# DS1 PC : programmation dynamique; graphes

Les deux parties de ce sujet sont indépendantes. Quand une complexité est demandée, c'est toujours la complexité temporelle au pire.

Dans la suite, les graphes sont notés sous la forme $G = (V, E)$, où $V$ est l'ensemble des sommets et $E$ celui des arcs. On pose $n = |V|, p = |E|$. Si l'arc $x \to y$ est dans $E$, on dit que $y$ est *voisin* de $x$. En revanche, $x$ n'est pas nécessairement voisin de $y$ sauf dans le cas où le graphe est non orienté.

Dans un graphe non orienté, le *degré* d'un sommet est le nombre de ses voisins.

Par commodité, on suppose que les graphes considérés dans ce devoir ne possèdent pas de *boucle*, c'est-à-dire qu'un sommet n'est jamais son propre voisin.

Le sous-graphe de $G = (V, E)$ *induit* par un ensemble de sommets $V' \subset V$ est le graphe dont les sommets sont ceux de $V'$ et dont les arêtes sont celles de $G$ qui relient deux sommets de $V'$ (on ne garde que les sommets de $V'$ et on retire toutes les arêtes qui sortent de $V'$).

Un graphe $G = (V, E)$ est dit *complet* si il y a une arête entre tout couple de sommets de $G$.

*Solution.* **On rappelle que, comme en mathématiques, toutes les variables employées dans une preuve doivent avoir été introduites !** □

## 1 Coloration de graphes

Dans ce problème, les graphes sont non orientés et donnés sous forme de tableaux de listes de voisins.

Par exemple, le graphe de la figure 1 est représenté par `[[2],[2],[0,1,3,4],[2,4,5],[2,3],[3]]`.

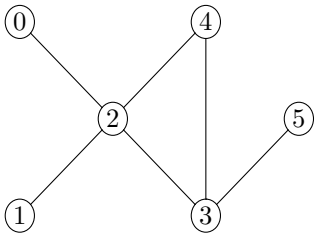



FIGURE 1 – Un graphe non orienté

On appelle *k-coloration* de $G$ une application $c : V \to [\![0, k-1]\!]$ telle que si $(a, b) \in E$ et $a \neq b$ alors $c(a) \neq c(b)$.

Le *cardinal* de la coloration est le nombre de couleurs employées. Deux colorations $c, c'$ sont dites *equivalentes* si il existe une bijection $f$ telle que $c' = f \circ c$.

Le problème de la coloration d'un graphe consiste à trouver une $k$-coloration de $G$ pour laquelle la valeur de $k$ est la moins élevée possible.

Une $k$-coloration est représentée par un tableau `c` de type `list[int]` tel que l'élément d'indice $i$ est la couleur du sommet numéro $i$.

Dans toute la suite, l'annotation de type `sommet` désigne un entier positif, `graph` une liste de listes d'entiers positifs et, `coloration`, une liste d'entiers positifs.

**Q.1** Soit $c$ une $k$-coloration d'un graphe $G = (V, E)$ avec $|V| = n$ (donc une liste de couleurs dans un $[\![0, n-1]\!]$). Montrer qu'il existe une coloration de même cardinal dont les couleurs forment un intervalle d'entiers commençant à 0.

Par exemple `c1=[0,1,2,0,1,3]` et `c2=[0,1,2,0,1,4]` sont deux 4-colorations du graphe 1. Par la suite, on privilégiera les colorations où les couleurs forment un intervalle d'entiers commençant à 0 comme le tableau `c1`.

*Solution.* La question a parfois été mal comprise : on ne demande pas de trouver UNE coloration dont les couleurs forment un intervalle (c'est trivial). Ce qu'on veut c'est, étant donné une coloration $c$, trouver une coloration *équivalente* (c.a.d. qu'existe une fonction $f$ bijective telle que $c' = f \circ c$) dont les couleurs forment un intervalle.

Il faut proposer une renumérotation et vérifier que c'est bien une coloration !

Soit $c$ une coloration et $c_0 < c_1 < \cdots < c_k$ les couleurs utilisées (toutes distinctes 2 à 2). Alors la coloration $c'$ définie par $c'(i)$ est le nombre $r$ tel que $c(i) = c_r$ est une coloration.

En effet, soit $\{a, b\}$ une arête. $c'(a)$ est le nombre $r_a$ tel que $c(a) = c_{r_a}$ et $c'(b)$ est $r_b$ tel que $c(b) = c_{r_b}$. Si $r_a = r_b$ alors $c(a) = c_{r_a} = c_{r_b} = c(b)$, ce qui est absurde puisque $c$ est une coloration. Ainsi, $c'$ est une coloration.

□

**Q.2** Montrer que le graphe 1 possède une 3-coloration et que toute coloration de ce graphe utilise au moins 3 couleurs.

*Solution.* `[0,1,2,0,1,1]` est une 3-coloration.

