Chapitre 2

Algorithme 1 : détection de l'arbre recouvrant de poids minimal par Prim

2.1 Objet de l'algorithme

Etant donné un graphe G = (X, U) non orienté, pondéré et connexe, représenté par sa matrice d'adjacence pondérée A, le problème est de déterminer un arbre couvrant de G tel que le poids soit minimum. Donc l'algorithme de Prim est un algorithme déterminant un arbre couvrant minimal d'un graphe connexe valué et non orienté. C'est-à-dire qu'il trouve un sous-ensemble d'arêtes formant un arbre incluant tous les sommets, tel que la somme des poids de chaque arête soit minimal. Si le graphe n'est pas connexe l'algorithme ne déterminera l'arbre couvrant minimal que d'une composante connexe du graphe. L'arbre couvrant minimum peut s'interpréter de différentes manières selon ce que représente le graphe. De manière générale si on considère un réseau où un ensemble d'objets doivent être reliés entre eux (par exemple un réseau électrique et des habitations), l'arbre couvrant minimum est la façon de construire un tel réseau en minimisant un coût représenté par le poids des arêtes (par exemple la longueur totale de câble utilisée pour construire un réseau électrique). Les arbres couvrants sont notamment utilisés dans plusieurs types de réseaux, comme les réseaux téléphoniques et les réseaux de distribution. Hors du contexte des réseaux, ils sont utiles par exemple pour le partitionnement de données et le traitement d'image.

2.2 Pseudo-code de l'algorithme

Nous rappelons ci-dessous le pseudo-code de l'algorithme de Prim. Nous faisons remarquer que cet algorithme est valide uniquement pour les graphes non orientés, pondérés et connexe, ce que nous supposerons donc dans la suite de cette section.

```
      Input : G = [X; U]
1   X'← {i} où i est un sommet de X pris au hasard
2   U'← ∅ ;
3   Tant que X' ≠ X faire
5   Choisir une arête (j; k) de poids minimal tel que j ∈ X' et k ∉ X'
6   X' ← X' ∪ {k}
7   U' ← U' ∪ {(j,k)}
8   Fin Tant que
9   Output : G' = [X'; U']
```

2.3 Code R de l'algorithme

Voici ci-dessous le code R de notre implémentation de l'algorithme présenté dans le paragraphe précédent.

```
 Prim = function(X,A)
{
#arbre recouvrant de poids minimal
#INPUT
#X est l'ensemble des sommets
#A est la matrice d'adjacence ponderee
#OUTPUT
#Up la matrice couvrant le poids minimal
n=length(X)
#On prend un sommet au hasard
Xp = vector()
Xp = append(Xp,sample(X,1))

#on definit X_prime_barre
Xpb = setdiff(X,Xp)

#on cree la matrice de possibilites de choix d'aretes allant de Xp a Xpb
Mc=matrix(A[Xp,Xpb],nrow=length(Xp),ncol=length(Xpb))

#on definit U_prime
Up = matrix(0,nrow=n,ncol=n)

while(length(Xp)<n) # arreter des que Xp = X
{
#on remplace les zeros dans matrice des possibilites par des NA
#pour ne pas les selectionnes
Mc = replace(Mc, Mc==0, NA)
# choix de l'arrete avec poids minimal
min = min(Mc,na.rm=TRUE)
print(Mc)
print(min)
#on cherche les indices du nombre min(du premier si il y'a des exaequos)
arete=which(Mc == min,arr.ind=TRUE)

#une fois les indices obtenus on change le referentiel pour obtenir les sommets
#connectes qui ont ce poids min

x = Xp[arete[1,1]]
y = Xpb[arete[1,2]]

#on met a jour les sommets de l'ensemble X_prime et X_prime_barre
Xp = append(Xp,y)
Xpb = setdiff(X,Xp)

#On rajoute le poids de l'arete dans U_prime
Up[x,y]= min
Up[y,x]= min

#on recalcule la nouvelle matrice des possibilites de selection
Mc=matrix(A[Xp,Xpb],nrow=length(Xp),ncol=length(Xpb))

}
return(Up)
}
```

2.4 Illustration sur un exemple

Nous illustrons le bon fonctionnement de l'algorithme implémenté en l'exécutant sur l'exemple suivant :

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

FIGURE 2.1 – Matrice correspondante.

