NE PAS IMPRIMER CE DOCUMENT

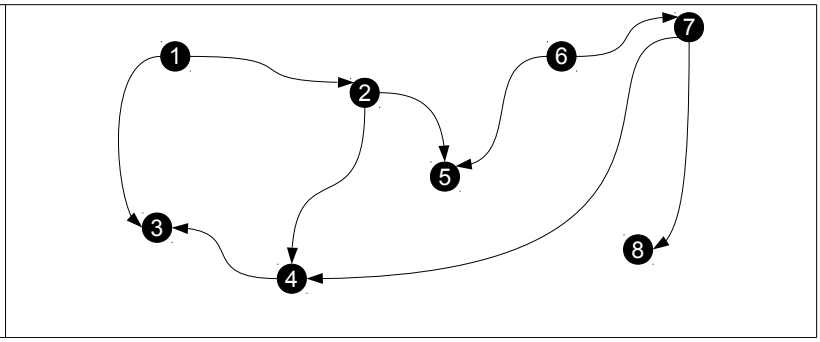
il va être modifié et complété

Document de travail pour le TP6
-plus courts chemins dans les graphes-

Créez une classe permettant l'initialisation et l'affichage d'un graphe de taille réduite (une dizaine de sommets). Vous devez disposer dans cette classe d'au moins quatre méthodes d'initialisation :

- **initialisation aléatoire** : des zéros partout avec tirage aléatoire d'un nombre d'arcs et pour chaque arc de sa source et de sa cible. Une des méthodes permettra d'initialiser ainsi un **graphe valué** avec des valuations positives, en tirant au hasard la valuation des arcs. Une autre méthode permettra d'initialiser un **graphe non valué** (tous les arcs ont une valuation de 1) ;
- initialisation du **graphe de l'exemple 2 en version valuée et non valuée**. Le graphe et la liste de ces arcs à initialiser sont donnés ci-dessous ;
- affichage simple de la matrice reprépsentative du graphe (les valeurs non significatives seront affichées sous forme d'un tiret).

```
s=1; c=2; t[s][c]=1;
s=1; c=3; t[s][c]=1;
s=2; c=4; t[s][c]=1;
s=2; c=5; t[s][c]=1;
s=4; c=3; t[s][c]=1;
s=6; c=5; t[s][c]=1;
s=6; c=7; t[s][c]=1;
s=7; c=8; t[s][c]=1;
s=7; c=4; t[s][c]=1;
```



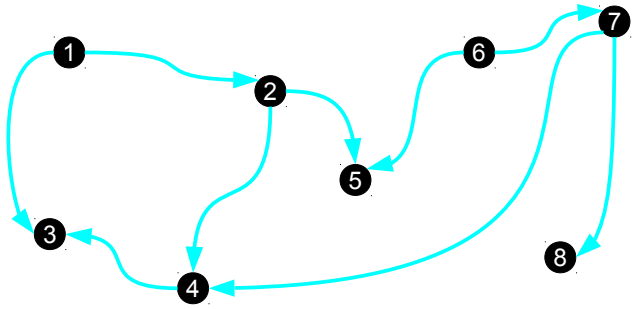
1) Algorithme de Dantzig

Ajoutez à votre classe les méthodes :

- définition d'une constante infinie ;
- création de la matrice M à partir de la matrice du graphe (remplacement des 0 par la valeur infinie sauf sur la diagonale) ;
- initialisations des matrices des distances(*D*) et des prédécesseurs (*P*) ;
- réalisation d'une étape permettant de passer de l'étape k à l'étape *k+1* ;
- algorithme de Dantzig.

Vous devez tester votre programme sur :

- la version non valuée de l'exemple 2. Sur cet exemple, les chemins existants ont été listés en TD. Vérifiez qu'en cas d'existence de plusieurs chemins, c'est bien le PCC qui apparaît dans la matrice *D.* Vérifiez aussi la matrice des prédécesseurs qui doit avoir la forme indiquée ci-dessous ;
- le même graphe en version valuée (avec des valuations positives choisies de façon aléatoire) ;
- un graphe construit de façon aléatoire (tirage au hasard du nombre d'arcs ainsi que des source, cible et valuation de chaque arc).



pcc de longueur 2
1-2-5
1-2-4
2-4-3
6-7-8
6-7-4
7-4-3

pcc de longueur 3
6-7-4-3

D

0	1	1	2	2	-	-	-
-	0	2	1	1	-	-	-
-	-	0	-	-	-	-	-
-	-	1	0	-	-	-	-
-	-	-	-	0	-	-	-
-	-	3	2	1	0	1	2
-	-	2	1	-	-	0	1
-	-	-	-	-	-	-	0

