Dossier
Optimisation et Recherche Opérationnelle

Julien Ah-Pine

Université Lyon 2 / ICOM
M1 Informatique 2019/2020

Contenu

Introduction

Ce dossier entre dans le cadre du contrôle continu des connaissances dans le contexte du cours de théorie des graphes.

Ce rapport comporte trois algorithmes : Kruskal, Ford-Bellman et Ford-Fulkerson. Pour chacun de ces algorithmes nous détaillerons les objectifs, le pseudo code ainsi que le code R et enfin l'utilisation sur un exemple.

Algorithme n°1 : Kruskal

Cet algorithme a pour objectif de déterminer un arbre recouvrant de poids minimal, c'est-à-dire de faire un chemin entre tous les sommets de l'arbre en minimisant le poids des arêtes. Pour se faire, nous utiliserons un graphe non orienté pondéré.

Pour illustrer l'intérêt de cet algorithme voici un exemple concret (et local !) : pendant la fête des lumières, certaines applications proposent des chemins optimisés pour voir toutes les « prestations visuelles » disponibles tout en minimisant le nombre total de pas à faire. Ce genre d'application est particulièrement utile pour les personnes âgées ou encore à mobilité réduite.

Pour rappel, voici le pseudo code de l'algorithme de Kruskal :

Input : $G = [X, U]$
Ranger les arêtes de U par ordre de poids croissants
$U' \leftarrow \{u_1\}$; $k \leftarrow 1$
Tant que $|U'| \neq N - 1$ **faire**
    $k \leftarrow k + 1$
    Tant que il existe un cycle dans $U' \cup \{u_k\}$ **faire**
        $k \leftarrow k + 1$
    Fin Tant que
    $U' \leftarrow U' \cup \{u_k\}$
Fin Tant que
Output : $G' = [X, U']$ est l'arbre partiel de poids minimum

Afin de l'implémenter sous R nous avons dû coder deux fonctions essentielles, la première qui, à partir de la matrice d'adjacence A, liste les arêtes ainsi que les sommets associés, et la seconde qui dans un premier temps, vérifie que la matrice d'adjacence est symétrique, et ensuite teste la présence d'un cycle dans l'arbre.

Pour la première fonction, voici ce que nous avons fait :

```r
# Création de la matrice comportant la liste des arêtes et des sommets qui les constituent
liste_aretes<-function(A){

  # Reccupération de la taille de la matrice
  Arow<-nrow(A)
  Acol<-ncol(A)

  # Création de la matrice finale
  U<-rbind()

  # Initialisation d'un compteur
  compteur<-0

  # Parcours de la i ème ligne
  for (i in 1:Arow){
    compteur<-compteur+1

    # Parcours de la j ème colonne de a i ème ligne
    for (j in compteur:Acol){

      #Vérification du poids de la présence d'arête
      if (A[i,j]>0){

        #Remplissage de la matrice
        a<-i
        b<-j
        c<-A[i,j]
        U<-rbind(U,c(a,b,c))


      }

    }

  }
  return(U)
}
```

Cette fonction « liste_aretes » nécessite en entré la matrice d'adjacence de l'arbre que l'on souhaite étudier et ressort en résultat une matrice comportant les numéros des deux sommets qui composent l'arête dans les deux premières colonnes et le poids de l'arête dans la troisième colonne. Voici un exemple pour illustrer la fonction.

```r
#Q2
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))
liste_aretes(A)
```

```
     [,1] [,2] [,3]
[1,]    1    2    6
[2,]    1    3    5
[3,]    1    4    1
[4,]    1    5   10
[5,]    2    3    4
[6,]    2    4    2
[7,]    2    5    8
[8,]    3    4    7
```

