Dossier Optimisation et Recherche Opérationnelle

PERENON Clément
REGAZZETTI Léa

Master 1 Informatique
Promotion 2020-2021

Table des matières

1. Introduction

Ce dossier s'intègre dans le cadre du contrôle continu des connaissances de nos études en première année de master informatique, et plus particulièrement lors de notre cours d'optimisation et recherche opérationnelle dans lequel nous avons étudié la théorie des graphes.

Dans ce dossier, nous présenterons différents algorithmes traitant des graphes ainsi que leur implémentation dans le langage de script R.

Nous nous sommes intéressés à 3 algorithmes. Tout d'abord, l'algorithme de Prim, qui permet de déterminer un arbre partiel de poids minimal. Ensuite, l'algorithme de Ford-Bellman pour le calcul des plus courts chemins. Et enfin, l'algorithme de Ford-Fulkerson pour la détermination du flot de valeur maximale dans un réseau avec capacités. Pour chacun de ces algorithmes, nous détaillerons par la suite le problème que tente de résoudre l'algorithme, puis nous en rappellerons son pseudo-code. Ensuite, nous donnerons notre implémentation dans le langage R et enfin nous vous illustrerons son fonctionnement par le biais de plusieurs exemples.

2. Algorithme 1 : Détection de l'arbre recouvrant de poids minimal par Prim

2.1. Objet de l'algorithme

Étant donné un graphe G = [X,U] non orienté, connexe et valué, représenté par sa matrice d'adjacence pondérée A. Le problème est de déterminer un arbre partiel de G de poids minimal.

La première étape consiste à choisir un sommet de X au hasard que l'on ajoute dans X' la liste des sommets de l'arbre partiel. Ensuite, à chaque nouvelle étape, on choisit une arête (j,k) de poids minimal ayant un sommet j dans X' et un sommet k n'appartenant pas à X'. On ajoute cette arête (j,k) à l'arbre partiel U' et le sommet k à la liste X' des sommets de l'arbre partiel. L'algorithme se finit lorsque tous les sommets X du graphe sont contenus dans la liste X' des sommets de l'arbre partiel U'.

La construction d'un arbre partiel de poids minimal dans un graphe intervient dans de nombreux contextes d'optimisation, notamment dans l'étude et la réalisation de réseaux (électrique, internet, etc…).

2.2. Pseudo-code de l'algorithme

Nous rappelons ci-dessous le pseudo-code de l'algorithme de Prim pour la détermination de l'arbre recouvrant de poids minimal.

Input : $G = [X, U]$
$X' \leftarrow \{i\}$ où i est un sommet de X pris au hasard
$U' \leftarrow \emptyset$
Tant que $X' \neq X$ **faire**
    Choisir une arête (j, k) de poids minimal tel que $j \in X'$ et $k \notin X'$
    $X' \leftarrow X' \cup \{k\}$
    $U' \leftarrow U' \cup \{(j, k)\}$
Fin Tant que
Output : $G' = [X', U']$