P

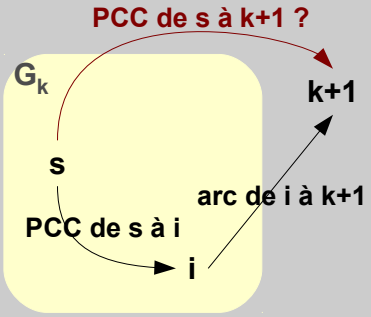
1	1	1	2	2	0	0	0
0	2	4	2	2	0	0	0
0	0	3	0	0	0	0	0
0	0	4	4	0	0	0	0
0	0	0	0	5	0	0	0
0	0	4	7	6	6	6	7
0	0	4	7	0	0	7	7
0	0	0	0	0	0	0	8

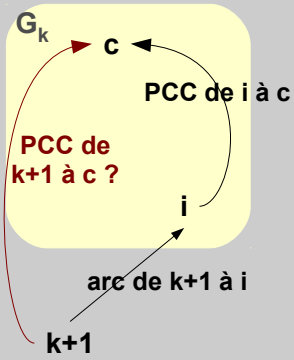
Les algorithmes correspondants sont donnés ci-dessous.

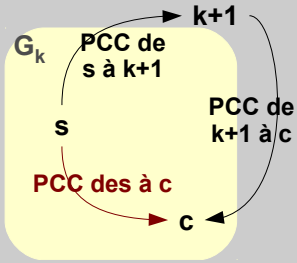
```
creerM ( M , adj , n )  :
// M est la matrice utilisée par Dantzig
// adj est la matrice initiale du graphe
// n est le nombre de sommets
        pour s de 1  à n
        faire pour c de 1  à n
              faire si s==c
                    alors M[s][c] ← 0
                    sinon si adj[s][c] == 0
                          alors M[s][c] ← ∞
                          sinon M[s][c] ← adj[s][c]
                          fsi
                    fsi
              fpour
        fpour
```

```
initialiser ( D , P , n )  :
// D est la matrice des distances
// P est la matrice des prédécesseurs
// n est le nombre de sommets
    pour s de 1  à n
    faire pour c de 1 à n
          faire si s==c
                alors D[s][c] ← 0
                      P[s][c] ← c
                sinon D[s][c] ← ∞
                      P[s][c] ← 0
                fsi
          fpour
     fpour
```

```
etape ( k, M , D , P , n )  :
// k est l'étape (introduction du sommet k+1)
// M est la matrice décrivant le graphe
// D est la matrice des distances
// P est la matrice des prédécesseurs
// n est le nombre de sommets

  // k+1 eme colonne : pour chaque source s, on minimise D[s][i]+M[i][k+1]
  pour s de 1 à k
  faire min ← ∞
        indMin ← 1
        pour i de 1 à k
        faire d ← D[s][i]+M[i][k+1]
              si d < min
              alors min ← d
                    indMin ← i
              fsi
        fpour
        D[s][k+1] ← min
        si min <  ∞
        alors P[s][k+1] ← indMin
        fsi
   fpour

  // k+1 eme ligne : pour chaque cible c, on minimise M[k+1][i]+D[i][c]
  pour c de 1 à k
  faire min ← ∞
        indMin ← 1
        pour i de 1  à k
        faire d ← M[k+1][i]+D[i][c]
              si d < min
             alors min ← d
                    indMin ← i
              fsi
        fpour
        D[k+1][c] ← min
        si min < ∞
        alors si indMin==c
              alors P[k+1][c] ← k+1
              sinon P[k+1][c] ← indMin
              fsi
        fsi
  fpour

  // intérieur de la matrice
  pour s de 1  à k
  faire pour c de 1 à k
        faire d ← D[s][k+1]+D[k+1][c]
              si d < D[s][c]
              alors D[s][c] ← d
                    P[s][c] ← P[k+1][c]
              fsi
        fpour
  fpour
```








```
dantzig ( M , D , P , n ) :
// M est la matrice décrivant le graphe
// D est la matrice des distances
// P est la matrice des prédécesseurs
// n est le nombre de sommets
   initialiser ( D, P, n )
   pour k de 1 à n-1
   faire etape (  k,  M, D, P, n )
   fpour
```

2) Algorithme de Floyd

Reprenez les méthodes définies pour l'algorithme de Dantzig (création de la matrice F_0 à partir de celle du graphe, affichages) et ajoutez à votre classe les méthodes :

- réalisation d'une étape permettant d'introduire le sommet *k* ;
- algorithme de Floyd.

Vous obtenez avec l'algorithme de Floyd, la matrice des distances des PCC. Vous devez tester votre programme sur les mêmes graphes que pour Dantzig … et obtenir pour le graphe de l'exemple 2 les mêmes résultats.