Ensuite il a fallu coder une nouvelle fonction afin de tester la présence d'un cycle dans l'arbre. Mais avant cela, nous avons créé des fonctions en amont afin de pouvoir symétriser la matrice d'adjacence dans le cas où elle ne l'était pas déjà.

```r
#Teste si la matrice donnée en paramètre A est symétrique. Renvoie un booléen.
isSymetric=function(A){
  return(!is.element(FALSE,t(A)==A));
}

#On rend la matrice symétrique (Les poids n'ont aucune importance dans cet algo)
setSymetric=function(A){
  return(t(A)+A);
}

#Renvoie la liste des successeurs du sommet s dans la matrice d'adjacence A
#Pré-requis entrée : A est symétrique et de dimension minimale 1
listeSuccesseurs=function(A,s){
  return(which(A[s,]!=0));
}

#Renvoie la liste des sommets découverts
listeNonDecouverts=function(d){
  return(which(d==-1));
}

#Renvoie la liste des sommets découverts
listeDecouverts=function(d){
  return(which(d==0));
}

#Renvoie la liste des sommets fermés
listeFermes=function(d){
  return(which(d==1));
}
```

Une fois ces fonctions créées, nous avons codé la fonction « test_cycle » ci-dessous :

```r
test_cycle = function(A,s){
  #On vérifie que la matrice soit carrée
  if(length(A[1,])!=length(A[,1])){
    print("Erreur, ca ne peut pas être une matrice d'adjacence");
    return(FALSE);
  }

  #On teste la symétrie de A
  if(!isSymetric(A)){
    A=setSymetric(A);
  }
  n=length(A[1,]);

  P=c(s);
  #Initialisation des di à -1
  d=c(1:n);
  for(i in 1:n){
    d[i]=-1;
  }
  d[s]=0;
  i=s;

  while(length(P)!=0){

    #Ensemble des successeurs de i non encore découvert
    S=intersect(listeSuccesseurs(A,i),listeNonDecouverts(d));

    if(length(S)!=0){
      j=S[1];
      d[j]=0;
      P=union(P,j);
      i=j;
    }
```

```
    else{

      #Ensemble des successeurs de i déjà découverts ou fermé
      t=intersect(listeSuccesseurs(A,i),union(listeDecouverts(d),listeFermes(d)));

      #P étant une LIFO les premiers élements sont les derniers de la liste. On supprime donc les 2 premiers sommets de P dans P2
      P2=setdiff(P,c(P[length(P)-1],P[length(P)]));

      if(length(intersect(t,intersect(P2,s)))!=0){
        return(TRUE);
      }
      else{
        d[i]=1;
        #On retire le premier sommet de P
        P=setdiff(P,P[length(P)]);
        i=P[length(P)];
      }
    }
  }
  return(FALSE);
}
```

Dans le cas où la matrice ne présente pas de cycle, la fonction retourne « FALSE » et à l'inverse, elle retourne « TRUE »

Exemple sur test_cycle sur la matrice B qui est la matrice non pondérée de A:

```
# 0 est associé à la valeur FALSE et tous les autres nombres à TRUE.
# Un "et logique" (respectivement un "ou logique") appliqué membre à membre sur la matrice A avec un TRUE (respectivement FALSE) nous donne donc un
une matrice qui contient FALSE lorque l'élement est égal à 0 et TRUE sinon.
#On applique as.numeric afin de convertir dans la matrice résultante B les TRUE en 1 et les FALSE en 0.
#B est ainsi la matrice d'adjacence non pondérée associée à A.
B=matrix(as.numeric(A & matrix(data=TRUE,nrow(A),ncol(A))),nrow(A),ncol(A))

print(B)

test_cycle(B,1)
```

On obtient ceci en résultat :

```
      [,1] [,2] [,3] [,4] [,5]
[1,]    0    1    1    1    1
[2,]    1    0    1    1    1
[3,]    1    1    0    1    0
[4,]    1    1    1    0    0
[5,]    1    1    0    0    0
[1] TRUE
```

Il y a donc un cycle, la fonction retourne TRUE.