2.3. Code R de l'algorithme

Vous trouverez ci-dessous notre code R pour l'implémentation de l'algorithme présenté dans la sous-section précédente.

```
Prim=function(X,A)
{
  #Fonction qui détermine un arbre partiel de poids minimal
  #INPUT:
  #X l'ensemble des sommets
  #A est la matrice d'adjacence pondérée
  #OUTPUT:
  #U_prim la liste des arêtes de l'arbre partiel de poids minimal
  i=1 #on choisit au hasard un sommet de X
  x_prim=c(i) #ce sommet choisi est intégré dans la liste des sommets de
l'arbre partiel de poids minimal
  u_prim=c() #initialisation de la liste des arêtes de l'arbre partiel de
poids minimal
  cpt=1 #compteur des itérations
```

```
  while(setequal(X,x_prim)==FALSE) #tant que tous les sommets X du graphe
ne sont pas contenus dans les sommets X' de l'arbre partiel de poids
minimal
  {
    j=intersect(X,x_prim) #Sommet j appartenant à la liste des sommets de
l'arbre partiel de poids minimal
    k=setdiff(X,x_prim) #Sommet k n'appartenant pas à la liste des sommets
de l'arbre partiel de poids minimal

    if (length(j)==1 || length(k)==1)
    {
      #si il n'y a qu'un sommet dans la liste des sommets, on transformera
sous forme de matrice

successeurs=which(matrix(A[j,k]!=0,nrow=length(j),ncol=length(k)),arr.ind=T
RUE) #indice des successeurs de j parmi les sommets k
    }
    else
    {
      successeurs=which(A[j,k]!=0,arr.ind=TRUE) #indice des successeurs de
j parmi les sommets k
    }

    successeurs=cbind(j[successeurs[,1]],k[successeurs[,2]]) #indices des
successeurs de j parmi les sommets k dans la matrice d'adjacence pondérée A
    indice=which.min(A[successeurs]) #indice du successeur ayant le poids
minimal dans la matrice "successeurs"
    res=cbind(successeurs[indice,1],successeurs[indice,2]) #indices du
successeur ayant le poids minimal dans la matrice A
    j=res[1] #sommet appartenant à la liste des sommets de l'arbre partiel
ayant permis de déterminer le sommet k
    k=res[2] #sommet ayant le poids minimal n'appartenant pas à la liste
des sommets de l'arbre partiel
    arete=c(j,k) #indice des sommets j et k constituant l'arête de poids
minimal
    x_prim=union(x_prim,k) #liste des sommets de l'arbre partiel de poids
minimal
    u_prim[[cpt]]=arete #liste des arêtes de l'arbre partiel de poids
minimal
    cpt=cpt+1
  }

  print('X prim : ')
  print(x_prim)

  print('U prim :')
  print(u_prim)
}
```

2.4. Illustration sur des exemples

2.4.1. Exemple 1 : donné dans le projet

Nous illustrons le bon fonctionnement de l'algorithme en l'exécutant sur le graphe dont la matrice d'adjacence est donnée par :

$$\mathbf{A} = \begin{pmatrix} 0 & 5 & 8 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 4 & 2 & 0 & 0 \\ 8 & 0 & 0 & 0 & 5 & 2 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 & 7 \\ 0 & 2 & 5 & 0 & 0 & 0 & 3 \\ 0 & 0 & 2 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 7 & 3 & 3 & 0 \end{pmatrix}$$

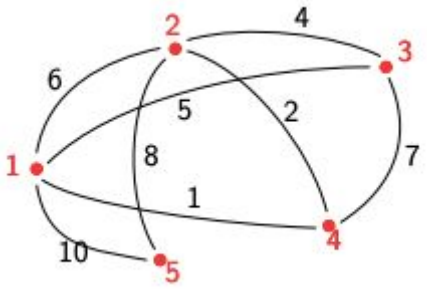
L'exécution du code R donné dans la sous-section précédente donne le résultat suivant :

```
X=c(1:7)

A=rbind(c(0,5,8,0,0,0,0),c(5,0,0,4,2,0,0),c(8,0,0,0,5,2,0),c(0,4,0,0,0,0,7)
,c(0,2,5,0,0,0,3),c(0,0,2,0,0,0,3),c(0,0,0,7,3,3,0))

Prim(X,A)

## [1] "X prim : "
## [1] 1 2 5 7 6 3 4
## [1] "U prim :"
## [[1]]
## [1] 1 2
##
## [[2]]
## [1] 2 5
##
## [[3]]
## [1] 5 7
##
## [[4]]
## [1] 7 6
##
## [[5]]
## [1] 6 3
##
## [[6]]
## [1] 2 4
```

L'arbre partiel de poids minimal est G = [X, U'] avec U' = {(1,2); (2,5); (5,7); (7,6); (6,3); (2,4)}.

