Optimisation - Recherche Opérationnelle
Dossier

DORONZO Franck, DUDOIT Romain
M1 Informatique - Université Lumière Lyon 2

20 décembre 2020

Table des matières

1 Introduction

Nous avons rédigé ce rapport dans le cadre du contrôle continu de l'UE optimisation et recherche opérationnelle. Trois algorithmes seront abordés : l'algorithme de Prim, l'algorithme de Ford-Bellman et l'algorithme de Ford-Fulkerson.

Pour chacun d'eux la présentation sera identique. Nous commencerons par présenter l'objet de l'algorithme en expliquant brièvement ce qu'il permet d'obtenir, son fonctionnement et des exemples d'applications dans la vie courante. Vient ensuite le pseudo-code de l'algorithme qui décrira son fonctionnement. En troisième partie nous présenterons leur implémentation que nous avons réalisé en R, où l'ensemble du code est accompagné de commentaires. Pour terminer la dernière partie consistera à appliquer notre code sur un exemple donné afin de constater le bon fonctionnement de celui-ci.

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

On rappel qu'un arbre est un graphe non orienté (matrice d'adjacence symétrique) connexe (chaque couple de sommets peut être relié par une chaîne) et acyclique.

2.1 Objet de l'algorithme

L'algorithme de Prim est un algorithme qui à pour but de trouver, dans un graphe non orienté connexe et pondéré, un arbre couvrant de poids minimal, c'est-à-dire un arbre qui connecte tous les sommets du graphe et dont la somme des poids des arêtes est minimale (inférieure ou égale à celle de tous les autres arbres couvrants du graphe).

L'algorithme consiste à choisir arbitrairement un sommet et de faire croître un arbre à partir de ce dernier. A chaque étape, on ajoute une arête de poids minimum ayant exactement une extrémité dans l'arbre en cours de construction (si on choisit une arête de poids minimum dont les deux extrémités sont déjà dans l'arbre, son ajout créerait un cycle et donc le résultat final ne serait plus un arbre). L'algorithme se termine lorsque tous les sommets du graphe sont contenus dans l'arbre.

Cet algorithme peut avoir beaucoup d'utilité dans les cas pratiques. Prenons par exemple une résidence privée où il faudrait installer un réseau électrique sur l'ensemble de la résidence. Il faudrait installer des lignes de telle sorte que toutes les maisons soient connectées et que le coût de la configuration soit minimal, l'algorithme de Prim est dans ce cas là très utile.

2.2 Pseudo-code

Le pseudo-code de l'algorithme de Prim prend en entrée un ensemble de sommets X et U l'ensemble des arêtes pondérée. Il donnera en sortie les arêtes qui constituent l'arbre couvrant de poids minimal.

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

2.3 Code R

Notre code R va prendre en entrée une matrice d'adjacence pondérée A qui représente les arêtes d'un graphe non orienté connexe et X l'ensemble des sommets de ce graphe.

```
Prim=function(A,X=rep(1:nrow(A))){

  # choix d'un sommet au hasard
  Xp <- sample(X,1)

  # lignes et colonnes des poids positifs
  pweight=which(A>0,arr.ind=T)

  # matrice U  = arêtes j-k de poids minimal + poids associé
  U <- matrix(0,length(X)-1,3)

  # compteur qui va servir à modifier la matrice U
  cpt = 1

  while (all(intersect(X,Xp)==X)==FALSE){
  # génère des warnings a cause des tailles différentes entre Xp et X,
  # ne pas en tenir compte

    Xb <- setdiff(X,Xp)


    # sommets J,K et P les poids de la matrice d'adjacence
    J = K = P = c()
```

```
    # ind =  sommets qui sont dans Xp et les sommets qui sont dans Xb
    ind=which((pweight[,1]%in%Xp)&(pweight[,2]%in%Xb))

    for (i in ind){
      j <- pweight[i,1] # sommet j qui vérifie j appartient à Xp
      k <- pweight[i,2] # sommet k qui vérifie k appartient à Xb
      J <- c(J,j)       # ensemble des sommets j
      K <- c(K,k)       # ensemble des sommets k
      P <- c(P,A[j,k])  # ensemble des poids
      poids_min=min(P)  # on récupère le poids minimal
      # on prend le 1er indice du poids minimal trouvé parmis les poids P
      ind_poids_min= which.min(P)
    }
    Xp <- c(Xp,K[ind_poids_min])
    U[cpt,] <- c(J[ind_poids_min],K[ind_poids_min],poids_min)
    cpt=cpt+1
  }
  return(U)
}
```

2.4 Illustration sur un exemple

Nous allons tester notre fonction sur le graphe représenté par la matrice d'adjacence pondérée suivante :

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

On vérifie bien qu'elle est symétrique et on pourra s'assurer que le graphe est bien connexe en le dessinant.

```
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))
res=Prim(A)
print(res)

##      [,1] [,2] [,3]
## [1,]    6    3    2
## [2,]    6    7    3
```

```
## [3,]    7    5    3
## [4,]    5    2    2
## [5,]    2    4    4
## [6,]    2    1    5
```

Le résultat de sortie est la matrice d'arêtes qui constituent l'arbre couvrant de poids minimal. La 1re colonne représente un sommet i et la 2e colonne le sommet j. La 3e colonne représente le poids de l'arête entre les sommets i et j.

On constate que le sommet de départ qui a été choisi par l'algorithme est 6. Il va donc par la suite choisir l'arête de poids minimal qui part de 6 à savoir ici 6-3 et qui constitue la première arête de l'arbre. L'opération se réitère ainsi jusqu'à ce que l'ensemble des sommets soient parcourus.