On modifie B afin de retirer l'arrête entre les sommets (1,5) :

```
#Suppression de l'arrête entre les sommets (1,5)
B[1,5]=0
B[5,1]=0

print(B)

test_cycle(B,5)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    1    1    1    0
[2,]    1    0    1    1    1
[3,]    1    1    0    1    0
[4,]    1    1    1    0    0
[5,]    0    1    0    0    0
[1] FALSE
```

Avant de s'attaquer à la fonction de Kruskal, nous avons réalisé une dernière fonction « matriceAdjacence » qui permet à partir de la liste des arêtes de créer la matrice d'adjacence associée.

```
#Matrice qui va donner la matrice d'adjacence associée à la liste d'arêtes U et de l'arête Uk
matriceAdjacence=function(U,Uk){

  #Chaque arrête contient 3 éléments
  nRowU=length(U)/3;

  #On récupère le sommets maximum liées aux arêtes afin de connaître la dimension de la matrice
  #résultante puis on construit cette matrice
  maxUk=max(Uk[1],Uk[2]);
  if(nRowU>0){
    if(nRowU==1){
      maxU=max(U[1],U[2]);
      max=max(maxU,maxUk);
      res=matrix(data = 0, nrow = max, ncol = max)
      res[U[1],U[2]]=U[3];
      res[U[2],U[1]]=U[3];
    }
    else{
      maxU=max(union(U[,1],U[,2]));
      max=max(maxU,maxUk);
      res=matrix(data = 0, nrow = max, ncol = max)
      for(i in 1:nRowU){
       res[U[i,][1],U[i,][2]]=U[i,][3];
       res[U[i,][2],U[i,][1]]=U[i,][3];
      }
    }

  }
  #Ajout des arêtes de Uk
  res[Uk[1],Uk[2]]=Uk[3];
  res[Uk[2],Uk[1]]=Uk[3];

  #On retourne la matrice d'adjacence
  return(res);
}
```

Nous nous sommes ensuite penchés sur l'algorithme de Kruskal en lui-même, voici notre code :

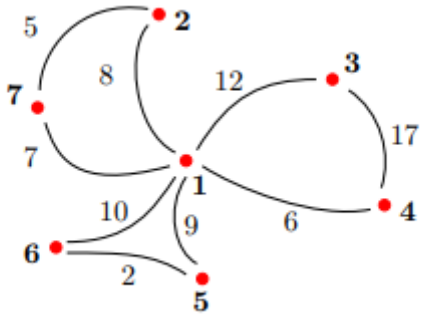
```
kruskal=function(A){
  #Récupération de la matrice des sommets et arêtes
  U=liste_aretes(A);
  nRowU=length(U)/3;
  #Tri par ordre coissant du poids des arêtes
  U=U[order(U[,3],decreasing = FALSE),];
  V=U[1,];
  k=1;
  #Une arête contient 3 éléments
  nrow_V=length(V)/3;


  while(nrow_V!=nrow(A)-1){
    k=k+1;
    s=U[k,1]
    #k est limité afin de ne pas être en dehors du domaine de définition des indices de la matrice U
    #On test si il existe un cycle dans V union Uk
    while(k<nRowU-1 && test_cycle(matriceAdjacence(V,U[k,]),U[k,1])){
      k=k+1;
    }
    V=rbind(V,U[k,]);
    nrow_V=length(V)/3;
  }
  #On retourne l'arbre partiel de poids minimum (liste d'arêtes)
  return(V);
}
```

Pour illustrer le code, voici un exemple d'utilisation le graphe de l'exercice 4.1:

```
kruskal(A)
```

```
   [,1] [,2] [,3]
V     1    4    1
      2    4    2
      2    3    4
      2    5    8
```

Voici le 2ème exemple sur ce graphe :



```
256 #Q3
257 M=cbind(c(0,8,12,6,9,10,7),c(8,0,0,0,0,0,5),c(12,0,0,17,0,0,0),c(6,0,17,0,0,0,0),c(9,0,0,0,0,2,0),
    c(10,0,0,0,2,0,0),c(7,5,0,0,0,0,0));
258 kruskal(M);
259 ```
```

```
  [,1] [,2] [,3]
V    5    6    2
     2    7    5
     1    4    6
     1    7    7
     1    5    9
     1    3   12
```

On obtient la liste des arêtes suivante, on remarque qu'elle recouvre bien tous les sommets avec des arêtes de poids minimales.

