Lebreton Noé
Moissonnier Rayan

2020-2021

Optimisation et recherche opérationnelle : Théorie des graphes

Table des matières

Introduction

Ce dossier entre dans le cadre du contrôle continu de connaissances dans le contexte du cours d'optimisation et de recherches opérationnelles et plus précisément si la théorie des graphes.

Dans ce dossier, nous présenterons l'objet de l'algorithme, son intérêt, son pseudo code, le code R de son implémentation, ainsi qu'un exemple d'utilisation de celui-ci, pour chaque algorithme de ce dossier.

Les algorithmes auxquels nous nous sommes intéressés ici sont les suivants :

- L'algorithme de détection de l'arbre recouvrant de poids minimal par Prim
- L'algorithme de calcul des plus courts chemins par Ford-Bellman
- L'algorithme de détermination d'un flot maximal dans une réseau avec capacités par Ford-Fulkerson

Nous allons donc les présenter par la suite.

Algorithme 1 : Détection de l'arbre recouvrant de poids minimum

Objet de l'algorithme

L'algorithme de Prim est une méthode distincte de celle de Kruskal qui permet de déterminer un arbre partiel de poids minimal. Il est utilisé dans le cadre d'un graphe connexe valué et non orienté.

Un arbre est un graphe non orienté simple, connexe et sans cycle. Et l'arbre partiel de poids minimum est un arbre qui connecte tous les sommets ensemble et dont la somme des poids des arretes est minimale

Il y a de nombreuses applications possibles, tel que des problèmes de distribution ou de télécommunication.

Pseudo code

Nous rappelons ci-dessous le pseudo code de l'algorithme permettant de détecter l'arbre recouvrant de poids minimum et que cet algorithme s'applique aux graphes connexes valués et non orientés.

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 : $G' = [X', U']$

Code R

```
Prim = function(X,A){
  #determine un arbre partiel de poids minimal
  #INPUT
  #X est l'ensemble des sommets
  #A est la matrice d'adjacence
  #OUTPUT
  #G_prim qui est l'arbre pariel de poids minimum(avec ses sommets X_prim, ses aretes U_prim et son poids)
  ###initialisation des variables###
  #A_prim contient A, cette variable est créée pour ne pas impacter la matrice initiale
  #X_prim correspond à un sommet de X pris au hasard...ou pas : X_prim = 3
  #X_prim_bar est le complementaire de X_prim par rapport à X
  #U_prim correspond à l'ensemble des aretes du sous graphe G_prim
  A_prim = A
  X_prim = sample(X,1)
  print("Le sommet de départ est : ")
  print(X_prim)
  X_prim_bar = setdiff(X,X_prim)
  U_prim = list()
  poids_graphe_partiel = 0
  indice = 1
  #tant qu'il existe des sommets de X qui ne sont pas dans X_prim
  while (length(X_prim) != length(X)) {
    #creation de data_sort contenant toutes les aretes allant de X_prim à d'autres sommets qui sont dans le complémetaire de X_prim par rapport à X ainsi que leurs poids
    ind = 1
    data_sort = as.data.frame(setNames(replicate(3,numeric(0), simplify = F),c("i","j","poids")))
    for (i in X_prim) {
      lignes = which(A_prim[i,]>0)
      for (j in lignes){
        if(is.element(j,X_prim)==F){
          data_sort[ind,] = c(i,j,A_prim[i,j])
          ind = ind + 1
        }
      }
    }
    #on tri data_sort sur la premiere colonne, càd les sommets présents dans X_prim, cela est utile pour le choix de l'arete de poids minimum
    #okay pour ce tri
    data_sort = data_sort[order(data_sort[,1],decreasing=F), ]
    res = which.min(data_sort[,3])
    poids_graphe_partiel = poids_graphe_partiel + data_sort[res,3]
    X_prim = c(X_prim,data_sort[res,2])
    #il est possible d'avoir des doublons dans X_prim lorqu'on trouve plusieurs aretes partant d'un sommet deja dans X_prim
```

```
    doublons = which(duplicated(X_prim))
    if (length(doublons)>0){
      X_prim = X_prim[-doublons]
    }
    X_prim_bar = setdiff(X,X_prim)
    #insertion de l'arete dans l'element U_prim contenant toutes les aretes du sous graphe partiel de poids minimal
    U_prim[[indice]] = c(data_sort[res,1],data_sort[res,2])
    #on met à 0 l'arete que l'on vient de mettre dans U_prim ainsi que son inserse (ex : (3,6) et (6,3)), cela permet d'exclure cette arete dans le choix de la prohaine itération
    A_prim[data_sort[res,1],data_sort[res,2]]=0
    A_prim[data_sort[res,2],data_sort[res,1]]=0
    indice = indice + 1
  }
  print("L'ensemble des sommets du graphe partiel de poids minimum :")
  print(sort(X_prim))
  print("Le poids du graphe partiel de poids minimum :")
  print(poids_graphe_partiel)
  print("L'ensemble des aretes du graphe partiel de poids minimum :")
  return(U_prim)
 }
```

