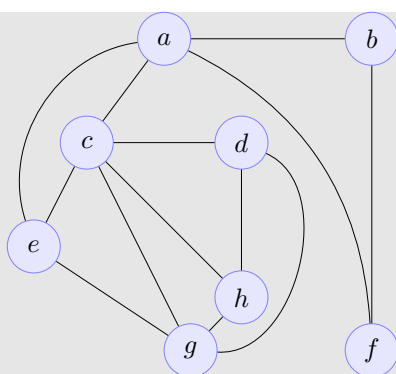


1. En analysant la structure du graphe, proposer un minorant de $\chi(G)$.

On voit que G contient K_4 (les sommets $\{c, d, g, h\}$ forment une clique), donc $\chi(G) \geq 4$.

2. Redessiner le graphe en modifiant le tracé des arêtes (ou la position des sommets), de façon à pouvoir déduire un majorant de $\chi(G)$, et en déduire la valeur de $\chi(G)$.

Si on fait passer l'arête $\{a, f\}$ au-dessus de d , l'arête $\{a, e\}$ au-dessus de c , et l'arête $\{d, g\}$ entre h et f , on obtient le résultat suivant :



Le graphe est donc planaire, ce qui signifie qu'il est k -coloriable pour un $k \leq 4$: $\chi(G) \leq 4$. Comme on avait $\chi(G) \geq 4$ on en déduit que le nombre chromatique de G est 4.

3. Appliquez l'algorithme de Welsh-Powell (en prenant l'ordre alphabétique lorsque des choix sont possibles) pour obtenir une coloration du graphe. Est-elle optimale ?

sommet (degré)	1er parcours	2e parcours	3e parcours	4e parcours
c (5)	rouge			
a (4)	voisin de c	bleu		
g (4)	voisin de c	bleu		
b (3)	rouge			
d (3)	voisin de c	voisin de g	vert	
h (3)	voisin de c	voisin de g	voisin de d	jaune
e (3)	voisin de c	voisin de a	vert	
f (2)	voisin de b	voisin de a	vert	

On trouve une 4-coloration. Elle est optimale puisque 4 est le nombre chromatique du graphe.

4. Arrivez-vous à faire augmenter le nombre chromatique de G en lui ajoutant une seule arête ? Et deux ?

Avec une seule arête ce n'est pas possible (le graphe reste toujours planaire). Avec deux arêtes, on peut faire apparaître K_5 comme sous-graphe (qui nécessite 5 couleurs).

Exercice 5 (Adapté de P. Boizumault). On a vu que tout graphe contenant K_n (c'est-à-dire, qui a un sous-graphe isomorphe à K_n) ne peut pas être coloré en moins de n couleurs. Autrement dit, contenir K_n est une condition suffisante pour nécessiter au moins n couleurs. On va voir maintenant que ce n'est pas une condition nécessaire.

1. Construire un graphe sans triangle qui nécessite trois couleurs.

Il suffit de considérer un cycle ayant un nombre impair de sommets, par exemple C_5 .

2. Comment, à partir du graphe précédent, construire un graphe nécessitant 4 couleurs **sans** contenir K_4 ?

Si l'on rajoute à ce graphe un sommet universel (relié à tous les sommets du cycle), il faudra nécessairement une 4e couleur pour le colorer. Le graphe obtenu est donc de nombre chromatique 4, tout en contenant pas K_4 .

3. Comment construire un graphe sans K_5 nécessitant 5 couleurs ? Peut-on continuer pour tout n ?

On peut itérer cette construction de façon à obtenir, pour tout n , un graphe de nombre chromatique n ne contenant pas K_n .

Exercice 6 (Exercice de P. Boizumault). Exprimez la résolution d'un Sudoku (classique) en termes de coloration de graphe, en décrivant le graphe (nombre de sommets, nombre d'arêtes...). Combien faut-il de couleurs ?

Résoudre un Sudoku classique revient à colorer un graphe à 81 sommets (un sommet correspondant à une case) avec 9 couleurs, certains sommets étant colorés au départ. Ce graphe a 810 arêtes, chaque case (sommets) étant reliée - aux 8 autres de sa colonne, - aux 8 autres de sa ligne - et aux 8 autres de sa région (4 sont déjà comptées), ce qui fait bien $(81 \cdot (8 + 8 + 4))/2 = 810$ arêtes.

Correction TD 17 : Arbres couvrants de poids minimal

On rappelle ci-dessous l'algorithme de Prim, qui prend en entrée un graphe non orienté valué simple connexe et un sommet source, et qui construit un arbre couvrant de poids minimal.

```

fonction Prim(G, source):
    priorité, pred = dictionnaires vides
    nonVisités = {}
    pour chaque sommet s de G:
        priorité[s] = +∞
        pred[s] = nil
        ajouter s à nonVisités
    priorité[source] = 0

    tant que nonVisités n'est pas vide:
        prendre le sommet s de nonVisités tq priorité[s] est minimale
        retirer s de nonVisités

        pour chaque successeur s' de s:
            si s' est dans nonVisités et poids[s,s'] < priorité[s']:
                pred[s'] = s
                priorité[s'] = poids[s,s']
    renvoyer pred