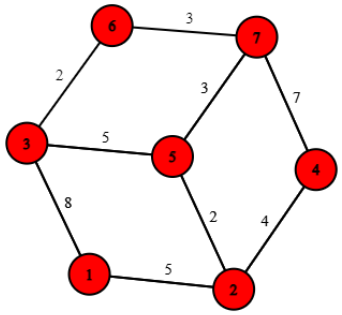

FIGURE 2.2 – Représentation Graphique.

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

```
> X=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))
> Up = Prim(X,A)
> print(paste("L'arbre recouvrant de poids min est : "))
[1] "L'arbre recouvrant de poids min est : "
> print(Up)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    0    5    0    0    0    0    0
[2,]    5    0    0    4    2    0    0
[3,]    0    0    0    0    0    2    0
[4,]    0    4    0    0    0    0    0
[5,]    0    2    0    0    0    0    3
[6,]    0    0    2    0    0    0    3
[7,]    0    0    0    0    3    3    0
>
```

FIGURE 2.3 – Résultat exécution code R.

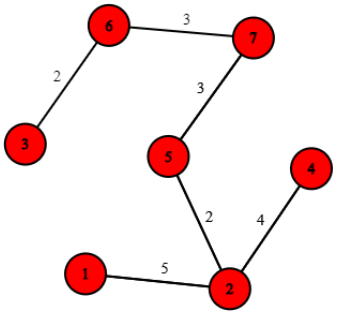

FIGURE 2.4 – Représentation graphique de l'arbre recouvrant de poids min obtenu(G' = [X',U']).

Chapitre 3

Algorithme 2 : calcul du plus court chemin par Ford-Bellman

De nombreux problèmes concrets se traduisent par la recherche d'un chemin de longueur minimale :
• Recherche de l'itinéraire le plus court (valuation d'un arc = distance entre deux villes).
• Recherche de l'itinéraire le plus rapide (valuation d'un arc = temps pour parcourir la route entre deux villes).
• Routage dans des réseaux
• etc.
Un problème fondamental, et qui se pose fréquemment dans les applications, est celui de la recherche de chemins de valuation minimale (on dit aussi "plus court chemin") ou maximale. Pour résoudre ce problème il existe plusieurs algorithmes, Bellman-Ford, Dijkstra. . . , permettant d'obtenir de tels chemins.

3.1 Objet de l'algorithme

Etant donné un graphe G = (X, U) pondéré et représenté par sa matrice d'adjacence pondérée A, l'algorithme de Ford-Bellman est un algorithme qui permet de trouver des plus courts chemins, depuis un sommet source donné aux autres sommets, pour des valuations quelconques. Contrairement à l'algorithme de Dijkstra, l'algorithme de Ford-Bellman autorise la présence de certains arcs de valuation négative et permet de détecter l'existence d'un circuit absorbant, c'est-à-dire de valuation totale négative, accessible depuis le sommet source.

3.2 Pseudo-code de l'algorithme

Nous rappelons ci-dessous le pseudo-code de l'algorithme de Ford-Bellman. Nous faisons remarquer que cet algorithme est valide uniquement pour les graphes pondérés et non forcement connexe, ce que nous supposerons donc dans la suite de cette section.

```
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 qu'une des valeurs π(i) change dans la boucle Pour
10 Output : π
```

3.3 Code R de l'algorithme

```
    Ford_Bellman <- function(X,A,s){
#Calcul de plus court chemin
#INPUT
#X est l'ensemble des sommets
#A est la matrice d'adjacence pondere
#s le sommet initial
#OUTPUT
#pi : vecteur des longueurs des plus courts chemins entre s et les autres sommets.
n=length(X)
Xb=setdiff(X,s) # ensemble des noeuds sauf s
pi=rep(0,n)
pi[s]=0
#initialisation
for(i in Xb){
pi[i]=Inf
}
iter =1
pi_old=pi #on garde une copie de pi pour comparer a la fin
repeat {
print(paste("+++ Iteration:",iter))
print(paste("pi avant mise a jour :"))
print(pi_old)
for(i in Xb){
pr_i=which(A[,i]!=0) #predecesseur de i
val_i=c() # vecteur qui contindra la somme de ponderation de predecesseur
for(j in pr_i){
val_i=c(val_i,pi[j]+A[j,i])
}
pi[i]=min(pi[i],val_i) # mise a jour de pi[i]
}#fin pour
print(paste("pi apres mise a jour :"))
print(pi)
if(identical(pi_old,pi)==FALSE){
pi_old = pi
iter=iter+1
}else
break # fin de la boucle

} #fin repeat
print(paste("le vecteur PI est :"))
print(pi)
return(pi)
}
```