2.4.2. Exemple 2 : donné dans le cours

Nous illustrons le bon fonctionnement de l'algorithme en l'exécutant sur le graphe suivant (dont la matrice d'adjacence est donnée dans le code R) :



L'exécution du code R donné dans la sous-section précédente donne le résultat suivant :

```
X=c(1:5)

A=rbind(c(0,6,5,1,10),c(6,0,4,2,8),c(5,4,0,7,0),c(1,2,7,0,0),c(10,8,0,0,0))

Prim(X,A)

## [1] "X prim : "
## [1] 1 4 2 3 5
## [1] "U prim :"
## [[1]]
## [1] 1 4
##
## [[2]]
## [1] 4 2
##
## [[3]]
## [1] 2 3
##
## [[4]]
## [1] 2 5
```

L'arbre partiel de poids minimal est G = [X, U'] avec U' = {(1,4); (4,2); (2,3); (2,5)}.

3. Algorithme 2 : Calcul des plus courts chemins par Ford-Bellman

3.1. Objet de l'algorithme

Étant donné un graphe valué dont les longueurs sont quelconques, mais qui ne possède pas de circuit de longueur négative. On représente ce graphe par sa matrice d'adjacence pondérée A. Le problème est de déterminer les longueurs des plus courts chemins entre un sommet s et tous les autres sommets du graphe .

La première étape consiste à initialiser la longueur du plus court chemin entre le sommet s et lui-même à 0 et pour les autres sommets à l'infini. Ensuite, pour tous les sommets i différents du sommet s, on essaye de mettre à jour la longueur du plus court chemin entre le sommet s et le sommet i. Pour cela, on calcule le minimum entre la valeur courante de la longueur du plus court chemin entre le sommet s et le sommet i et le minimum pour tous les prédécesseurs j du sommet i, entre la valeur courante de la longueur du plus court chemin entre s et j + la longueur entre j et le sommet i. On répète cela jusqu'à ce que plus aucune des valeurs des longueurs des plus courts chemins change.

Ce type de problème peut intervenir dans de nombreux contextes, comme par exemple dans les réseaux informatiques, routiers, ou bien encore de télécommunication, pour le cheminement.

3.2. Pseudo-code de l'algorithme

```
    Input : G = [X, U], s
1   π(s) ← 0
2   Pour tout i ∈ {1, 2, ..., N} \ {s} faire
3       π(i) ← +∞
4   Fin Pour
5   Répéter
6       Pour tout i ∈ {1, 2, ..., N} \ {s} faire
7           π(i) ← min(π(i), min_{j∈Γ⁻¹(i)} π(j) + l_ji) ;
8       Fin Pour
9   Tant que une des valeurs π(i) change dans la boucle Pour
10  Output : π
```

3.3. Code R de l'algorithme

Vous trouverez ci-dessous notre code R pour l'implémentation de l'algorithme présenté dans la sous-section précédente.

```
Ford_Bellman = function(X,A,s)
{
  #Détermine les longueurs des plus courts chemins entre un sommet s et les autres sommets
  #INPUT:
  #X l'ensemble des sommets
  #A est la matrice d'adjacence
  #s un sommet de X (sommet de départ)
  #OUTPUT:
   #N les longueurs des plus courts chemins entre s et tous les autres sommets de X

  p=c()
  p[s]=0 #Pour le sommet de départ, la longueur vaut 0
  s_barre=setdiff(X,s)
  for (i in s_barre)
  {
      p[i] = Inf #initialise la longueur de tous les autres sommets à l'infini
  }
  temp = p #variable temporaire contenant la valeur courante des longueurs
   change = 1 #variable permettant de tester si une des valeurs de p[i] change

  print("Initialisation : ")
  print(p)

  cpt=1
  while (change > 0) #si la valeur de la variable "change" est supérieure à 0, cela signifie qu'une des valeurs de p[i] a changé lors de l'itération
  {
      print(paste("Itération n°" , cpt))
      for (i in s_barre) #pour chaque sommet i différent du sommet de départ
      {
        pre=which(A[,i]!=0) #indices des prédécesseurs du sommet i
      res=min(p[pre]+A[pre,i]) #pour tous les prédécesseurs de i on prend la longueur du chemin de s jusqu'au sommet prédécesseur du sommet i + la longueur du sommet prédécesseur jusqu'au sommet i et on récupère la longueur minimale
      p[i]=min(p[i],res)  #valeur minimale entre le résultat obtenu précédemment et la valeur courante de p[i]
      }
```

```
      change=sum(as.numeric(temp!=p)) #somme des valeurs différentes entre
temp et p est supérieure à 0
      temp = p #mise à jour de la variable temporaire
      cpt=cpt+1
      print(p) #affichage de la valeur courante du vecteur p
  }
}
```