```

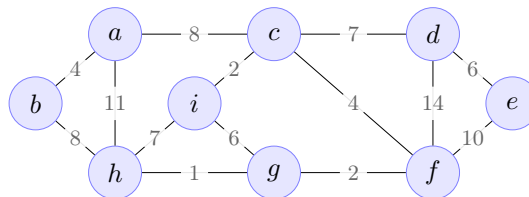
On rappelle maintenant l'algorithme de Kruskal, qui prend en entrée un graphe non orienté valué simple connexe et qui construit un arbre couvrant de poids minimal.

```

fonction Kruskal(G):
    composantes = {} (ensemble d'ensembles de sommets)
    T = {} (ensemble d'arêtes)
    pour chaque sommet s de G:
        ajouter {s} à composantes
    pour chaque arête (s,s') de G, dans l'ordre croissant des poids:
        si s et s' ne sont pas dans la même composante:
            ajouter (s,s') à T
            fusionner les composantes de s et s'
    renvoyer T

```

Exercice 1. On considère le graphe non orienté simple valué connexe suivant :



1. Dérouler l'algorithme de Prim sur ce graphe, en partant du sommet e , et en notant à chaque itération l'état des tableaux `priorité` et `pred`, le sommet visité s et l'arête choisie ($\{s, \text{pred}[s]\}$), et un schéma de l'état courant de l'arbre construit (c'est-à-dire l'arbre constitué par les sommets visités et les arêtes choisies). Si deux sommets sont de priorité minimale, choisir d'abord le premier dans l'ordre alphabétique.

sommet visité	/	e	d	c	i	f	g	h	a	b
arête choisie	/	/	de	cd	ci	cf	fg	gh	ac	ab
priorité[a]	+oo			8						
priorité[b]	+oo							8	4	
priorité[c]	+oo		7							
priorité[d]	+oo	6								
priorité[e]	0									
priorité[f]	+oo	10		4						
priorité[g]	+oo				6	2				
priorité[h]	+oo				7		1			
priorité[i]	+oo			2						
pred[a]	nil									
pred[b]	nil							h	a	
pred[c]	nil		d							
pred[d]	nil	e								
pred[e]	nil									
pred[f]	nil	e		c						
pred[g]	nil				i	f				
pred[h]	nil				i		g			
pred[i]	nil			c						

2. Dérouler l'algorithme de Kruskal sur ce graphe, en notant à chaque itération l'arête considérée, si elle est choisie ou rejetée, l'état de la partition des sommets et un schéma de l'état courant de la forêt construite. À poids égal, on considérera les arêtes dans l'ordre lexicographique ($\{a, b\}$, $\{a, c\}$, $\{a, h\}$, $\{b, h\}$, $\{c, d\}$, $\{c, f\}$, $\{c, i\}$, $\{d, e\}$, ...).

J'écris les ensembles (et les arêtes) comme des mots, pour simplifier :

0. début. a | b | c | d | e | f | g | h | i
1. choix de gh (1). a | b | c | d | e | f | gh | i
2. choix de ci (2). a | b | ci | d | e | f | gh
3. choix de fg (2). a | b | ci | d | e | fgh
4. choix de ab (4). ab | ci | d | e | fgh
5. choix de cf (4). ab | cfghi | d | e
6. choix de de (6). ab | cfghi | de
7. rejet de gi (6).
8. choix de cd (7). ab | cdefghi
9. rejet de hi (7).
10. choix de ac (8). abcdefghi

Toutes les arêtes suivantes sont rejetées puisqu'il n'y a plus qu'une seule composante.

Exercice 2 (Adapté de M. Langer et R. Sedgewick).

1. Considérons un graphe simple non orienté valué connexe. Si on ajoute 42 au poids de chacune de ses arêtes, cela change-t-il l'arbre construit par l'algorithme de Prim ?

Non : l'algorithme choisit toujours l'arête de poids minimal. Ajouter une constante à tous les poids ne change pas l'ordre de sélection.

2. Même question avec l'algorithme de Kruskal.

Même réponse : l'algorithme considère les arêtes dans l'ordre croissant des poids. Cet ordre est le même si on ajoute la même constante à chaque poids.

3. Considérons un graphe simple non orienté valué connexe dont les valeurs peuvent être positives ou négatives. Si l'on fait tourner l'algorithme de Prim ou de Kruskal dessus, est-ce que le résultat est bien un arbre couvrant de poids minimal ?

Oui : supposons qu'on ajoute une très grande constante à tous les poids, afin que tous deviennent positifs. Par les questions précédentes, l'arbre construit par Prim (resp. Kruskal) serait le même, et il est clair que si cet arbre est de poids minimal pour le graphe « augmenté », il l'est aussi pour le graphe d'origine.

4. Supposons que l'on souhaite trouver des arbres couvrants de poids *maximal*. Peut-on utiliser les algorithmes de Prim et/ou Kruskal (éventuellement en les adaptant un peu) ?

Toujours pour les mêmes raisons, il suffit d'inverser le signe de tous les poids le temps de faire tourner un des deux algos. L'arbre construit sera de poids minimal pour le graphe inversé, donc de poids maximal pour le graphe d'origine.

5. Considérons un graphe simple non orienté valué connexe G . Supposons qu'on modifie les poids des arêtes de G pour obtenir G' . Parmi les modifications suivantes de G , quelles sont celles pour lesquelles l'arbre construit par les algorithmes de Prim ou Kruskal sur G' est aussi de poids minimal pour G (avec les poids d'origine) ? On supposera d'abord que les poids sont tous positifs, puis qu'ils peuvent être positifs ou négatifs.