Comme il y a une clique (c.a.d. un sous-graphe complet) contenant 3 sommets $2, 3, 4$, il faut 3 couleurs au minimum. □

Une telle coloration est appelée *coloration optimale* (elle utilise le minimum de couleurs possibles). Le *nombre chromatique* est le nombre de couleurs utilisées par une coloration optimale.

**Q.3** Soit $n \in \mathbb{N}$. Montrer qu'il existe un graphe $G = (V, E)$ avec $|V| = n$ et un nombre chromatique de $n$.

*Solution.* Il suffit de considérer un graphe complet à $n$ sommets. Considérons une coloration $c$. Elle ne peut pas utiliser plus de couleurs que de sommets ($c$ induit une *surjection* des sommets vers les couleurs utilisées).

Si deux sommets ont la même couleur, on obtient une absurdité puisque, étant voisin, $c$ devrait leur attribuer deux couleurs différentes. Ainsi, $c$ utilise au moins $n$ couleurs.

□

**Q.4** Écrire la fonction `coloration(g:graph,c:coloration)->bool` qui teste si un tableau d'entiers `c` est bien une coloration du graphe `g` (réponse `True`) ou non (réponse `False`).

*Solution*

```
def valide(g,c):
    for i in range(len(g)):
        for v in g[i]:#parcourt des voisins de i uniquement
            if c[i] == c[v]:
                return False
    return True
```

Complexité : on n parcourt tous les sommets une fois ( `for i` ) et toutes les arêtes deux [1] fois ( `for j` ) donc $O(n+p)$.
Si on préfère, on peut aussi dire que la complexité est majorée par un multiple de :

$$\sum_{i=0}^{n-1}(1+\sum_{j \text{voisin de } i} 1)) = \sum_{i=0}^{n-1}(1+\deg^+(i)) = n+p.$$

□

**Q.5** Indiquer brièvement la complexité de votre fonction en fonction de $n$ et $p$.

## 1.1 Graphe biparti

Un graphe qui possède une 2-coloration est dit *biparti*.

**Q.6** Soit $G = (V, E)$ un graphe. Montrer que $G$ est biparti si et seulement si il existe une partition de $V$ en deux ensembles $A$ et $B$ telle que toute arête de $E$ relie un sommet de $A$ à un sommet de $B$.

*Solution.* Si $G$ est biparti, il possède une coloration $c$ à 2 couleurs 0 et 1. On pose $A = c^{-1}(0)$ et $B = c^{-1}(1)$. un arc de $E$ ne peut relier deux sommets de $A$ (resp. $B$), sionon $C$ ne serait pas une coloration.
Réciproquement, si on a une partition de $G$ telle que définie, on pose $c(x) = 0$ si $x \in A$et $c(x) = 1$ sinon. On a bien une coloration. □

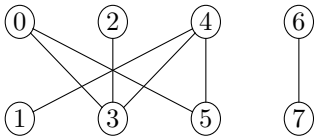



FIGURE 2 – Un graphe biparti

On se donne

```
class NotBiparti(Exception):
    pass
```

C'est une exception à soulever (par `raise NotBiparti` ) lorsque qu'une exploration de graphe réalise que ce graphe n'est pas biparti.

1. car graphe non orienté

**Q.7** Pour déterminer si un graphe est biparti, on initialise un tableau de couleurs à -1 puis on réalise un parcours en profondeur. On examine les voisins du sommet courant $x$ (lequel possède déjà une couleur)

- Si le voisin considéré n'a pas encore de couleur, on lui attribue une couleur différente de celle de $x$ ;
- Si le voisin a déjà une couleur et qu'elle est égale à celle de $x$, le graphe n'est pas biparti. On soulève une exception.

Il ne reste plus qu'à parcourir tous les sommets et à lancer l'exploration depuis chaque sommet de couleur -1.

Écrire la fonction `biparti(g:graph)->coloration` qui implémente cet algorithme.

Donner la complexité de votre fonction.

*Solution.* Complexité : c'est un parcours en profondeur, donc $O(n+p)$.

Attention à bien colorier le sommet initial.

```
def biparti(g):
    c = [-1 for _ in g]
    def dfs(x):
        for v in g[x]:
            if c[v] == -1:
                c[v] = 1-c[x]
                dfs(v)
            if c[x] == c[v]:
                raise NotBiparti
    for i in range(len(g)):
        if c[i] == -1:
            c[i] = 0# ne pas oublier !
            dfs(i)
    return c
```

□

## 1.2 Un algorithme glouton

Les algorithmes connus garantissant une coloration optimale sont de complexité exponentielle. On préfère des heuristiques gloutonnes (faisant le meilleur choix à chaque instant mais sans garanti d'optimalité à la fin) qui, à défaut de garantir une solution minimale, fournissent des colorations acceptables.