Les algorithmes correspondants sont donnés ci-dessous (où `créer(adj,M,n)` a été donné pour l'algorithme de Dantzig).

```
etapeF ( k, AV , n ) :
      // on suppose la matrice des PCC calculée jusqu'à k-1 : dans AV
      // on calcule pour k : dans la matrice AP, renvoyée comme résultat
      // n est le nombre de sommets

       pour s de 1 à n
       faire pour c de 1 à n
             faire // le passage par l'intermédiaire de k
                   // est-il meilleur que le chemin déjà calculé
                   // en utilisant les sommets 1 à k-1 ?
                   tempo ← AV[s][k] + AV[k][c]
                   si AV[s][c] <= tempo
                   alors AP[s][c] ← AV[s][c]
                   sinon AP[s][c] ← tempo
                   fsi
              fpour
        fpour
      resultat AP
```

Dans l'algorithme qui évalue séquentiellement les matrices pour k variant de 2 à n, deux matrices sont utilisées en boucle.

```
floyd ( adj , n ) :
      // adj est la matrice d'adjacence du graphe
      // n le nombre de sommets
      // le résultat est la matrice des distances sur les PCC


      creer( adj, A , n )
      resultatB ← vrai  // pour utiliser en boucle les matrices A et B
      pour k de 2 à n
      faire si resultatB
              alors etapeF (  k,  AV, AP, n )
              sinon etapeF (  k,  AP, AV, n )
              fsi
              resultatB ← NOT (resultatB)
      fpour
      si resultatB
      alors le resultat à renvoyer est B
      sinon le resultat à renvoyer est A
      fsi
```

Si vous avez un peu de temps, vous pouvez modifier votre méthode de génération aléatoire de graphe pour ajouter des arcs de valuation négative : il faut alors **détecter les cycles absorbants** (de valuation globale négative) en vérifiant à chaque étape que *F(i,i)* n'est pas négatif **et interrompre le**

calcul si un cycle absorbant est trouvé. Vous pouvez utiliser le mécanisme des exceptions de Java :

- définissez une classe exception qui aura pour seul champ le numéro du sommet *i* sur lequel vous avez un cycle absorbant ;

```
public class cycleAbsorbant extends Exception
{  int sommet;
   ...
}
```

- vous devez indiquer que votre méthode de calcul d'une étape est susceptible de générer une telle exception :

```
public  profilDeLaMethode throws cycleAbsorbant
```

  et activer cette exception lorsqu'un cycle absorbant est détecté :

```
if ( F(s,s) < 0)          // cycle négatif
   { throw new cycleAbsorbant (s);  // DONC exception
   }
```

- lorsque vous appelez la méthode de calcul d'une étape, indiquez ce qui doit être fait en cas d'exception (en l'occurrence, interrompre le programme) *:*

```
try { ...
    }
  catch ( cycleAbsorbant e)
        { afficher le cycle absorbant et interrompre le programme
        }
```

3) Algorithme de Ford-Bellman

Reprenez sur les classes précédentes :

- définition d'une constante infinie ;
- création de la matrice M à partir de la matrice du graphe (remplacement des 0 par la valeur infinie sauf sur la diagonale) ;
- affichage de la matrices du graphe.

Ajoutez à cette classe les méthodes :

- l'affichage des tableaux des distances(*d*) et des prédécesseurs (*p*) ;
- algorithme de Ford-Bellman.

Vous devez tester votre programme sur :

- la version non valuée de l'exemple 2 ;
- le même graphe en version valuée (avec des valuations positives choisies de façon aléatoire) ;
- un graphe construit de façon aléatoire (tirage au hasard du nombre d'arcs ainsi que des source, cible et valuation de chaque arc).

Les algorithmes correspondants sont donnés ci-dessous (où `créer(adj,M,n)` a été donné pour l'algorithme de Dantzig).

```
ford ( adj , d , p,  n ) :
      // adj est la matrice d'adjacence du graphe
      //  d est le tableau des distances sur les PCC partant de 1
      // d est le tableau des prédécesseurs sur les PCC partant de 1
      //  n le nombre de sommets

     creer ( adj , M , n )
     initialiser ( p, d, n )
     répéter
             modif=faux
             pour s de 1  à  n
             faire pour c de 1  à n
                   faire si 0 < M[s][c]) < ∞
                       alors // il existe un arc entre s et c
                             // on teste si le fait de passer par cet arc
                              // diminue la distance entre 1 et c
                                 tempo=d[s]+ M[s][c]
                                 si d[c] > tempo
                                 alors d[c] ← tempo
                                           p[c] ← s
                                           modif ← vrai
                                 fsi
                          fsi
                    fpour
              fpour
      jusqu'à modif=faux
```

```
initialiser ( p, d, n ):
      // d est le tableau des distances sur les PCC partant de 1
      // d est le tableau des prédécesseurs sur les PCC partant de 1
      // n le nombre de sommets

      d[1]=0
      p[1]=1
      pour i de 2 à n
      faire d[k]← ∞
            p[k]←0
      fpour
```