Illustration sur un exemple

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

Nous allons donc essayer le code sur cette matrice :

```
X = 1:7
A =
cbind(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)
```

En exécutant le code, cela nous donne :

```
[1] "Le sommet de départ est : "
```

```
[1] 3
[1] "L'ensemble des sommets du graphe partiel de poids minimum :"
[1] 1 2 3 4 5 6 7
[1] "Le poids du graphe partiel de poids minimum :"
[1] 19
[1] "L'ensemble des arêtes du graphe partiel de poids minimum :"
[[1]]
[1] 3 6
[[2]]
[1] 6 7
[[3]]
[1] 7 5
[[4]]
[1] 5 2
[[5]]
[1] 2 4
[[6]]
[1] 2 1
```

Ici, nous obtenons l'ensemble des sommets de l'arbre partiel de poids minimum qui est le même que celui du graphe complet. On obtient également le poids du graphe partiel de poids minimum qui est ici 19, il est important de préciser que peu importe le sommet de départ le poids sera toujours 19 pour cet exemple. Enfin nous voyons l'ensemble des arêtes constituants l'arbre partiel de poids minimum.

Algorithme 2 : Calcul des plus courts chemins

Objet de l'algorithme

L'algorithme de Ford-Bellman est un algorithme qui calcule des plus courts chemin depuis un sommet source donné dans un graphe pondéré. Contrairement à l'algorithme de Dijkstra, il autorise la présence de certains arcs de poids négatif et permet de détecter l'existence d'un circuit absorbant, c'est à dire de poids strictement négatif accessible depuis la source. Une application serait par exemple la determination du cheminement des messages à travers le protocole de routahe RIP.

Pseudo Code

Input : $G = [X, U], s$
1 $\pi(s) \leftarrow 0$
2 **Pour tout** $i \in \{1, 2, \ldots, N\} \setminus \{s\}$ **faire**
3 $\pi(i) \leftarrow +\infty$
4 **Fin Pour**
5 **Répéter**
6 **Pour tout** $i \in \{1, 2, \ldots, N\} \setminus \{s\}$ **faire**
7 $\pi(i) \leftarrow \min(\pi(i), \min_{j \in \Gamma^{-1}(i)} \pi(j) + l_{ji})$;
8 **Fin Pour**
9 **Tant que** une des valeurs $\pi(i)$ change dans la boucle Pour
10 **Output :** π

Code R

```
ford_bellman = function(X,A,s){
  #detremine les longueurs des plus courts chemins entre s et tous les autres sommets de X
  #INPUT
  #X est l'ensemble des sommets
  #A est la matrice d'adjacence
  #s un sommet de X
  #OUTPUT
  #pi  qui correspond aux  les longueurs des plus courts chemins entre s et tous les autres sommets de X

  #initialisation des variables :
  #pi contient + inf pour touts les sommets sauf s qui prend comme valeur 0
  #Sb contient l'ensemble des sommets de X sauf s
  #pi_bis est initialisé à 0 pour toutes les valeurs de X, celui ci va permettre à observer si le vecteur pi connait une variation entre les itérations
  pi = rep(Inf,length(X))
  pi[s] = 0
  Sb = setdiff(X,s)
  pi_bis = rep(0,length(X))
  #tant qu'une des valeurs de pi hange
  while(length(which((pi!=pi_bis)!=0))>0){
   pi_bis=pi
   #pour tout les elements de Sb
   for (i in Sb){
    #on conserve dans indice ceux qui sont differents de 0
    indice = which(A[,i]!=0)
    #pour tous ces elements on calcul le minimum entre pi[i] et (pi[j]+la longueur entre j et i) ou j est un predecessur de i
    for (j in indice){
     pi[i] = min(pi[i],pi[j]+A[j,i])
    }
```

```
 }
}
print("Les plus courts chemins allant de ")
print(s)
print(" vers les autres sommets sommets du graphe sont de longueurs : ")
return(pi)}
```

Illustration sur un exemple

Premier exemple :

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

Nous allons donc essayer le code sur cette matrice :

```
X=1:7
s=2
A=cbind(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))
ford_bellman(X,A,s)
```

Pour cet exemple nous avons pris comme sommet de départ 2, dans le cas de notre exemple les conditions d'existence de plus courts chemins ne sont pas respectée car dans ce graphe il existe un circuit absorbant. Pour etre legerement plus clair, dans notre graphe nous avons un chemin allant de i à j comprenant un circuit de longueur negative, c'est à dire qu'a chaque nouvelle iteration l'algorithme trouve un plus court chemin allant de i à j. Concerant l'execution de la fonction avec ce graphe, la fonction boucle sans s'arreter car elle ne rempli pas les conditions d'arrets de cet algorithme.