3.4 Illustration sur un exemple

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

Exemple 1 :
L'exécutions du code R présenté dans la sous-section précédente donne le résultat suivant :

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

FIGURE 3.1 – Matrice de l'exemple 1.

```
> #matrice exmeple projet Q2
> 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
> pi=Ford_Bellman(X,A,s)
```

```
[1] 0 -15203 -15205 -15203 -15207 -15213 -15216
[1] "pi après mise à jour :"
[1]      0 -15209 -15211 -15209 -15213 -15219 -15222
[1] "+++ Iteration: 2540"
[1] "pi avant mise à jour :"
[1]      0 -15209 -15211 -15209 -15213 -15219 -15222
[1] "pi après mise à jour :"
[1]      0 -15215 -15217 -15215 -15219 -15225 -15228
[1] "+++ Iteration: 2541"
[1] "pi avant mise à jour :"
[1]      0 -15215 -15217 -15215 -15219 -15225 -15228
[1] "pi après mise à jour :"
[1]      0 -15221 -15223 -15221 -15225 -15231 -15234
[1] "+++ Iteration: 2542"
[1] "pi avant mise à jour :"
[1]      0 -15221 -15223 -15221 -15225
```

FIGURE 3.2 – résultat de l'exécution sur l'exemple 1.

Comme on peut le constater, notre programme boucle à l'infinie parce qu'il y a la présence d'un circuit de poids total strictement négatif(circuit absorbant), donc il n'existe pas de plus court chemin.

Exemple 2 :

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

FIGURE 3.3 – Matrice de l'exemple 1.

En prenant 7 comme sommet de départ, le résultat est le suivant :

```
> #matice exemple Q3
> X=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
> pi=Ford_Bellman(X,A,s)
[1] "+++ Iteration: 1"
[1] "pi avant mise à jour :"
[1] Inf Inf Inf Inf Inf Inf   0
[1] "pi après mise à jour :"
[1] Inf Inf Inf Inf   3  -3   0
[1] "+++ Iteration: 2"
[1] "pi avant mise à jour :"
[1] Inf Inf Inf Inf   3  -3   0
[1] "pi après mise à jour :"
[1] Inf Inf  -1 Inf   3  -3   0
[1] "+++ Iteration: 3"
[1] "pi avant mise à jour :"
[1] Inf Inf  -1 Inf   3  -3   0
[1] "pi après mise à jour :"
[1]  7 12 -1 16  3 -3  0
[1] "+++ Iteration: 4"
[1] "pi avant mise à jour :"
[1]  7 12 -1 16  3 -3  0
[1] "pi après mise à jour :"
[1]  7 12 -1 16  3 -3  0
[1] "le vecteur PI est :"
[1]  7 12 -1 16  3 -3  0
>
```

FIGURE 3.4 – resultat de l'execution sur l'exemple 2.

Sommet d'arrivé	1	2	3	4	5	6	7
Plus court chemin	7	12	-1	16	3	-3	0

TABLE 3.1 – Les Plus courts chemins en partant du sommet 7

Chapitre 4

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

Le problème de flot maximal consiste à transporter la quantité maximale possible d'une origine (source) à une destination (puits) données, sans dépasser les capacités des arcs. On veut par exemple trouver le trafic maximal entre deux villes d'un réseau routier dont on connait la capacité (nombre de voiture par heure sur chaque tronçon) et sans dépasser la capacité de chaque tronçon. Les réseaux de transport sont les problèmes de circulation d'objets (voiture, information, liquide, électricité ...) dans un réseau (routier, informatique, canalisation, électrique ...). Cette circulation doit s'écouler depuis une source, où il est produit, jusqu'à un puits, où il est consommé.