3.4. Illustration sur des exemples

3.4.1. Exemple 1 : donné dans le projet

Nous illustrons le bon fonctionnement de l'algorithme en l'exécutant sur le graphe dont la matrice d'adjacence est donnée par :

$$\mathbf{A} = \begin{pmatrix} 0 & 5 & 8 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 4 & -2 & 0 & 0 \\ 8 & 0 & 0 & 0 & 5 & 2 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 & 7 \\ 0 & -2 & 5 & 0 & 0 & 0 & 3 \\ 0 & 0 & 2 & 0 & 0 & 0 & -3 \\ 0 & 0 & 0 & 7 & 3 & -3 & 0 \end{pmatrix}$$

L'exécution du code R donné dans la sous-section précédente donne le résultat suivant :

```
X=c(1:7)
A=rbind(c(0,5,8,0,0,0,0),c(5,0,0,4,-2,0,0),c(8,0,0,0,5,2,0),c(0,4,0,0,0,0,7
),c(0,-2,5,0,0,0,3),c(0,0,2,0,0,0,-3),c(0,0,0,7,3,-3,0))
s=1
Ford_Bellman(X,A,s)
```

Toutefois, on s'aperçoit ici de la présence d'un circuit de longueur négative, c'est pourquoi la longueur des chemins tend vers - ∞.

3.4.2. Exemple 2 : donné dans le projet

Nous illustrons le bon fonctionnement de l'algorithme en l'exécutant sur le graphe dont la matrice d'adjacence est donnée par :

$$\mathbf{A} = \begin{pmatrix} 0 & 5 & 8 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 4 & 0 & 0 & 0 \\ 8 & 0 & 0 & 0 & 5 & 2 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 & 0 & 3 \\ 0 & 0 & 2 & 0 & 0 & 0 & -3 \\ 0 & 0 & 0 & 0 & 3 & -3 & 0 \end{pmatrix}$$

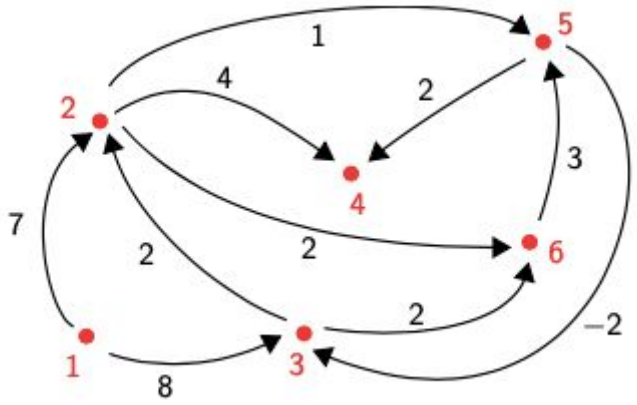
L'exécution du code R donné dans la sous-section précédente donne le résultat suivant :

```
X=c(1:7)
A=rbind(c(0,5,8,0,0,0,0),c(5,0,0,4,0,0,0),c(8,0,0,0,5,2,0),c(0,4,0,0,0,0,0)
,c(0,0,5,0,0,0,3),c(0,0,2,0,0,0,-3),c(0,0,0,0,3,-3,0))
s=7
Ford_Bellman(X,A,s)
## [1] "Initialisation : "
## [1] Inf Inf Inf Inf Inf Inf   0
## [1] "Itération n° 1"
## [1] Inf Inf Inf Inf   3  -3   0
## [1] "Itération n° 2"
## [1] Inf Inf  -1 Inf   3  -3   0
## [1] "Itération n° 3"
## [1]  7 12 -1 16  3 -3  0
## [1] "Itération n° 4"
## [1]  7 12 -1 16  3 -3  0
```

Les plus courts chemins entre le sommet 7 et les autres sommets du graphe sont de longueur π*=(7,12,-1,16,3,-3,0).