Algorithme n°2 : Ford-Bellman

Suite à l'algorithme de Kruskal, nous avons réfléchis à celui de Ford-Bellman. Cet algorithme permet de trouver le plus court chemin d'un sommet « s » donné vers tous les autres sommets. Il est utilisable pour tout graphe orienté, voici le pseudo code associé :

Input : $G = [X, U], s$
$\pi(s) \leftarrow 0$
Pour tout $i \in \{1, 2, \ldots, N\} \setminus \{s\}$ **faire**
    $\pi(i) \leftarrow +\infty$
Fin Pour
Répéter
    Pour tout $i \in \{1, 2, \ldots, N\} \setminus \{s\}$ **faire**
        $\pi(i) \leftarrow \min(\pi(i), \min_{j \in \Gamma^{-1}(i)} \pi(j) + l_{ji})$;
    Fin Pour
Tant que une des valeurs $\pi(i)$ change dans la boucle Pour
Output : π

Nous avons donc réalisé cet algorithme sous R, voici ce que nous avons fait :

```r
Ford_Bellman<-function(A,s){
  # Initialisation des variables temporaires
  d<-rbind()
  verif<-rbind()
  π<-c()

  # Initialisation de la variable "π" au rang "s", la valeur 0
  π[s]<-0

  # Création du vecteur comportant tous les sommets de "A"
  d<-c(1:nrow(A))

  # Création du vecteur comportant tous les sommets de "A" sauf "s"
  d2 <- setdiff(d,s)

  #Initialisation de π pour les rangs différents de "s"
  for (i in d2){
    π[i]<-Inf
  }

  # Initialisation de test
  preced<-π

  # Début du repeat
  repeat{

    # Initialisation de verif
    verif<-TRUE

    # Boucle qui parcours les sommets de "A" sans "s"
    for (i in d2){

      # Boucle qui parcours les sommets de "A"
      for (j in d){

        # Verification de l'antécédent
        if (A[j,i] != 0){
          π[i]<-min(c(π[i],π[j]+A[j,i]))
        }
      }
```

```
        # Test
        if (π[i]!=preced[i]) {
            verif<-FALSE
        }

        # "preced" prend la valeur de "π"
        preced[i]<-π[i]
      }

      # Sortie du repeat
      if(verif==TRUE){
        print(π)
        break
      }


    }
}
```

Exemple d'exécution avec la matrice A ci-dessous et le sommet 7 :

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

```
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))
Ford_Bellman(A,7)
```

```
[1]  7 12 -1 16  3 -3  0
```

La sortie correspond bien aux distances minimales entre le sommet « 7 » et les autres.

Algorithme n°3 : Ford-Fulkerson

L'algorithme numéro 3 est celui de Ford-Fulkerson, il permet de déterminer le flot de valeur maximale à partir d'une matrice d'adjacence associée à un graphe pondéré orienté.

Il peut être utile pour déterminer par exemple le nombre maximum de voitures allant de A vers B en passant par n'importe quel chemin. On modéliserait les routes par des arêtes ayant un poids variant selon le flux de voiture qu'elle peut faire passer.

Voici le pseudo code de cet algorithme :

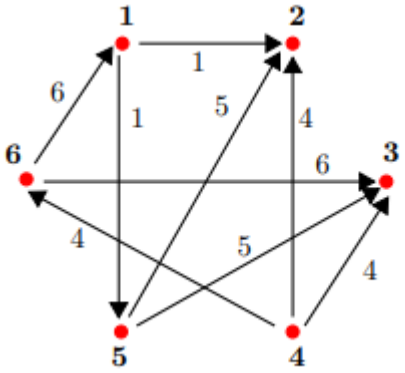
Input : $G = [X, U, C]$, φ un flot réalisable
1 $m_s \leftarrow (\infty, +)$ et $S = \{s\}$
2 **Tant que** $\exists (j \in \overline{S}, i \in S) : (c_{ij} - \varphi_{ij} > 0) \vee (\varphi_{ji} > 0)$ **faire**
3 **Si** $c_{ij} - \varphi_{ij} > 0$ **faire**
4 $m_j \leftarrow (i, \alpha_j, +)$ avec $\alpha_j = \min\{\alpha_i, c_{ij} - \varphi_{ij}\}$
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**