L'algorithme glouton consiste à parcourir les sommets par ordre croissant d'index, en attribuant à chaque sommet la plus petite couleur disponible (c'est-à-dire non déjà donnée à un de ses voisins). Au départ, on considère que tous les sommets sont non coloriés (on leur attribue donc la couleur -1).

**Q.8** Écrire la fonction `ppcd(lcv:list[int])->int` qui prend en paramètre une liste de couleurs positives et renvoie la plus petite couleur qui n'est pas dans la liste.

```
ppcd([0,3,2,1]),ppcd([4,2,0,1]),ppcd([3,3,0,0,1,1,2,2]),\
ppcd([0,0,2,2,3,3])
```

```
(4, 3, 4, 1)
```

*Solution.* Voici

```
def ppcd(lcv:list[int])->int:
    if lcv == []:
        return 0
    #les couleurs dans lcv sont positives
    d={k:None for k in lcv}#suppression des doublons : O(len(lcv))
    m = max(lcv)#O(len(lcv))
    for c in range(m+1):#O(min(len(lcv),m))
        if c not in d:
            return c
    return m+1
```

Complexité $O(n)$ si $|lcv| = n$ □

**Q.9** Écrire la fonction `glouton(g:graph)->coloration` qui renvoie une coloration de `g` obtenue selon l'algorithme glouton.

*Solution.* Voici

```
def glouton(g):
    c=[-1 for _ in g]
    for i in range(len(g)):
        lcv = [c[v] for v in g[i] if c[v]>=0]
        c[i]=ppcd(lcv)
    return c
```

□

**Q.10** Faire tourner l'algorithme glouton à la main pour le graphe biparti de la figure 2 en indiquant bien pour chaque étape : le sommet concerné, la couleur de ses voisins et la plus petite couleur disponible.
La coloration obtenue est-elle optimale ?

*Solution.*
```
glouton appliqué à g1 :
il attribue 0 à 0 (aucun voisin marqué)
il attribue 0 à 1 (aucun voisin marqué)
il attribue 0 à 2 (aucun voisin marqué)
il attribue 1 à 3 (2 est marqué 0)
il attribue 2 à 4 (0 marqué 0, 1 marqué 1)
il attribue 1 à 5 (0 marqué 0, 4 marqué 2)
il attribue 0 à 6 (aucun voisin marqué)
il attribue 1 à 7 (6 marqué 0)
```

Non optimale : le graphe étant biparti ,une 2-coloration suffit. □

**Q.11** Montrer que les couleurs employées par la coloration gloutonne forment un intervalle d'entiers.

*Solution.* Supposons que la coloration gloutonne ne forme pas un intervalle de couleurs. Soit $i$ la plus petite couleur non utilisée et $j > i$ la première utilisée après $i$. Soit $v$ un sommet dont la couleur est $j$. Alors toutes les couleurs de 0 à $j-1$ sont utilisées par les voisins de $v$, donc $i$ aussi : ABSURDE □

**Q.12** Montrer que pour un graphe $G$ quelconque il existe toujours un ordonnancement des sommets pour lequel l'algorithme glouton fournit un résultat optimal.

*Indication.* Considérer une coloration optimale $c$ formant un intervalle commençant à 0 et construire l'ordonnancement à partir de $c$. La preuve désirée se fait par récurrence.

*Solution.* On ordonne d'abord les sommets en commençant par la couleur 0, puis 1 etc. On montre que l'algo glouton retourne une coloration $c'$ telle que $c'_k \leq c_k$ pour tout sommet $k$.
On raisonne par récurrence sur le numéro des sommets.
C'est vrai pour le sommet 0 : il est coloré en 0 par $c$ et $c'$.
Supposons que $c'_q \leq c_q$ pour tout $q \leq k$.
Au tour $k+1$ de l'algo glouton, par choix de numérotation des sommets, les voisins déjà colorés du sommet $k+1$ ont des couleurs par $c$ plus petites que $c_{k+1}$ (et même strictement plus petiteq car il s'agit de voisins). Pour tout voisin $v < k+1$ de $k+1$ on a donc $c_v < c_{k+1}$ et par HR on a aussi $c'_v \leq c_v$. Donc la plus petite couleur disponible pour $k+1$ est au plus $c_{k+1}$. Ainsi $c'_{k+1} \leq c_{k+1}$. Hérédité OK.
Puisque les couleurs de $c$ forment un intervalle $I$ commençant à 0, les couleurs de $c'$ forment un sous-ensemble de $I$ (et même un sous-intervalle puisque les couleurs d'une coloration gloutonne forment un intrvalle). Le nombre de couleurs utilisées par $c'$ est donc inférieur au nombre de couleurs de $c$. Or ce dernier est minimal, donc le nombre de couleur utilisé par $c'$ aussi.