3.4.3. Exemple 3 : donné dans le cours

Nous illustrons le bon fonctionnement de l'algorithme en l'exécutant sur le graphe suivant (dont la matrice d'adjacence est donnée dans le code R) :



L'exécution du code R donné dans la sous-section précédente donne le résultat suivant :

```
A=rbind(c(0,7,8,0,0,0),c(0,0,0,4,1,2),c(0,2,0,0,0,2),c(0,0,0,0,0,0),c(0,0,-
2,2,0,0),c(0,0,0,0,3,0))
X=1:6
s=1
Ford_Bellman(X,A,s)
## [1] "Initialisation : "
## [1]   0 Inf Inf Inf Inf Inf
## [1] "Itération n° 1"
## [1]  0  7  8 11  8  9
## [1] "Itération n° 2"
## [1]  0  7  6 10  8  8
## [1] "Itération n° 3"
## [1]  0  7  6 10  8  8
```

Les plus courts chemins entre le sommet 1 et les autres sommets du graphe sont de longueur $\pi^*=(0,7,6,10,8,8)$.

4. Algorithme 3 : Détermination d'un flot maximal dans un réseau avec capacités par Ford-Fulkerson

4.1. Objet de l'algorithme

Soit G = [X,U,C] un réseau avec capacité. Le flot de valeur maximale phi est parmi l'ensemble des flots réalisables celui qui maximise la quantité V(phi). L'algorithme de Ford-Fulkerson consiste à déterminer un flot maximum dans un graphe entre un point initial, communément appelé "source" et un sommet final, le "puits". Le principe est simple : On parcourt le graphe à la recherche d'une suite améliorante qui permettrait d'augmenter le flot. Pour se faire, l'algorithme utilise le marquage de sommets pour déterminer en fonction du flot s'il existe une coupe qui empêche d'améliorer la valeur du flot. L'enjeu est de faire passer un maximum de volume sans faire saturer le réseau.

4.2. Pseudo-code de l'algorithme

Input : $G = [X, U, C]$, φ un flot réalisable
1 $m_s \leftarrow (\infty, +)$ et $S = \{s\}$
2 **Tant que** $\exists (j \in \overline{S}, i \in S) : (c_{ij} - \varphi_{ij} > 0) \vee (\varphi_{ji} > 0)$ **faire**
3 **Si** $c_{ij} - \varphi_{ij} > 0$ **faire**
4 $m_j \leftarrow (i, \alpha_j, +)$ avec $\alpha_j = \min\{\alpha_i, c_{ij} - \varphi_{ij}\}$
5 **Sinon Si** $\varphi_{ji} > 0$ **faire**
6 $m_j \leftarrow (i, \alpha_j, -)$ avec $\alpha_j = \min\{\alpha_i, \varphi_{ji}\}$
7 **Fin Si**
8 $S \leftarrow S \cup \{j\}$
9 **Si** $j = p$ **faire**
10 $V(\varphi) \leftarrow V(\varphi) + \alpha_p$
11 Aller en 14
12 **Fin Si**
13 **Fin Tant que**
14 **Si** $p \in S$ **faire**
15 **Tant que** $j \neq s$ **faire**
16 **Si** $m_j(3) = +$ **faire**
17 $\varphi_{m_j(1)j} \leftarrow \varphi_{m_j(1)j} + \alpha_p$
18 **Sinon Si** $m_j(3) = -$ **faire**
19 $\varphi_{jm_j(1)} \leftarrow \varphi_{jm_j(1)} - \alpha_p$
20 **Fin Si**
21 $j \leftarrow m_j(1)$
22 **Fin Tant que**
23 Aller en 1
24 **Sinon faire**
25 **Output :** φ
26 **Fin Si**