Deuxième exemple :

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

Nous allons donc essayer le code sur cette matrice :

```
X=1:7
s=7
A=cbind(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(0,0,0,0,0,3,-3,0))
ford_bellman(X,A,s)
```

En executant le code, celà nous donné :

```
[1] "Les plus courts chemins allant de "
[1] 7
[1] " vers les autres sommets sommets du graphe sont de longueurs : "
[1]  7 12 -1 16  3 -3  0
```

Nous obtenons donc les plus courts chemins allant de 7 vers les autres sommets du graphe.

Algorithme 3 : Détermination d'un flot maximal dans un réseau avec capacités

Objet de l'algorithme

L'algorithme de Ford-Fulkerson est un algorithme de determination d'un flot maximum dans un réseau avec capacité. Le flot represente la circulation de l'entrée vers la sortie.
Les applications sont nombreuses : Problèmes routiers, ferroviaires, flux migratoires...

Pseudo Code

Nous rappelons ci-dessous le pseudo code de l'algorithme permettant de détecter l'arbre recouvrant de poids minimum et que cet algorithme s'applique aux graphes connexes valués et non orientés.

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**

Code R

```
   ford_fulkerson = function(X,A,s,p){
    #INPUT
    #X est l'ensemble des sommets
    #A est la matrice d'adjacence du réseau avec capacités
    #s un sommet source
    #p un sommet puit
    #OUTPUT
    #P est le flot de valeur maximale
    #val_flot est la valeur du flot maximale
    #init
    #P une matrice de meme taille que X contenant uniquement des zeros
    #m un data frame de taille n (avec n le nombre de sommets) contenant l'ensemble des
marques
    P=matrix(0,nrow=length(X),ncol=length(X))
    val_flot=0
    m=as.data.frame(setNames(replicate(3,numeric(0), simplify = F),c("1","2","3")))
    for (v in X) {
     m[v,] = c(0,0,0)
    }
    #premier appel de la fontion marquage_sommets
    res = marquage_sommets(X,A,s,p,P,m,val_flot)
    #tant qu'il est possible d'accroitre de la valeur du flot
    while(length(res) != 1){
     #si p appartient à S (res[[8]]=S)
     if (is.element(p,res[[8]])==T){
      j=p
      #tant que l'on est pas remonter à la source, on modifie le flot P (res[[5]]=P) en fontion
des marques "+" ou "-"(res[[6]]=m)
      while (j != s) {
       if (res[[6]][j,3]=="+"){
        res[[5]][as.numeric(res[[6]][j,1]),j] = res[[5]][as.numeric(res[[6]][j,1]),j]+
as.numeric(res[[6]][p,2])
       }else{
        if (res[[6]][j,3]=="-"){
```

```
            res[[5]][j,as.numeric(res[[6]][j,1])] = res[[5]][j,as.numeric(res[[6]][j,1])]+
as.numeric(res[[6]][p,2])
           }
          }
          j = as.numeric(res[[6]][j,1])
         }
         #on rappel la fontion permettant de marquer les sommets avec le nouveau flot
         res = marquage_sommets(res[[1]],res[[2]],res[[3]],res[[4]],res[[5]],res[[6]],res[[7]])
       }
      }
      return(res)
    }
    marquage_sommets = function(X,A,s,p,P,m,val_flot){
      #on initialise la marque de la source,S avec seulement la source et Sb avec la totalité des
sommets sans la source
      m[s,] = c("vide",Inf,"+")
      S = c(s)
      Sb = setdiff(X,S)
      #matrice avec c_ij-phi_ij > 0
      R1=A-P>0
      #matrice phi_ji > 0
      R2=t(P)>0
      #combinaison des 2 conditions
      C=R1|R2
      #temp contient la liste des aretes qui valident la combinaison des 2 conditions C
      temp = which(matrix(C[S,Sb]==TRUE,nrow=length(S),ncol=length(Sb)),arr.ind=TRUE)
      while (length(temp)!=0) {
        #test_S regarde si il y a des doublons dans la premiere colonne de temp,ind_S
contiendra alors le ou les sommets de départ
        test_S = which(duplicated(temp[,1]))
        if(length(test_S)>0){
          ind_S = temp[-test_S,1]
          ind_S = S[ind_S]