4.1 Objet de l'algorithme

L'algorithme de Ford-Fulkerson permet de résoudre le problème de la détermination d'un flot maximal dans un réseau c-à-d en déterminant un flot réalisable entre un sommet initial s et un sommet d'arriver t. Soit G=(X,U) un graphe orienté et sans boucle, représenté par sa matrice de pondération qu'on appelle ici le flot, admettant aussi un point de départ(source s) et un point d'arrivé(puit t), c-à-d qu' aucun arc n'arrive à la source et aucun arc ne quitte le puits. L'algorithme de Ford-Fulkerson permet de trouver une chaîne augmentante et augmenter le flot sur cette chaine, en suivant deux phases. La phase de marquage qui est la recherche d'une chaîne augmentant et la phase d'augmentation qui permet d'augmenter le flot sur la chaine trouvé en phase de marquage. Ces deux phases sont répétées jusqu'au moment où il n'existe plus de chaîne augmentante et par suite le flot courant est maximal, on arrête.

4.2 Pseudo-code de l'algorithme

Nous rappelons ci-dessous le pseudo-code de l'algorithme de Ford-Fulkerson. Nous faisons remarquer que cet algorithme est valide uniquement pour les graphes orientés pondérés positifs ou nuls, sans circuit et admettant d'autres caractéristiques comme la conservation du flux, ce que nous supposerons donc dans la suite de cette section.

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

Voici ci-dessous le code R de notre implémentation de l'algorithme présenté dans le paragraphe précédent.

```
Ford_Fulkerson = function(X,A,s,p,Flr,m)
{
#Definir le flot max
#INPUT
#s est la source
#p est le puit
#X les sommets
#A matrice pondere des capacites
#m : Matrice de marquage
#Flr : Matrice des flots realisables
#OUTPUT : le flot max


#definition du Flotmax
Flotmax = 0
#definition de alphaj
alphaj = 0

#L'ensemble S
S = vector()

#table de marquage
m[s,] = c(NA,Inf,1)
```

```
S = append(S,s)

#permet d'obtenir la position des arcs ou le flot est diff de la capacite de l'arc
R1=A-Flr>0

#permet d'obtenir la position des arcs ou le flot de j a i est sup a zero
R2=t(Flr)>0

#l'union de R1 et R2
C=R1|R2

#Permet d'avoir S barre
Sb=setdiff(X,S)

#renvoi la position des arcs respectant les conditions de la boucle
Cnd=which(matrix(C[S,Sb]==TRUE,nrow=length(S),ncol=length(Sb)),arr.ind=TRUE)


while(length(Cnd)>0)#ligne 2 pseudo-code
{
x = S[Cnd[1,1]]
y = Sb[Cnd[1,2]]

if(R1[x,y]==TRUE)#ligne 3 pseudo-code
{
V = A[x,y]-Flr[x,y]
alphaj = min(c(m[x,2],V))
m[y,] = c(x,alphaj,1)
}
else if(R2[x,y]==TRUE){# ligne 5 pseudo-code
V = Flr[y,x]
alphaj = min(c(m[x,2],V))
m[y,] = c(x,alphaj,-1)
}
S = append(S,y)#ligne 8 pseudo-code
if(y == p){
Flotmax = Flotmax + alphaj
break
}
Sb=setdiff(X,S)
Cnd=which(matrix(C[S,Sb]==TRUE,nrow=length(S),ncol=length(Sb)),arr.ind=TRUE)
#Nous remettons a jour la matrice de condition de la boucle
}
if(is.element(p,S))#Nous verifions que le puit appartienne a l'ensemble S
{
y = p
x = m[y,1]
while(y != s)
{
if(m[y,3]==1)#ligne 16 pseudo-code
{
Flr[x,y] = Flr[x,y] + m[p,2]
}
else if(m[y,3]==-1)#ligne 18 pseudo-code
{
Flr[x,y] = Flr[x,y] - m[p,2]
}
y = m[y,1]
```

```
x = m[y,1]
}
Flotmax = Flotmax + Ford_Fulkerson(X,A,s,p,Flr,m)#Nous utilisons la recurssion
#pour calculer le flotmax et rappeler la fonction
#jusqu'a ce que le puit n'appartienne plus a l'ensemble S

}
else
{
print(Flr)
print(m)
return(Flotmax)
}
}
```

4.4 Illustration sur un exemple

Nous illustrons le bon fonctionnement de l'algorithme implémenté en l'exécutant sur l'exemple suivant :

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 4 & 0 & 0 & 4 \\ 0 & 5 & 5 & 0 & 0 & 0 \\ 6 & 0 & 6 & 0 & 0 & 0 \end{pmatrix}$$

FIGURE 4.1 – Matrice correspondante.