4.3. Code R de l'algorithme

Vous trouverez ci-dessous notre code R pour l'implémentation de l'algorithme présenté dans la sous-section précédente.

```
Ford_Fulkerson = function(X,A,s,p)
{
  n = nrow(A) #Nombre de ligne de la matrice A en paramètre
  sommets = X #ensemble de sommets
  flot_debut = matrix(0,nrow=n,ncol=n)
  marque = matrix(0,ncol=3,nrow=n) #matrice pour les 3 paramètres de ms
   s_marque = s #affecte le sommet source aux sommets marqués (car pas
besoin de le modifier)
   s_pasmarque = setdiff(sommets,s_marque) #On met les sommets pas marqués
pour le moment
  matrice_arete_bool = (A-flot_debut>0) #matrice de booléen des arêtes
  indice_aretes=
which(matrix(matrice_arete_bool[s_marque,s_pasmarque]==TRUE,nrow=length(s_m
arque),ncol=length(s_pasmarque)), arr.ind=TRUE) #le which nous permet de
récupérer les indices des arêtes
  flot_maximum = 0 #Initialiser le flot maximum


################################################################################

  while (TRUE)
  {
      #On cherche si deux sommets satisfont les conditions
      if(length(indice_aretes>0))
      {
      i=s_marque[indice_aretes[1,1]]
      j=s_pasmarque[indice_aretes[1,2]]
      }
      else
      {
      break;
      }
      #Si Cij - Phiij > 0 (Cij capacité de l'arc et Phiij le flot) alors :
      if (A[i,j]-flot_debut[i,j] > 0)
      {
      #Calculer la valeur maximum et on marque avec 1 qui correspondra au
troisième argument, +
      a = abs(min(marque[i,2],A[i,j]-flot_debut[i,j]))
      marque[j,] = c(i,a,1)
      }
      else
      {
      if (flot_debut[j,i] >-1)
```

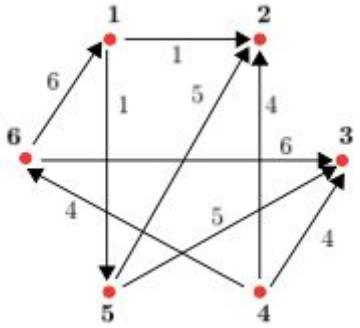
```
      {
      #Marquer avec -1 qui correspondra au troisième argument, -
      a = abs(min(marque[i,2],flot_debut[j,i]))
      marque[j,] = c(i,a,-1)
      }
      }
      #Réinitialiser la matrice 'indice_arêtes'
      s_marque = union(s_marque,j)#On fait l'union des sommets marqués et j
      s_pasmarque = setdiff(s_pasmarque,s_marque)#On récupère ceux qui ne
sont pas marqués ici
      indice_aretes=
which(matrix(matrice_arete_bool[s_marque,s_pasmarque]==TRUE,nrow=length(s_m
arque),ncol=length(s_pasmarque)),arr.ind=TRUE)


#############################################################################

      if(j==p) #Si notre j correspond au puit, alors :
      {
      #On calcule l'accroissement du flot initial
      if(is.element(p,s_marque))
      {
      while(j!=s) #Si j est différent de la source alors :
      {
            if(marque[j,3] == 1) #Si mj est +, alors :
            {
            #Modifier le flot entre i et j
            flot_debut[marque[j,1],j]=flot_debut[marque[j,1],j]+marque[p,2]
            }
          else if(marque[j,3] == -1)
            {
            #Modifier le flot entre j et i
            flot_debut[j,marque[j,1]]=flot_debut[j,marque[j,1]]-marque[p,2]
            }
            j = marque[j,1]
      }
      #Nouvelle valeur du flot maximum
      flot_maximum = flot_maximum+marque[p,2]
      }
      marque[s,c(2,3)] = c(Inf,1)
      s_marque = s
      s_pasmarque = setdiff(sommets,s_marque)
      matrice_arete_bool = (A-flot_debut>0)
      indice_aretes=
which(matrix(matrice_arete_bool[s_marque,s_pasmarque]==TRUE,nrow=length(s_m
arque),ncol=length(s_pasmarque)),arr.ind=TRUE)
      }
  }

  #############################################################################
```

```
  #Calculer l'accroissement de la valeur du flot réalisable
  if(is.element(p,s_marque))
  {
      while(j != s)
      {
      if(marque[j,3] == 1)
      {
      #Modifier le flot entre i et j
      flot_debut[marque[j,1],j] = flot_debut[marque[j,1],j]+marque[p,2]
      }
      else if(marque[j,3] == -1)
      {
      #Modifier le flot entre j et i
      flot_debut[j,marque[j,1]] = flot_debut[j,marque[j,1]]-marque[p,2]
      }
      j = marque[j,1]
      }
  }

  ###########################################################################
  else
  {
      return(flot_maximum) #on retourne le flot maximum
  }

}
```