- (a) $\text{poids}_{G'}(e) = 17 \times \text{poids}_G(e)$.
- (b) $\text{poids}_{G'}(e) = \text{poids}_G(e) \times \text{poids}_G(e)$.
- (c) $\text{poids}_{G'}(e) = \text{poids}_G(e) \times \text{poids}_G(e) \times \text{poids}_G(e)$.

Si les poids d'origine sont tous positifs, alors aucune des trois modifications ne changent le fait que l'arbre construit est de poids minimal, toujours pour la même raison : ce qui compte c'est l'ordre des poids, et ces modifications ne changent pas l'ordre.

C'est pareil si les poids d'origine peuvent être positifs ou négatifs pour les modifications (a) et (c). En revanche, la 2e modification peut alors changer l'ordre, car la fonction $x \mapsto x^2$ n'est pas croissante sur \mathbb{R} . Par exemple, si la liste ordonnée des poids est $(-7, -5, -3, -2, 1)$, elle deviendra $(49, 25, 9, 4, 1)$ après la modification, et les arêtes seront donc considérées dans l'ordre exactement inverse !

Exercice 3. On considère des graphe non orienté simple valué connexes.

1. Montrer la « propriété des coupes » : pour une coupe donnée d'un graphe G , une arête traversante a de poids strictement inférieur à toutes les autres arêtes traversantes appartient forcément à tout arbre couvrant de poids minimal de G .

Par l'absurde, supposons qu'il existe un ACPM T qui ne contient pas a . Si on ajoute a à T , on obtient nécessairement un cycle. Ce cycle « franchit » forcément la coupe deux fois : via a et via une autre arête a' . Si maintenant on enlève a' de T , le résultat $T' = T - a' + a$ est un arbre (puisque'il y a le bon nombre d'arêtes) couvrant (on n'a pas enlevé de sommet) de poids strictement inférieur à T (puisque le poids de a est strictement inférieur au poids de a' par hypothèse de départ). Cela contredit le fait que T est de poids minimal. Tout ACPM doit donc contenir a .

2. Prouver qu'un graphe non orienté simple valué connexe dont tous les poids sont distincts (il n'y a pas d'arêtes qui ont le même poids) possède un unique arbre couvrant de poids minimal.

Supposons, par l'absurde, qu'il existe deux ACPM distincts, T_1 et T_2 . Puisqu'ils sont distincts et ont les mêmes sommets et le même nombre d'arêtes (car ce sont des arbres), il y a forcément au moins une arête de T_1 qui n'est pas dans T_2 . Parmi ces arêtes, appelons a celle qui a le plus petit poids.

Si on supprime a de T_1 , le graphe résultant a forcément deux composantes connexes. Soit la coupe correspondant aux deux ensembles de sommets associés. Parmi les arêtes traversantes de cette coupe, a est forcément celle de poids minimum (car sinon, T_1 ne serait pas de poids minimum), et il ne peut pas y avoir d'autre arête de poids minimum (par hypothèse de départ, toutes les arêtes sont de poids distincts).

On obtient donc que a est de poids strictement inférieur à toutes les autres arêtes traversantes de cette coupe, et doit donc appartenir à tout ACPM, ce qui contredit le fait qu'elle n'appartient pas à T_2 .

On a donc montré par l'absurde qu'il ne peut pas exister deux ACPM distincts.

3. Montrer la « propriété des cycles » : pour tout cycle donné d'un graphe G , une arête qui a un poids strictement supérieur à tous les autres du cycle n'appartient à aucun arbre couvrant de poids minimal.

Soit c un cycle de G , et soit a une arête de c qui est de poids strictement supérieur à toutes les autres arêtes du cycle. Par l'absurde, supposons que a appartient à un ACPM T . Considérons le sous-graphe ayant les arêtes $T \setminus \{a\}$, qui a deux composantes connexes. Puisque c est un cycle contenant a , il y a forcément une autre arête a' sur le cycle qui connecte les deux composantes. Par hypothèse, le poids de a' est strictement inférieur à celui de a . Mais alors l'arbre $T \setminus \{a\} \cup \{a'\}$ est un arbre couvrant de poids strictement inférieur à T , ce qui contredit notre hypothèse que T est un ACPM.

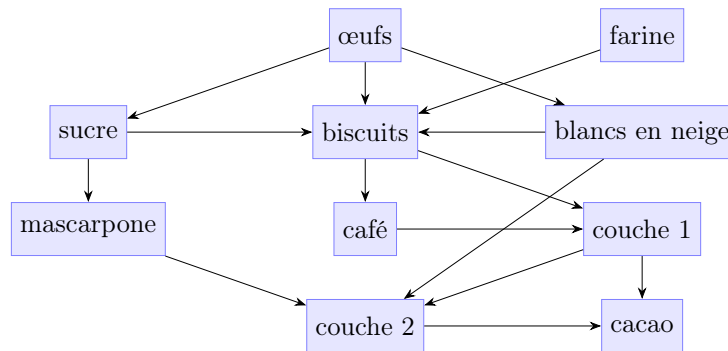
Exercice 4 (Adapté de M. Langer). Supposons qu'on s'intéresse à un ensemble de n objets entre lesquels on a une notion de distance (ça peut être une distance physique, ou autre chose, par exemple une distance d'édition entre des mots). On peut représenter les distances par des poids dans un graphe pondéré. Supposons qu'on veuille faire du *clustering* sur les objets, c'est-à-dire qu'on souhaite grouper les objets en k groupes (*clusters*) de sorte que la distance entre les objets d'un groupe soit plutôt petite, et la distance entre des objets de groupes différents soit plutôt grande.