        }else{
          ind_S = temp[,1]
          ind_S = S[ind_S]
        }
        #ind_S = which(duplicated(temp[,1]))
        #ind_S = temp[-ind_S,1]
        #test_Sb regarde si il y a des doublons dans la deuxieme colonne de temp, ind_Sb
contiendra alors le ou les sommets d'arrivée
        test_Sb = which(duplicated(temp[,2]))
        if(length(test_Sb)>0){
          ind_Sb = temp[-test_Sb,2]
          ind_Sb = Sb[ind_Sb]
        }else{
          ind_Sb = temp[,2]
```

```
      ind_Sb = Sb[ind_Sb]
    }
    #pour chaque combinaisons d'aretes (i,j) qui valident les conditions necessaires(C)
    for (i in ind_S) {
      for (j in ind_Sb) {
        #si l'arete (i,j) valide c_ij-phi_ij > 0 alors on met à jour la deuxieme valeur du
marquage,on ajoute j à S et on enleve j à Sb
        if(R1[i,j]==T){
          alpha = min(m[i,2],A[i,j]-P[i,j])
          m[j,] = c(i,alpha,"+")
          S = c(S,j)
          Sb = setdiff(X,S)
          test_val = which(ind_Sb==j)
          ind_Sb =ind_Sb[-test_val]
        }else{
          #si l'arete (i,j) valide phi_ji > 0 alors on met à jour la deuxieme valeur du
marquage,on ajoute j à S et on enleve j à Sb
          if(R2[i,j]==T){
            alpha = min(m[i,2],P[j,i])
            m[j,]=c(i,alpha,"-")
            S = c(S,j)
            Sb = setdiff(X,S)
            test_val = which(ind_Sb==j)
            ind_Sb =ind_Sb[-test_val]
          }
        }
      }
      #on met à jour temp avec les nouvelles valeurs de S et Sb
      temp = which(matrix(C[S,Sb]==TRUE,nrow=length(S),ncol=length(Sb)),arr.ind=TRUE)
    }
    #si on atteint le puits on met à jour la valeur du flot et on sort de la boucle
    if (is.element(p,S)==T){
      val_flot = val_flot + as.numeric(m[p,2])
      break()
    }
  }
  #si on atteint le puits on retourne les nouveaux elements, sinon on affiche le flot ainsi
que ca valeur
  if (is.element(p,S)==T){
    res = list(X,A,s,p,P,m,val_flot,S)
    return(res)
  }else{
    print("Le flot de valeur maximale est : ")
    print(P)
    print("La valeur du flot maximale est : ")
    return(val_flot)
  }
```

```
}
```

Illustration sur un exemple

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

Nous allons donc essayer le code sur cette matrice :

```
X = 1:6
A=cbind(c(0,0,0,0,0,6),c(1,0,0,4,5,0),c(0,0,0,4,5,6),c(0,0,0,0,0,0),c(1,0,0,0,0,0),c(0,0,0,4,0,0)
)
s=4
p=2
ford_fulkerson(X,A,s,p)
```

En exécutant le code, cela nous donne :

```
[1] "Le flot de valeur maximale est : "
     [,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] "La valeur du flot maximale est : "
[1] 6
```

Nous voyons donc que dans cet exemple, la valeur du flot maximale est de 6.

Conclusion

Lors de ce projet, nous avons pu nous intéresser à trois algorithmes de graphes que nous avons implémentés sur R.

Cela nous a permis de nous entraîner à implémenter divers algorithmes sur R ainsi que nous montrer des exemples d'utilisation de ceux-ci grâce à des exemples. De plus le fait de les avoir

étudiés et implémentés nous a permis de mieux comprendre leur fonctionnement et utilisation.

Les principales difficultés que nous avons rencontrés ont été de trouver une solution pour mettre en fonction le dernier algorithme, pour cela nous avons donc décidé de séparer cet algorithme en deux parties ce qui nous a permis de pallier à la fonction ”goto” qui n’est pas disponible sur R. De plus le fait d’avoir séparer cet algorithme en deux parties montre plus explicitement la manière dont il fonctionne c’est à dire le coté consacré au marquage de sommets et celui à la mise à jour du flot.

Globalement nous sommes satisfait de notre travail car la totalité des algorithmes fonctionnent sur les exemples du sujet mais aussi sur des exemples annexes trouvés sur internet.