□

## 1.3 Graphes d'intervalles

À un ensemble fini d'intervalles non vides $[ai, bi[$, $0 \leq i \leq n-1$ on associe un graphe $G = (V, E)$ pour lequel les sommets de $V$ sont les intervalles et dont les arêtes relient les couples d'intervalles dont l'intersection est non vide. Un tel graphe est appelé *graphe d'intervalle.*

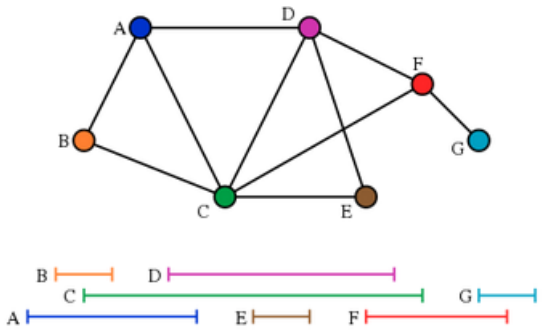



FIGURE 3 – Un graphe d'intervalle (Wikipedia)

**Q.13** On se place dans un graphe d'intervalle dont les sommets sont ordonnés par valeurs de $a_i$ croissantes. Par exemple, l'ordonnancement des sommets du graphe 3 est $A, B, C, D, E, F, G$.

— Montrer que, au moment de la coloration gloutonne du sommet $v_i$ d'un graphe d'intervalle ainsi ordonné, le graphe induit par $v_i$ et ses voisins déjà colorés est complet.

— En déduire que l'algorithme glouton fournit une coloration optimale.

*Solution.* Quand l'algo glouton colore $v_i$, tous les intervalles de borne inf inférieure strictement à $a_i$ sont colorés. Et les voisins de $v_i$ déjà colorés ont une borne inf inférieure ou égale à $a_i$.
Si le sommet/intervalle $v_i$ reçoit la couleur gloutonne $k$ : alors les couleurs 0 à $k-1$ sont prises par ses voisins déjà colorés. Puisque les intervalles sont rangés par ordre croissant de borne inf, cela veut dire que $a_i$ est contenue dans chaque intervalle voisin de $v_i$ (ie ayant une intersection avec $v_i$). Mézalors tout couple de voisin de $v_i$ a une intersection donc il y a une arête entre les deux membres du couple. Ainsi le graphe induit par $v_i$ et ses voisins déjà colorés est complet.
Considérons $v$ de couleur maximale $k$ par l'algorithme glouton. On vient de montrer que $v$ et ses voisins forment un graphe complet à $k+1$ sommets. Une coloration quelconque utilise donc au moins $k+1$ couleurs donc aussi une coloration optimale.
Ainsi la coloration gloutonne est optimale puisqu'elle n'utilise pas plus de couleurs qu'une coloration optimale. □

## 1.4 Algorithme DSATUR

Cet algorithme est une variante de l'algorithme glouton, dans lequel on essaie de choisir « au mieux » le prochain sommet à devoir être coloré. Pour cela on introduit la notion de *degré de saturation* : en cours d'exécution, et pour tout $v \in S, d_s(v)$ est le nombre de couleurs distinctes d'ores et déjà utilisées pour colorer les voisins de $v$.

À chaque étape, l'algorithme DSATUR attribue la plus petite couleur disponible au sommet non encore coloré de degré de saturation maximal. En cas d'égalité de degrés de saturation, c'est le sommet de degré maximal qui est choisi. En cas d'égalité, on prend le sommet de numéro maximal.

Au départ, on considère que tous les sommets sont non coloriés (on leur attribue donc la couleur -1).

**Q.14** Écrire une fonction `degsat(g:graph,c:coloration,s:sommet)->int` qui prend en arguments un graphe $g$ en cours de coloriage, un tableau `c` de coloration incomplet (certains sommets sont de couleur $-1$) et un sommet $s$ et qui retourne le degré de saturation de $s$.

```
g1 = [[0,3,5],[4],[3],[0,2,4],[0,1,3,5],[0,4],[7],[6]]
c = [2,-1,0,2,-1,4,0,0]
v = 4
print(degsat(g1,c,v))
```

```
2
```

*Solution.* Voici

```
def degsat(g,c,s):
    #renvoie le nombre de voisins de v déjà colorés
    d = {c[v]:None for v in g[s] if c[v]>=0}
    return len(d)
```

□

**Q.15** Rédiger une fonction `satmax(g:graph,c:coloration,ac:list[sommet])->sommet` qui prend en arguments un graphe en cours de coloriage, un tableau de coloration `c` (des couleurs déjà attribuées) et une liste des sommets à colorier.