Proposer une adaptation de l'algorithme de Kruskal pour séparer les n sommets d'un graphe (non orienté simple connexe valué) en k groupes.

On peut tout simplement dérouler Kruskal en s'arrêtant après avoir choisi $n - k$ arêtes. En effet, on part avec n composantes, et chaque arête choisie fusionne deux composantes, c'est-à-dire qu'elle réduit de 1 le nombre de composantes. Après avoir choisi $n - k$ arêtes, on a donc $n - (n - k) = k$ composantes. De plus, si on note p le poids de la dernière arête choisie, alors on sait que toutes les arêtes choisies, qui relient deux sommets d'une même composante, sont de poids inférieur à p , et que toutes les arêtes qui relient deux sommets de composantes distinctes sont de poids supérieur à p (car sinon elles auraient été choisies). On a donc un *clustering* tel que la distance entre toute paire d'objets de même groupe est inférieure à la distance entre toute paire d'objets de groupes différents.

Correction TD 18 : Ordonnancement

Exercice 1 (Exercice de P. Boizumault). Le graphe ci-dessous représente une (pseudo-)recette de tiramisù. Chaque arc indique une contrainte de précédence : il faut avoir terminé l'étape représentée par le sommet origine avant de pouvoir commencer l'étape du sommet destination.



- Donner l'ordre des étapes de la recette construit par l'algorithme de tri topologique vu en cours, en considérant que les sommets et les voisins sont toujours parcourus dans l'ordre alphabétique.

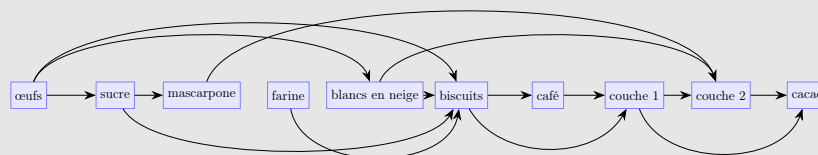
Pour rappel, l'algorithme vu en cours :

```

fonction triTopologique(G):
  créer une pile p vide
  pour chaque sommet s de G:
    si s est blanc:
      DFS(G, s)
  tant que p n'est pas vide:
    dépiler s de p
    afficher s

fonction DFS(G, s):
  colorer s en gris
  pour tout successeur s' de s:
    si s' est blanc:
      DFS(G, s')
  colorer s en noir
  empiler s dans p
  
```

L'ordre trouvé est (1) œufs (2) sucre (3) mascarpone (4) farine (5) blancs en neige (6) biscuits (7) café (8) couche 1 (9) couche 2 (10) cacao On peut représenter le graphe en mettant les sommets sur une ligne, de droite à gauche dans l'ordre topologique trouvé : on constate que chaque arc est toujours dirigé vers la droite.



- Trouver deux autres ordres topologiques distincts.

On peut notamment trouver des ordres différents facilement en faisant remonter la farine plus haut dans la liste (à n'importe quelle position, puisque la seule contrainte est qu'elle doit être avant les blancs en neige) ou en faisant descendre le mascarpone plus bas (à n'importe quelle position tant que c'est avant la couche 2).

Exercice 2. Soit $G = (S, A)$ un graphe orienté. Un ordre total $<$ sur S est un *ordre topologique* de G s'il respecte la condition suivante : pour tout couple de sommets (s, s') , s'il y a un arc de s à s' dans G , alors $s < s'$.

1. Démontrer que si G admet un ordre topologique, alors G est un DAG.

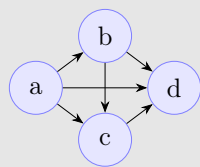
Supposons que G admet un ordre topologique. Par l'absurde, supposons que G contient un cycle orienté, dont on note (s_1, \dots, s_k) la suite de sommets associée. Pour tout $1 \leq i \leq k-1$, il y a un arc de s_i à s_{i+1} (par définition d'un cycle orienté), et donc $s_i < s_{i+1}$ puisque $<$ est un ordre topologique. Cela implique donc en particulier que $s_1 < s_k$. Or, un cycle orienté étant un chemin orienté fermé, on sait que $s_1 = s_k$, ce qui est contradictoire. Il est donc impossible que G contienne un cycle orienté : c'est nécessairement un DAG.

2. Démontrer que si G est un DAG, alors il admet un ordre topologique. On procédera par récurrence sur le nombre de sommets n de G , en utilisant le fait qu'un DAG a toujours au moins un puits.

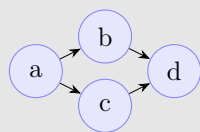
On va prendre $n = 1$ comme cas de base. Il y a un seul DAG avec un seul sommet, et le seul ordre possible sur son ensemble de sommets est bien topologique (par vacuité). Supposons que la propriété est vraie pour tout DAG avec k sommets, et montrons qu'elle l'est pour tout DAG avec $k+1$ sommets. Soit G un DAG avec $k+1$ sommets ; il a forcément (au moins) un puits. Choisissons-en un et notons le p . Soit G' le graphe obtenu à partir de G en enlevant p et tous ses arcs entrants. On n'a pas pu créer de cycle en enlevant des arêtes, donc G' est acyclique, et en outre il a k sommets. Par l'hypothèse de récurrence, il admet donc un ordre topologique (s_1, s_2, \dots, s_k) . On peut en déduire un ordre topologique sur G , en plaçant p à la fin : $(s_1, s_2, \dots, s_k, p)$.