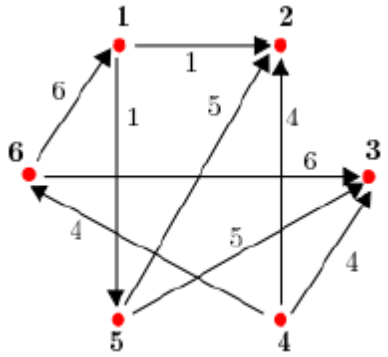

FIGURE 4.2 – Représentation Graphique.

• Exécutons l'algorithme de Ford-Fukelson sur l'exemple ci dessus en prenant une source de 4 et un puit de 2

```
> X=1:6
>
> B=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))
> s=4
> p=2
> #Matrice des flots realisables
>
> Flr=matrix(0,nrow=length(X),ncol=length(X))
>
> #Matrice de marquage m
>
> m= matrix(NA,nrow =length(X),ncol =3)
>
> valflotMax = Ford_Fulkerson(X,B,s,p,Flr,m)
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    0    1    0    0    1    0
[2,]    0    0    0    0    0    0
[3,]    0    0    0    0    0    0
[4,]    0    4    0    0    0    2
[5,]    0    1    0    0    0    0
[6,]    2    0    0    0    0    0
     [,1] [,2] [,3]
[1,]    6    2    1
[2,]    5    1    1
[3,]    4    4    1
[4,]   NA  Inf    1
[5,]    1    1    1
[6,]    4    2    1
> print(paste("Le flot de valeur max obtenu par l'algo de Ford Fulkerson est : ",valflotMax))
[1] "Le flot de valeur max obtenu par l'algo de Ford Fulkerson est :  6"
>
```

• Le **flot de valeur Max** alors obtenu est de **6**.

• **La première matrice** est **la matrice des flots réalisables** a la fin de l'algorithme de Ford-fukelson

• **La deuxième matrice** est **La table de marquage des sommets** au fur et a mesure de l'algorithme

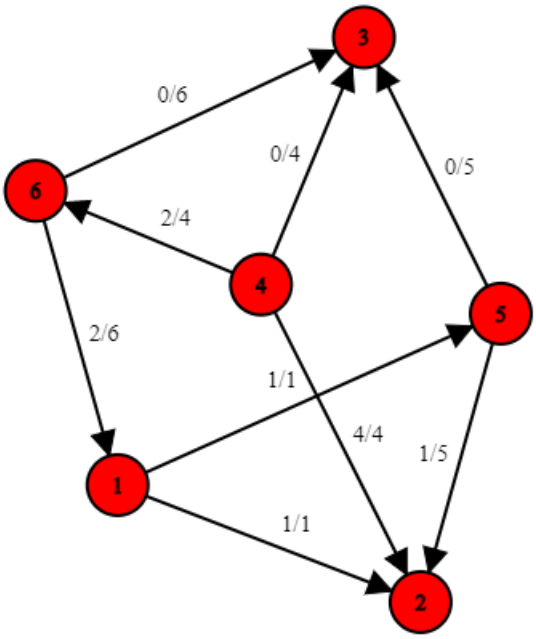

FIGURE 4.3 – Représentation Graphique du réseau et flot réalisable obtenu(puit = 2 ,source = 4).

• En prenant 4 comme source et 3 comme puit, On obtient une valeur de flot Max obtenu de 8.

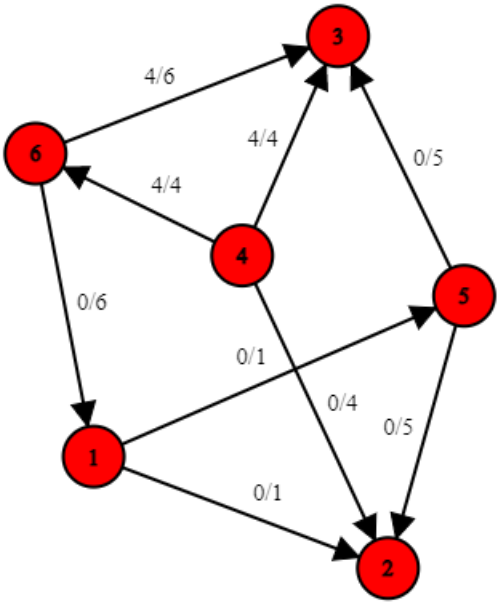

FIGURE 4.4 – Représentation Graphique du réseau et flot réalisable obtenu(puit = 3 ,source = 4).