La fonction retourne le sommet non encore colorié de saturation maximale (et en cas d'égalité, celui de degré maximal). En outre, elle met à jour la liste des sommets à colorier en supprimant le sommet qui vient d'âtre coloré.

```
g2 = [[1,6,7,2],[0,6,7,3],[0,5,4],[1,5,4],[2,3,5],[2,3,4],[0,1],[0,1]]
c = [ 0, -1, 1, -1, -1, -1, 2, 2]
ac = [1, 3, 4, 5] # sommets à colorier
print("ac avant appel : {}".format(ac))
print("sommet élu : {}".format(satmax(g2,c,ac)))
print("ac après appel : {}".format(ac))
```

```
ac avant appel : [1, 3, 4, 5]
sommet élu : 1
ac après appel : [3, 4, 5]
```

*Solution.* Voici 2 versions.

```
def satmax(g,c,ac):
    d = {s:(degsat(g,c,s),len(g[s]),s) for s in ac}
    ac.sort(key=lambda s:d[s])# O(n log n)
    x = ac.pop()#effet de bord en O(1)
    return x

def satmax(g,c,ac):
    d = {s:(degsat(g,c,s),len(g[s]),s) for s in ac}#O(n)
    best = max([d[s] for s in d])#O(n)
    #print(ac,d)
    ac.remove(best[2])#effets de bords en O(n)
    #print()
    return best[2]
```

La seconde est pas mal, mais elle impose de se souvenir de la méthode `remove`. Rappelons que `pop` attend un indice pas une valeur. □

**Q.16** Rédiger une fonction `dsatur(g:graph)->coloration` qui prend en argument un graphe et retourne la coloration de ce dernier obtenue par application de l'algorithme DSATUR.

```
g = [[2],[2],[0,1,3,4],[2,4,5],[2,3],[3]]
dsatur(g)
```

```
[1, 1, 0, 1, 2, 0]
```

*Solution.* Voici

```
def dsatur(g):
    c=[-1 for _ in g]
    ac = [i for i in range(len(g))]
    while ac!=[]:
        s = satmax(g,c,ac) #ac est modifiée
        lcv = [c[v] for v in g[s] if c[v]>=0]
        c[s]=ppcd(lcv)
    return c
```

□

**Q.17** Faire tourner l'algorithme DSATUR à la main pour le graphe biparti de la figure 2 en indiquant bien pour chaque étape : le sommet choisi par `satmax`, la couleur de ses voisins et la plus petite couleur disponible.
La coloration obtenue est-elle optimale ?

*Solution.* Voici

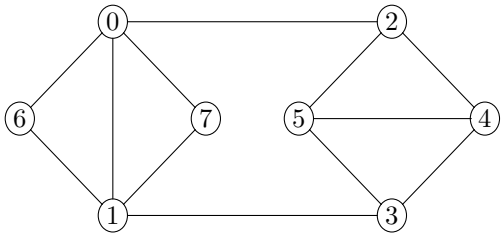
```
gbi = [[3,5],[4],[3],[0,2,4],[1,3,5],[0,4],[7],[6]]
dsatur(gbi) # version verbeuse, avec des "print"
```

```
sommet élu : 4
couleur des voisins : []
couleur attribuée : 0
sommet élu : 3
couleur des voisins : [0]
couleur attribuée : 1
sommet élu : 5
couleur des voisins : [0]
couleur attribuée : 1
sommet élu : 0
couleur des voisins : [1, 1]
couleur attribuée : 0
sommet élu : 2
couleur des voisins : [1]
couleur attribuée : 0
sommet élu : 1
couleur des voisins : [0]
couleur attribuée : 1
sommet élu : 7
couleur des voisins : []
couleur attribuée : 0
sommet élu : 6
couleur des voisins : [0]
couleur attribuée : 1

[0, 1, 0, 1, 0, 1, 1, 0]
```

Coloration optimale : 2 couleurs pour un graphe biparti.
En fait, on peut montrer que DSATUR fournit toujours une coloration optimale pour les graphes bipartis. □

**Q.18** Même question avec le graphe ci-dessous.



La coloration obtenue est-elle optimale ?

*Solution.* Déroulement dans l'ordre où les sommets sont colorés :

```
g18 = [[1 ,2 ,6 ,7] ,[0 ,3 ,6 ,7] ,[0 ,4 ,5] ,[1 ,4 ,5] ,[2 ,3 ,5] ,[2 ,3 ,4] ,[0 ,1] ,[0 ,1]]

sommet élu : 1
couleur des voisins : []
couleur attribuée : 0
sommet élu : 0
couleur des voisins : [0]
couleur attribuée : 1
sommet élu : 7
couleur des voisins : [1, 0]
couleur attribuée : 2
sommet élu : 6
couleur des voisins : [1, 0]
couleur attribuée : 2
sommet élu : 3
couleur des voisins : [0]
couleur attribuée : 1
sommet élu : 5
couleur des voisins : [1]
couleur attribuée : 0
sommet élu : 4
couleur des voisins : [1, 0]
couleur attribuée : 2
sommet élu : 2
couleur des voisins : [1, 2, 0]
couleur attribuée : 3

[1, 0, 3, 1, 2, 0, 2, 2]
```

Or, une coloration en 3 couleurs est possible. Donc, non : la coloration obtenue n'est pas optimale.