Exercice 3 (Adapté de M. Langer). La définition d'un ordre topologique d'un graphe orienté $G = (S, A)$ est un ordre total $<$ sur S tel que s'il y a un arc de s à s' dans G , alors $s < s'$.

1. Trouver un graphe (avec au moins 4 sommets) et un ordre topologique sur ce graphe tel que l'énoncé réciproque est également vrai, c'est-à-dire que pour tout couple de sommets (s, s') , si $s < s'$ suivant l'ordre, alors il y a un arc de s à s' .



2. Trouver un graphe (avec au moins 4 sommets) et un ordre topologique sur ce graphe tel que, cette fois, l'énoncé réciproque est *faux*.



Avec l'ordre $a < b < c < d$, l'énoncé réciproque est faux : le couple

de sommets (b, c) (par exemple) est tel que $b < c$ et pourtant il n'y a pas d'arc de b à c .

3. Pouvez-vous trouver un graphe qui admet (au moins) un ordre topologique pour lequel l'énoncé réciproque est vrai, et (au moins) un ordre topologique pour lequel il est faux ?

Non. En effet, il est facile de voir que l'énoncé réciproque implique l'unicité de l'ordre topologique. (*à développer*)

Exercice 4 (Adapté de M. Langer). Quelle est la taille maximale d'un DAG simple d'ordre n ?

Soit G un DAG simple d'ordre n qui soit de taille maximale. On sait que G DAG admet forcément au moins un ordre topologique : choisissons-en un et notons ses sommets (s_1, \dots, s_n) suivant cet ordre.

Soient $1 \leq i < j \leq n$ deux indices de sommets. Puisque $s_i < s_j$ selon notre ordre topologique, s'il n'y avait pas d'arc de s_i à s_j , on pourrait en mettre un sans créer de cycle. Comme G est de taille maximale, il contient donc un arc de s_i à s_j .

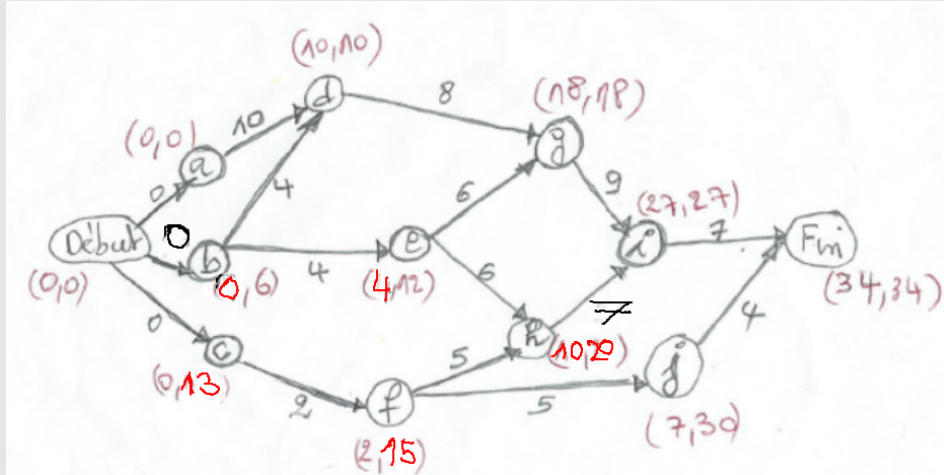
Au final, on voit que s_1 a $n - 1$ arcs sortants, s_2 en a $n - 2$, etc., s_{n-1} en a un seul et s_n aucun. La taille de G est donc $(n - 1) + (n - 2) + \dots + 1 + 0 = \sum_{i=0}^{n-1} i = n(n - 1)/2$.

Exercice 5 (Adapté de P. Boizumault). On considère le problème d'ordonnancement suivant. Les durées sont données en nombre de jours.

Tâches	a	b	c	d	e	f	g	h	i	j
Durées	10	4	2	8	6	5	9	7	7	4
Précédence	-	-	-	a, b	b	c	d, e	e, f	g, h	f