Code R de l'algorithme :

```
Ford_Fulkerson=function(X,A,s,p){

  #Initialisation de la matrice m que l'on initialise avec +Inf
  m=matrix(data = Inf, nrow = length(X), ncol =3)
  v=0;

  #Initialisation du flot P
  P=matrix(0,nrow=length(X),ncol=length(X));
  repeat{

    #ms<-(Inf,+)
    m[s,3]="+";
    #S={s}
    S=c(s);
    repeat{
       #calcul des couples (i,j) appartenant à (S,Sb) vérifiant (c_i,j - gamma_i,j >0) U (gamma_j,i>0)
      Sb=setdiff(X,S);
      R1=A-P>0;
      R2=t(P)>0;
      C=R1|R2;
      couple_i_j=which(matrix(C[S,Sb]==TRUE,nrow=length(S),ncol=length(Sb)),arr.ind=TRUE);

      if(length(couple_i_j)<=0){
        break;
      }

      i=S[couple_i_j[1,1]];
      j=Sb[couple_i_j[1,2]];
      #R1 étant la matrice binaire de c_i,j - gamma_i,j > 0
      if(R1[i,j]){
        #mj=(i,alpha_j,'+')
        m[j,1]=i;
        m[j,2]=min(m[i,2],(A-P)[i,j]);
        m[j,3]="+";
      }
      #et R2 celle de gamma_j,i > 0
      else if(R2[i,j]){
        #mj=(i,alpha_j,'-')
        m[j,1]=i;
        m[j,2]=min(m[i,2],(t(P))[i,j]);
        m[j,3]="-";
      }
      #Concaténation de l'ensemble S avec l'élément j
      S=append(S,j);
```

```
      if(p==j){
        #V(gamma)=V(gamma)+alpha_p
        v=as.numeric(m[p,2])+v;
        #Ce break nous fait passer à la ligne 14 dans le pseudo-code
        break;
      }
    }
    #Si p appartient à l'ensemble S
    if(is.element(p,S)){
      while(j!=s){
          if(m[j,3]=="+"){

            P[as.numeric(m[j,1]),j]= P[as.numeric(m[j,1]),j]+as.numeric(m[p,2]);

          }
          else if (m[j,3]=="-"){
            P[j,m[j,1]]= P[j,m[j,1]]-m[p,1];
          }
          #j=m_j1
          j=as.numeric(m[j,1]);

      }
    }
    else{
      #Flot de valeur maximale
      return(v);
    }
  }
}
```

Exemple sur ce graphe orienté pondéré en prenant le sommet 4 pour source et le sommet 2 comme puit:



La matrice d'adjacence associée à ce graphe étant :

	V1	V2	V3	V4	V5	V6
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	4
5	0	5	5	0	0	0
6	6	0	6	0	0	0

On peut exécuter notre code R grâce à ces lignes :

```
X=1:6
A=cbind(c(0,0,0,0,0,6),c(1,0,0,4,5,0),c(0,0,0,0,5,6),c(0,0,0,0,0,0),c(1,0,0,0,0,0),c(0,0,0,4,0,0))

Ford_Fulkerson(X,A,4,2)
```

Le résultat est :

```
[1] 6
```

Le flot de valeur maximal allant du puit (Sommet 4) au puit (Sommet 2) a donc pour valeur 6

Conclusion

Ce projet nous a permis d'implémenter des algorithmes de graphes. Les trois algorithmes ont été rapidement vus en TD, et le fait de les implémenter sous R a été très utile pour être d'avantage à l'aise avec l'outil R.

Les véritables difficultés ont été de transformer le pseudo code en code R. Nous nous sommes rendu compte qu'il fallait souvent préparer des fonctions en amont afin de rendre le code lisible.

Si l'on devait améliorer notre code, il faudrait se poser les bonnes questions afin d'améliorer la complexité de nos algorithmes, et par conséquent, les rendre plus optimisés.