□

# 2 Recherche de plus courts chemins entre tout couple de sommets

Un triplet $G = (V, E, p)$ est appelé un graphe *pondéré* si le couple $(V, E)$ est un graphe et si $w$ est une fonction de $E$ dans $\mathbb{R}$.

— On dit que $w$ est la (fonction de) *pondération* ($w$ pour *weight*).

— Si $e \in E$, alors $w(e)$ est appelé *poids de* $e$ ou *pondération de* $e$.

— Les graphes sont munis par défaut de la fonction de pondération $w : E \to \mathbb{R},\ e \mapsto 1$. Donc tout graphe est un graphe pondéré qui s'ignore.

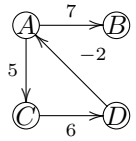
Le *poids* d'un chemin dans un graphe pondéré est la somme des poids des arcs qui le constituent. Si $c$ est le chemin chaîne $x_1 \dots x_n$ :

$$w(c) = \sum_{i=1}^{n-1} w(x_i \to x_{i+1})$$

Le *poids* d'un cycle se définit de la même façon.

Le *plus court chemin* (PCC) entre deux sommets est le chemin de poids minimum parmi tous les chemins entre les deux sommets. On rappelle que, par convention, $\min(\emptyset) = +\infty$. La *distance* $d(i,j)$ entre deux sommets $i$ et $j$ est la longueur d'un PCC entre $i$ et $j$.

**Exemple 1.**




$w(A,C,D,A) = -2+6+5 = 9$
$w(A,D,C) = -2+6 = 4$ et $w(A,C) = 5$
Le plus court chemin de $A$ à $C$ est $A, D, C$

## 2.1 Généralités

**Q.19** Montrer que si le graphe $G = (V, E, w)$ possède un cycle de poids négatif, alors, pour tout $y$ sur ce cycle, et pour tout sommet $x \in V$, il n'existe pas de PCC entre $x$ et $y$.

*Solution.* Soit $C$ un cycle, $y$ sur ce cycle et $p < 0$ le poids du cyche.
Soit $x$ un sommet.
Si il existe un PCC $x \rightsquigarrow y$ entre $x$ et $y$, alors soit $p'$ son poids. En empruntant le chemin de $x$ à $y$ puis le cycle partant de $y$ et revenant à $y$, on a un chemin de $x$ à $y$ de poids $p' + p < p'$. Absurde.

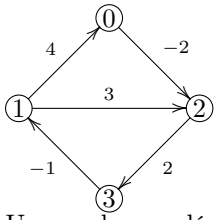
□

Ainsi, la recherche de plus court chemin n'a de sens que pour les graphes dépourvus de cycle de poids négatif.

Dans toute la suite de ce problème, les graphes considérés sont orientés.

Un graphe pondéré $G = (V = [\![0, n-1]\!], E, w)$ est représenté par sa *matrice de pondérations* $W$ telle que $w_{i,j} = w(i,j)$. Si l'arc $i \to j$ n'existe pas, alors $w_{i,j} = +\infty$.

**Exemple 2.**




Un graphe pondéré orienté.

$$\begin{pmatrix} 0 & \infty & -2 & \infty \\ 4 & 0 & 3 & \infty \\ \infty & \infty & 0 & 2 \\ \infty & -1 & \infty & 0 \end{pmatrix}$$

Sa matrice de pondérations.

*Remarque.* On met les coefficients diagonaux à 0 ce qui peut se discuter. Nous avons supposé que les graphes ne contiennent pas de boucle (arcs $x \to x$). Mettre les coefficients diagonaux à 0 revient implicitement à supposer la présence d'une boucle de poids nul sur chaque sommet.

Avec cette convention, la matrice des distances obtenue possède une diagonale nulle. C'est moral : dans un graphe sans cycle de poids négatif, la distance d'un sommet à lui même est nulle.

**Q.20** Montrer que, dans un graphe pondéré, si un PCC de $A$ à $C$ passe par $B$, alors le sous-chemin entre $A$ et $B$ est aussi un chemin de poids minimum.

*Solution.* En effet, s'il existe un autre chemin plus court entre $A$ et $B$, il suffit de le mettre à la place du premier pour obtenir un nouveau chemin de $A$ à $C$ encore plus court.
Ceci contredit le fait que le premier chemin avait un poids minimum. □

Ainsi, les PCC vérifient le *principe d'optimalité de Bellman*. Il est donc naturel d'appliquer un algorithme de programmation dynamique à la recherche de PCC dans un graphe pondéré.

## 2.2 Algorithme de Floyd-Warshall

L'*algorithme de Floyd-Warshall* (parfois appelé algorithme de Roy-Floyd-Warshall car décrit par Bernard Roy en 1959) est un algorithme pour déterminer les distances des plus courts chemins entre toutes les paires de sommets dans un graphe orienté et pondéré, en temps cubique en le nombre de sommets. Il est basé sur le principe de *programmation dynamique* et prend en entrée une matrice de pondération d'un graphe sans cycle de poids négatif.