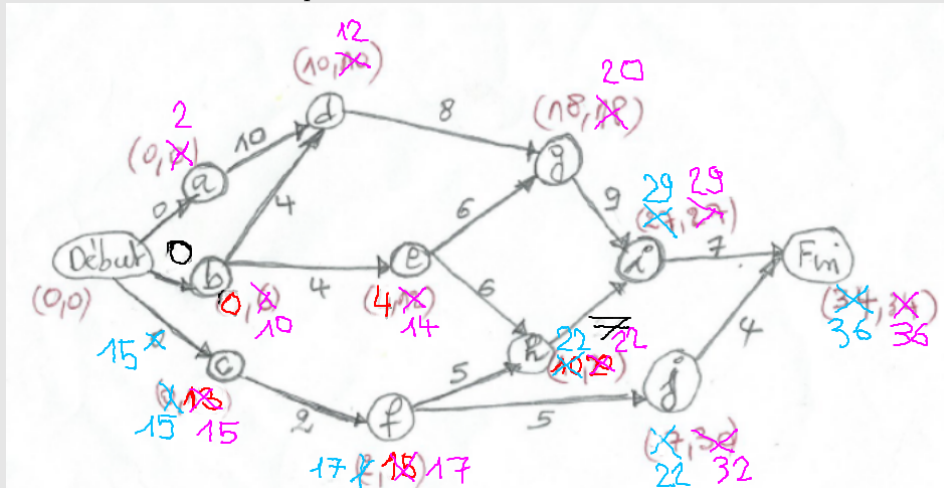
- Construire le graphe d'ordonnancement (« PERT à potentiels tâches ») de ce problème : les sommets sont les tâches, les arcs sont les contraintes de précédence et sont valués par la durée de la tâche de leur sommet origine. Il doit y avoir deux tâches fictives, **Start** et **End**, avec des arcs de valuation 0 qui font en sorte que **Start** soit la seule source et **End** le seul puits du graphe.
- Calculer la date de début au plus tôt (EST, *earliest starting time*) de chaque tâche, et l'écrire près du sommet associé.
- En déduire la durée minimale du projet.
- Calculer la date de début au plus tard (LST, *latest starting time*) de chaque tâche (c'est-à-dire la date la plus tardive à laquelle on peut faire commencer la tâche sans augmenter la durée du projet), et l'écrire près du sommet associé.
- Calculer la marge totale de chaque tâche, c'est-à-dire la différence entre ses dates de début au plus tard et au plus tôt. Il s'agit du retard maximal qu'on peut prendre sur la tâche sans augmenter la durée du projet.
- Une tâche dont la marge totale est 0 est dite *critique*. Mettre en valeur les tâches critiques sur le graphe.
- Un chemin de **Start** à **End** qui ne passe que par des tâches critiques est appelé *chemin critique*. Mettre en valeur le chemin critique, et constater qu'il s'agit du plus long chemin de **Start** à **End**.
- Finalement, on apprend qu'il ne sera pas possible de commencer c avant le 15e jour.
 - Redessiner le graphe afin de prendre en compte cette nouvelle contrainte.
 - Le plus long chemin de **Start** à **End** est-il le même qu'avant ? Le mettre en valeur.
 - Recalculer les EST et LST et vérifier que les tâches critiques sont toujours exactement celles qui sont sur le plus long chemin.
- Le chemin critique est-il toujours unique dans un graphe d'ordonnancement ? Justifier votre réponse par une démonstration ou un contre-exemple.

Grappe de départ :



le chemin critique est début,a,d,g,i,fin.

Grappe avec début de c différé de 15 :



le chemin critique est début,c,f,h,i,fin.

Dernière question : il n'est pas forcément unique, il suffit qu'il y ait deux plus longs chemins. Par ex. si on ne met que 13 jours de retard à c (au lieu de 15), les deux chemins suscités sont critiques.

Exercice 6.

1. Écrire le pseudo-code d'une fonction `dureeMinimale(G, Start, End)` qui prend en entrée en graphe d'ordonnancement et ses deux sommets `Start` et `End`, et qui calcule la durée minimale du projet. La complexité au pire cas de votre algorithme doit être linéaire.
2. En vous inspirant de la question précédente, écrire le pseudo-code d'une fonction `plusLongsCheminsDag(G, source)` qui prend en entrée un DAG pondéré et un de ses sommets, et qui renvoie un arbre des plus longs chemins de `G` qui partent du sommet `source`.

Exercice 7. Soit G un graphe d'ordonnancement. On note $\pi(s, s')$ le poids du plus long chemin du sommet s au sommet s' dans G . Démontrer formellement les propositions suivantes.

1. La date de début au plus tôt (EST) d'une tâche t est égale à $\pi(\text{Start}, t)$.
2. La durée minimale du projet est égale à $\pi(\text{Start}, \text{End})$.
3. La date de début au plus tard (LST) d'une tâche t est égale à $\pi(t, \text{End}) - \pi(\text{Start}, \text{End})$.

4. Le long d'un plus long chemin de **Start** à **End**, tous les sommets sont des tâches critiques.
5. Si une tâche est critique, elle est forcément sur un plus long chemin de **Start** à **End**.

Correction TD 19 : Cycles hamiltoniens et voyageur de commerce

Exercice 1. Déterminer pour quelles valeurs de $n \in \mathbb{N}$ les graphes des familles suivantes sont hamiltoniens.

1. K_n , les graphes complets d'ordre n .

K_n est hamiltonien pour $n > 2$.

2. C_n , les graphes-cycles d'ordre n – on considérera que C_1 est le graphe à un sommet et une boucle, et C_2 le graphe à deux sommets et deux arêtes entre eux.

C_n est hamiltonien pour $n > 0$.

3. W_n , les roues d'ordre n , définies comme suit : W_0 est le graphe nul, et pour tout $n > 0$, on prend C_{n-1} et on ajoute un sommet universel (relié à tous les autres).

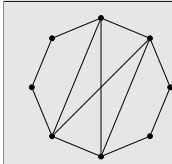
W_n est hamiltonien pour $n > 2$.

4. les arbres d'ordre $n > 0$.

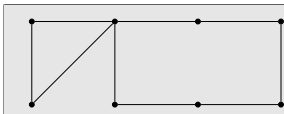
Les arbres sont acycliques, donc ils ne contiennent aucun cycle, encore moins hamiltonien, quel que soit leur ordre.