4.4. Illustration sur des exemples

4.4.1. Exemple 1 : donné dans le projet

Nous illustrons le bon fonctionnement de l'algorithme en l'exécutant sur le graphe suivant (dont la matrice d'adjacence est donnée dans le code R) :



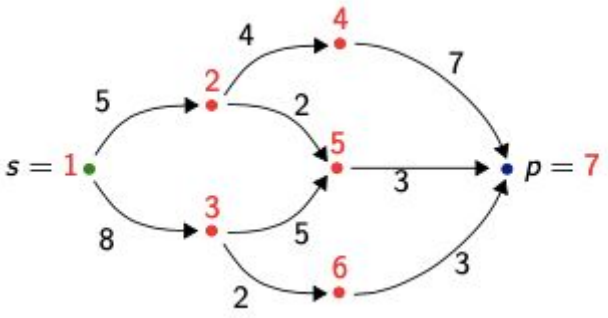
L'exécution du code R donné dans la sous-section précédente donne le résultat suivant :

```
A=
rbind(c(0,1,0,0,1,0),c(0,0,0,0,0,0),c(0,0,0,0,0,0),c(0,4,4,0,0,4),c(0,5,5,0
,0,0),c(6,0,6,0,0,0))
X=1:nrow(A)
Ford_Fulkerson(X,A,4,2)
## [1] 6
```

La valeur maximale obtenue pour le flot est de 6.

4.4.2. Exemple 2 : donné dans le cours

Nous illustrons le bon fonctionnement de l'algorithme en l'exécutant sur le graphe suivant (dont la matrice d'adjacence est donnée dans le code R) :



L'exécution du code R donné dans la sous-section précédente donne le résultat suivant :

```
A=rbind(c(0,5,8,0,0,0,0),c(0,0,0,4,2,0,0),c(0,0,0,0,5,2,0),c(0,0,0,0,0,0,7)
,c(0,0,0,0,0,0,3),c(0,0,0,0,0,0,3),c(0,0,0,0,0,0,0))
X=1:nrow(A)
Ford_Fulkerson(X,A,1,7)
## [1] 9
```

La valeur maximale obtenue pour le flot est de 9.

5. Conclusion

Dans ce projet, nous avons implémenté 3 algorithmes de graphes: l'algorithme de Prim, de Ford-Bellman, et celui de Ford-Fulkerson. L'étude théorique de ces algorithmes en cours magistral nous a permis de comprendre leur fonctionnement, leur utilité et d'ensuite les développer nous-même avec une technologie concrète dans l'optique de les confronter avec notre développement réalisé sur papier. C'est un travail très complémentaire, qui challenge à la fois notre compréhension des algorithmes et leur fonctionnement, ainsi que notre capacité à les développer avec R.

Les principales difficultés rencontrées ont été de partir de "rien", notamment sur l'algorithme de Ford-Fulkerson qui est très complexe à comprendre et à conceptualiser. La condition d'arrêt dans l'algorithme de Ford-Bellman a été un peu problématique mais nous avons réussi à pallier ce détail.

Pour les pistes d'améliorations, nous pensons que certaines parties peuvent être optimisées, et le code allégé. Notamment pour l'algorithme de Ford-Bellman, pour lequel nous pourrions gérer les cas particuliers des graphes possédant un circuit de longueur négative.

En définitive ce fut un travail long et compliqué mais qui retrace tout notre programme sur les graphes de ce semestre. Un travail applicatif et concret est selon nous essentiel pour illustrer les concepts appris, et gratifiant étant donné que nous l'avons mené à bien.