Dans un chemin $x_0, x_1, \ldots, x_m$, on appelle *sommets intermédiaires* les sommets $x_1, \ldots, x_{m-1}$. Dans un chemin $x_1, x_2$, l'ensemble des sommets intermédiaires est vide.

On note dans ce qui suit $w_{i,j}^k$ le poids minimal d'un chemin du sommet $i$ au sommet $j$ n'empruntant que des sommets *intermédiaires* dans $\{0, 1, 2, \ldots, k-1\} \setminus \{i, j\}$ s'il en existe un, et $+\infty$ sinon. La matrice des $w_{i,j}^k$ est notée $W^k$.

Le listing 1 donne l'algorithme de Floyd-Warshall. La fonction renvoie la matrice des distances entre tout couple de sommets du graphe.

Listing 1 – Algorithme de Floyd-Warshall

```
fonction fw(W):
    entree
      W : matrice de pondération d'un graphe à n sommets
    sortie
      W^n matrice des distances minimales
    debut
      W^0 ← W
      pour k = 1 à n faire
        pour tous les couple (i,j) ∈ [[0, n-1]]^2 faire
          w^k_{i,j} ← min(w^{k-1}_{i,j}, w^{k-1}_{i,k-1} + w^{k-1}_{k-1,j})
     renvoyer W^n
    fin
```

**Q.21** Faire tourner l'algorithme de Floyd-Warshall sur le graphe de l'exemple 2.
On indiquera les valeurs successives de la matrice $W^k$.

*Solution.*
$$\begin{pmatrix} 0 & \infty & -2 & \infty \\ 4 & 0 & 3 & \infty \\ \infty & \infty & 0 & 2 \\ \infty & -1 & \infty & 0 \end{pmatrix}$$
$$\begin{pmatrix} 0 & \infty & -2 & \infty \\ 4 & 0 & \min(\mathbf{3}, \mathbf{4}-\mathbf{2}) & \infty \\ \infty & \infty & 0 & 2 \\ \infty & -1 & \infty & 0 \end{pmatrix}$$
$$\begin{pmatrix} 0 & \infty & -2 & \infty \\ 4 & 0 & 2 & \infty \\ \infty & \infty & 0 & 2 \\ \min(+\infty, -\mathbf{1}+\mathbf{4}) & -1 & \min(+\infty, -\mathbf{1}+\mathbf{2}) & 0 \end{pmatrix}$$
$$\begin{pmatrix} 0 & \infty & -2 & \min(\infty, -\mathbf{2}+\mathbf{2}) \\ 4 & 0 & 2 & \min(\infty, \mathbf{2}+\mathbf{2}) \\ \infty & \infty & 0 & 2 \\ 3 & -1 & 1 & 0 \end{pmatrix}$$
$$\begin{pmatrix} 0 & \min(\infty, \mathbf{0}-\mathbf{1}) & -2 & 0 \\ 4 & 0 & 2 & 4 \\ \min(\infty, \mathbf{3}+\mathbf{2}) & \min(\infty, -\mathbf{1}+\mathbf{2}) & 0 & 2 \\ 3 & -1 & 1 & 0 \end{pmatrix}$$
□

**Q.22** Montrer par récurrence sur $k \leq n$ la propriété $P(k)$ : « Pour tout couple de sommets $(i, j)$, $w^k_{i,j}$ indique la longueur d'un plus court chemin entre $i$ et $j$ n'empruntant que des sommets intermédiaires dans $0, \ldots, k-1$ ».
En déduire que l'algorithme renvoie bien la matrice des distances.

*Solution.* La propriété est évidemment vraie pour $k = 0$ puisque l'ensemble des sommets intermédiaires entre deux sommets $i$ et $j$ est vide. $w^0_{i,j} = w_{i,j}$ indique bien le plus court chemin de $i$ à $j$ sans sommet intermédiaire.
Supposons $P(k-1)$ et montrons $P(k)$.
Considérons un plus court chemin $C$ entre deux sommets $i$ et $j$ n'empruntant que des sommets intermédiaires dans $0, \ldots, k-1$. Comme il n'y a pas de cycle de poids négatif, aucun sommet du chemin n'est emprunté deux fois.
Constatons que $w(C) \leq w^{k-1}_{i,j}$ puisque, par HR, $w^{k-1}_{i,j}$ est la longueur d'un PCC ne passant que par $0, \ldots, k-2$ et qu'on s'autorise un sommet supplémentaire dans $C$.