Exercice 2 (Exercice de Gross & Yellen). Pour chacune des descriptions suivantes, dessiner un graphe correspondant, ou prouver qu'il ne peut en exister.

1. Un GSNO d'ordre 8 et de taille au moins 8 qui est à la fois eulérien et hamiltonien.



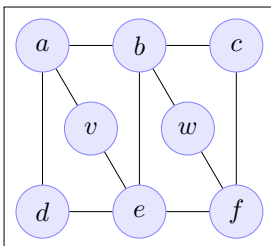
2. Un GSNO d'ordre 8 et de taille au moins 8 qui est eulérien, mais pas hamiltonien.



3. Un GSNO d'ordre 8 et de taille au moins 8 qui est hamiltonien, mais pas eulérien.

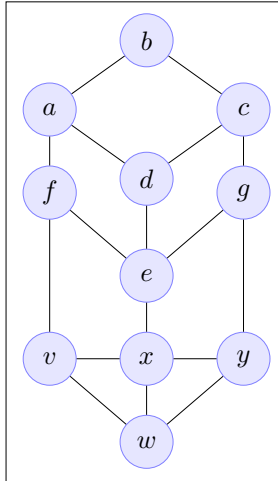
K_8 .

Exercice 3 (Exercice de Gross & Yellen). Pour chacun des graphes suivants, trouver un cycle hamiltonien ou prouver que le graphe n'est pas hamiltonien.



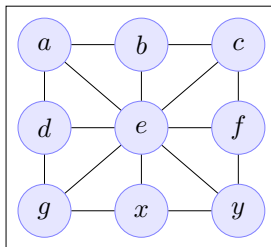
- 1.

c et w sont de degré 2, donc leurs arêtes incidentes appartiennent à tout cycle hamiltonien. Cependant la réunion de ces 4 arêtes forme un cycle qui ne contient pas tous les sommets. Il ne peut donc pas exister de cycle hamiltonien.



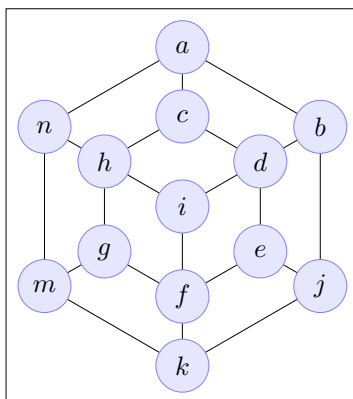
2.

$(a, b, c, d, e, g, y, x, w, v, f, a)$ est un cycle hamiltonien.



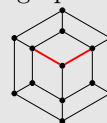
3.

$(a, b, c, f, y, x, g, d, e, a)$ est un cycle hamiltonien. (Le graphe est en fait W_9 .)



4.

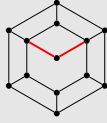
S'il y a un cycle hamiltonien, il passe forcément par exactement deux arêtes incidentes au sommet central i . On les choisit arbitrairement, le graphe étant symétrique (j'enlève



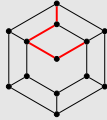
les noms des sommets pour que ce soit plus clair) :

La troisième arête incidente au sommet central ne peut pas appartenir à un cycle

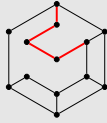
hamiltonien, on la supprime (règle 3) :



Regardons maintenant le sommet juste au-dessus des deux arêtes choisies : on doit choisir deux de ses arêtes incidentes. On est obligé de prendre l'arête verticale, car sinon il y aurait un cycle de longueur 4 (règle 2). On choisit la deuxième arête arbitrairement, l'autre choix est symétrique.



On peut supprimer de nouvelles arêtes grâce à la règle 3 :



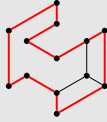
Cela révèle deux nouveaux sommets de degré 2, on choisit leurs arêtes incidentes (règle 1) :



Et on supprime les arêtes inutiles par la règle 3 :

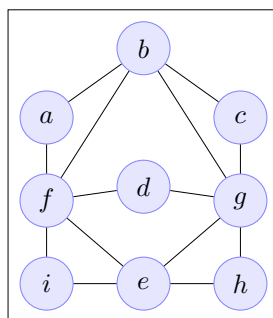


On a deux nouveaux sommets de degré 2, on applique la règle 1 :

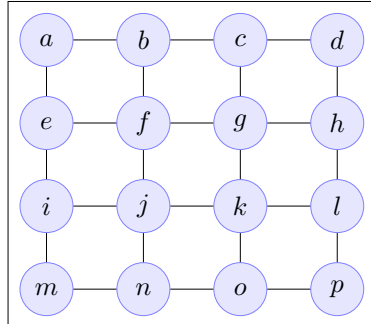


On trouve un cycle qui ne passe pas par tous les sommets, c'est impossible (règle 2), donc quand on a dû faire un choix, ce n'était pas le bon. Sauf que les autres choix donnaient une situation identique par symétrie. Il ne peut donc pas exister de cycle hamiltonien.

5.



d, i et h sont de degré 2, donc leurs 6 arêtes incidentes font partie de tout cycle hamiltonien. Cependant la réunion de ces arêtes forme un cycle qui ne contient pas tous les sommets. Il ne peut donc pas exister de cycle hamiltonien.



6.

$(a, e, f, j, i, m, n, o, p, l, k, g, h, d, c, b, a)$ est un cycle hamiltonien.