— Soit $C$ ne passe pas par $k-1$. Il ne passe que par des sommets intermédiaires dans $0,\dots,k-2$. Alors, par HR, $w(C) = w_{i,j}^{k-1}$.
Il n'est donc pas plus court de passer par $k-1$ pour rejoindre $j$ depuis $i$ que de passer par $C$. Constatons que, par HR, $w_{i,k-1}^{k-1} + w_{k-1,j}^{k-1}$ renseigne le poids d'un chemin $C'$ qui emprunte un plus court chemin de $i$ à $k-1$ avec sommets intermédiaires dans $0\dots k-2$ puis un plus court chemin de $k-1$ à $j$.
Ainsi $w(C') \geq w(C)$. Il vient que $w_{i,k-1}^{k-1} + w_{k-1,j}^{k-1} \geq w_{i,j}^{k-1}$. Et donc $w_{i,j}^{k} = w_{i,j}^{k-1}$ renseigne bien la longueur d'un PCC de $i$ à $j$ ne passant que par les sommets intermédiaires $0,\dots,k-1$

— Soit $C$ passe par $k-1$. Par principe d'optimalité, les sous-chemins de $C$ de $i$ à $k-1$ puis de $k-1$ à $j$ sont des PCC. Par HR, leurs poids sont $w_{i,k-1}^{k-1}$ et $w_{k-1,j}^{k-1}$. Le poids de $C$ est donc $w_{i,k-1}^{k-1} + w_{k-1,j}^{k-1}$. C'est justement la valeur de $w_{i,j}^{k}$.

L'hérédité est donc prouvée.
Lorsque $k = n$, $w_{i,j}^{n}$ renseigne la longueur d'un plus court chemin de $i$ à $j$ ne passant que par des sommets intermédiaires entre 0 et $n-1$. Comme il n'y a pas d'autres sommets dans le graphe, $d(i,j) = w_{i,j}^{n}$ □

Pour représenter l'infini, le plus simple est d'utiliser un type qui permet d'exprimer cette notion. Les flottants de `numpy` sont ce qu'il nous faut. Nous utiliserons des matrices `numpy` pour implémenter les matrices de pondérations. Par exemple, la matrice suivante implémente la matrice de pondérations du graphe 2 :

```
import numpy as np
W = np.matrix([[0.,np.inf,-2.,np.inf],\
               [4.,0.,3.,np.inf],\
               [np.inf,np.inf,0.,2],\
               [np.inf,-1.,np.inf,0.]])
```

On peut utiliser `inf` plutôt que `np.inf` pour alléger l'écriture dans les codes qui suivent.

**Q.23** Écrire la procédure `fw(W)` qui prend en paramètre une matrice de pondérations et la transforme en matrice des distances.
Il n'y a donc aucune valeur retournée mais la matrice de pondérations est modifiée.

**Q.24** Donner, en fonction du nombre de sommets $n$ du graphe, la complexité de votre fonction.

*Solution.* Voici

```
def fw(W):
    n=len(W)
    for k in range(n):
        for i in range(n):
            for j in range(n):
                W[i,j]=min(W[i,j],W[i,k]+W[k,j])
```

Complexité en $O(n^3)$

□

**Q.25** Une modification minime de la définition de la matrice de pondérations, permet à l'algorithme de Floyd-Warshall de détecter la présence de cycles dans un graphe pondéré.
Que faut-il faire ?

*Solution.* On initialise la diagonale de la matrice à l'infini. A la fin on scrute la diagonale pour voir si une valeur est finie : c'est l'indication de la présence d'un cycle. □

*Remarque.* L'algorithme étudié ne donne que la matrice des distances. Pour construire un plus court chemin entre tout couple de sommets, on peut gérer en plus une *matrice des prédecesseurs* $P$ de taille $n \times n$. Le coefficient $p^k_{i,j}$ indique le dernier sommet avant $j$ dans un plus court chemin de $i$ à $j$ n'empruntant que des sommets dans $0, \dots, k-1$.

Initialisé à -1 au début de l'algorithme, le coeficient $p^k_{i,j}$ prend la valeur $k-1$ chaque fois que

$$w^{k-1}_{i,k-1} + w^{k-1}_{k-1,j} = \min(w^{k-1}_{i,j}, w^{k-1}_{i,k-1} + w^{k-1}_{k-1,j}).$$

# Appendice

**Tri** Pour trier une liste de nombres `t` par ordre croissant, il est toujours possible d'utiliser `t.sort()`. La complexité est alors en $O(|t| \log |t|)$. On peut aussi utiliser le paramètre `key`. Par exemple, `t.sort(key=lambda x : 1/x)` trie la liste `t` par ordre croissant d'inverse des valeurs qu'il contient.

**Matrices** Voici comment on utilise une matrice `numpy` :

```
1  import numpy as np
2  A = np.matrix([[1.,2.],[3.,4.]])
3  A[1,1] = np.inf # infini des flottants
4  A + 0.5 * A
```

```
matrix([[ 1.5,  3.],
        [ 4.5, inf]])
```

Par ailleurs, `np.inf > 3.` renvoie `True`.