Exercice 4 (Exercice de Gross & Yellen). Le théorème d'Ore donne une condition suffisante pour qu'un graphe soit hamiltonien. Montrer que la propriété en question n'est pas une condition nécessaire, en donnant un graphe hamiltonien qui ne la respecte pas.

C_5 est un exemple de graphe hamiltonien qui ne respecte pas la condition du théorème d'Ore : tous les sommets sont de degré 2, donc la somme des degrés de deux sommets (voisins ou non) ne pourra jamais dépasser 4.

Exercice 5 (Adapté de P. Boizumault). Le problème du voyageur de commerce est NP-difficile, ce qui signifie que l'on estime très improbable qu'il existe un algorithme polynomial pour le résoudre. On peut cependant utiliser des *heuristiques* de choix des sommets, qui donnent généralement des résultats satisfaisants tout en étant très efficaces – la contrepartie étant que les résultats peuvent aussi être très mauvais.

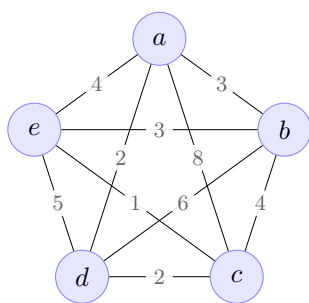
L'heuristique la plus simple pour le PVC est celle du *plus proche voisin* : on choisit toujours comme sommet suivant le sommet le plus proche du sommet courant (c'est un algorithme glouton).

```

fonction plusProcheVoisin(G, s):
    chemin = liste vide
    ajouter s à chemin
    colorer s
    u = s
    tant qu'il reste un sommet non coloré :
        soit v le sommet non coloré le plus proche de u
        ajouter v à chemin
        colorer v
        u = v
    ajouter s à chemin (pour fermer le cycle)
    retourner chemin

```

On considère G le graphe suivant :



1. Dérouler l'algorithme du plus proche voisin sur le graphe G en partant du sommet a . Quel est le cycle retourné? Quel est son poids? En observant les poids du graphe, montrer qu'aucun cycle hamiltonien ne peut être de poids inférieur, et que le cycle hamiltonien trouvé est donc bien minimal.

Cycle trouvé : (a, d, c, e, b, a) , de poids 11. Ce cycle emprunte les cinq arêtes les moins lourdes du graphe, il est donc impossible de trouver un cycle de poids inférieur.

2. En modifiant le poids d'une seule arête dans G , faire en sorte que l'algorithme du plus proche voisin partant de a trouve un cycle hamiltonien de poids 10000, alors qu'il existe des cycles hamiltoniens plus courts.

La faiblesse de cette heuristique est qu'en choisissant des petites arêtes au début, elle peut s'obliger à prendre des grosses arêtes plus tard. En particulier, l'algorithme ne se donne aucune marge de manœuvre quant au choix de la toute dernière arête. On peut se servir de cela pour augmenter arbitrairement le poids du cycle trouvé : en changeant le poids de l'arête $\{a, b\}$ en 9992, on ne change pas du tout le déroulement de l'algorithme au départ de a , mais le poids du cycle passe à 10000. Tous les cycles hamiltoniens ne passant pas par cette arête sont évidemment (beaucoup) plus courts.

3. Donner la complexité de l'algorithme du plus proche voisin en fonction du nombre n de sommets.

Chaque sommet est coloré une seule fois, donc il y a n itérations de la boucle. À chaque tour de boucle, il faut trouver le sommet non coloré le plus proche, ce qui nécessite d'étudier au plus n arêtes. On a donc une complexité en $O(n^2)$. (NB : même si on prend en compte le fait qu'il reste de moins en moins de sommets colorés au fur et à mesure des itérations, ça reste en $O(n^2)$.)

Exercice 6 (Exercice de P. Boizumault). Un atelier d'électronique doit produire une série de circuits imprimés tous identiques. Chaque plaque de circuit imprimé doit être percée de n trous P_i ($1 \leq i \leq n$) définis par leurs coordonnées (x_i, y_i) dans un repère orthonormé (O, x, y) relatif à la plaque. Les plaques arrivent en file (i.e. les unes après les autres) via une courroie transporteuse qui s'arrête le temps nécessaire sous une machine de perçage. La tête de perçage part du point origine O , effectue les n trous, puis revient à son point de départ. Elle se déplace à une vitesse de v cm par seconde et perce un trou en p secondes. Le responsable d'atelier souhaite déterminer l'ordre de perçage des n trous afin de minimiser la durée de traitement d'une plaque. Modéliser ce problème sous forme d'un PVC (on précisera : sommets, arêtes et fonction de poids) en supposant que la tête de perçage peut se déplacer : (a) dans n'importe quelle direction ; (b) uniquement horizontalement et verticalement.

- Sommets = $\{O\} \cup \{P_i \mid 1 \leq i \leq n\}$
- Arêtes : graphe complet
- Cycle hamiltonien : départ en O , visiter tous les P_i et revenir en O pour traiter la plaque suivante
- Fonction de poids : le poids de l'arête $\{P_i, P_j\}$ est la distance entre P_i et P_j
- Choix de la distance :
 1. dans n'importe quelle direction \rightarrow distance euclidienne entre P_i et P_j
 2. uniquement horizontalement et verticalement \rightarrow distance de Manhattan entre P_i et P_j