Pour afficher le poids de l'arbre, il suffira de faire la somme des éléments de la 3e colonne de la matrice res :

```
print(paste("L'arbre couvrant de poids minimal à un poids de",sum(res[,3])))
```

```
## [1] "L'arbre couvrant de poids minimal à un poids de 19"
```

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

3.1 Objet de l'algorithme

L'algorithme de Ford-Bellman est un algorithme qui consiste à déterminer le plus court chemin reliant un sommet s à tous les autres sommets d'un graphe orienté valué dont les longueurs sont quelconques (positives ou négatives). La longueur d'un chemin étant la somme des longueurs des arcs le constituant. Le problème des plus courts chemins à partir d'une sommet s n'a de solution que s'il n'existe pas de circuit absorbant atteignable, c'est-à-dire un circuit de poids total strictement négatif.

Le principe est le suivant : On commence par étiqueté chacun des sommets à $+\infty$ à l'exception du sommet de départ s qui sera étiqueté à 0. Ces étiquettes représentent la dernière longueur connue à un instant t du plus court chemin entre s et un sommet en question. Ensuite il va y avoir un certain nombre d'itérations dans lesquelles l'étiquette de chaque sommet autre que s va être passée en revue et être modifiée ou non selon les valeurs des étiquettes des prédécesseurs du sommet sur lequel on veut modifier l'étiquette. L'algorithme s'arrêtera lorsque les étiquettes auront cessées d'êtres modifiées. On se référa au pseudo-code pour comprendre plus en profondeur le fonctionnement de l'algorithme.

L'algorithme de Ford-Bellman est notamment utilisé dans les réseaux informatiques pour déterminer le cheminement des messages, à travers le protocole de routage RIP. (Cf.Wikipedia)

3.2 Pseudo-code de l'algorithme

L'algorithme prend en entrée l'ensemble de sommets noté X, l'ensemble des arcs U du graphe (que l'on représentera par une matrice d'adjacence lors de l'implémentation du code) ainsi qu'un sommet de départ noté s. Son pseudo-code est le suivant :

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

3.3 Code R

```
Bellman=function(X,A,s){
  k=c()
  # pi = vecteur qui représente les étiquettes de chaque sommet
  pi=rep(Inf,length(X))
  pi[s]=0

  # Tant qu'une étiquette du vecteur pi change
  while(setequal(k,pi)==FALSE){
    k=pi # on mémorise le dernier vecteur pi connu afin de pouvoir
         # le comparer au prochain qui aura été modifié ou non

    for (i in setdiff(X,s)){
      # mise à jour des étiquettes
      pi[i]=min(pi[i],(pi[which(A[,i]!=0)]+c(A[which(A[,i]!=0),i])))
    }
  }
  return(pi)
}
```

3.4 Illustration

Nous allons tester notre fonction sur un premier graphe représenté par la matrice d'adjacence pondérée suivante avec 5 comme sommet de départ :

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

```
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))
X=rep(1:nrow(A))

res=Bellman(X,A,5)

print(res)
```

Cependant, le code ci-dessus n'aboutira pas. En effet nous avons dû stopper l'exécution car le programme exécutait la boucle *while* à l'infini peut importe le sommet de départ. Pour comprendre cela, il faut revenir sur les conditions d'applications de Ford-Bellman. En effet ce graphe contient un circuit absorbant (2 au total) et par conséquent à partir du moment où l'algorithme rentre dedans, il va l'emprunter au tant de fois que la longueur du chemin va pouvoir être minimisé à savoir à l'infini.

Nous allons maintenant tester notre fonction sur un deuxième graphe représenté par la matrice d'adjacence pondérée suivante en prenant 7 comme sommet de départ.

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

```
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))
X=rep(1:nrow(A))

res=Bellman(X,A,7)

print(res)

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

Sur ce graphe l'algorithme de Ford-Belmman ne fonctionne qu'en partant du sommet 6 ou du sommet 7, car ici aussi en dehors de ces deux sommets l'algorithme est en présence d'un circuit absorbant ce qui fait que notre programme boucle à l'infini. Ici le résultat en sortie nous dit que en partant du sommet 7, il faut suivre un chemin de longueur 7 pour atteindre le sommet 1, un chemin de longueur 12 pour atteindre le sommet 2 ...

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

On rappelle qu'un réseau de flot est un graphe orienté où chaque arc possède une capacité et peut recevoir un flot ne pouvant excéder sa capacité.

4.1 Objet de l'algorithme

L'algorithme de Ford-Fulkerson est un algorithme qui permet la résolution du problème de flot maximum qui consiste à trouver dans un réseau de flot, un flot réalisable depuis une source unique vers un unique puits. Pour qu'un flot soit valide, il faut que la somme des flots atteignant un nœud soit égale à la somme des flots quittant ce nœud, sauf s'il s'agit d'une source (qui n'a pas de flot entrant) ou d'un puits (qui n'a pas de flot sortant).

Son principe est le suivant : on part d'un flot réalisable et à chaque itération on essaie d'accroître le flot jusqu'à ce que les contraintes de capacité ne soient plus respectées. On réalise cela en marquant certains sommet par un triplé qui rend compte de la situation dans l'itération courante.

L'algorithme de Ford-Fulkerson est par exemple utilisé dans certains pays pour établir un débit maximum dans le réseau de distribution d'eau potable.

4.2 Pseudo-code

Le pseudo-code suivant prend en entrée un ensemble de nœuds ou sommets X, les arcs U du réseau de flot dont les capacités sont représentées par la matrice C, une source s, un puit p et un flot réalisable φ.

Input : $G = [X, U, C], s, p, \varphi$
1 $m_s \leftarrow (\infty, +)$ et $S = \{s\}$
2 **Tant que** $\exists(j \in \bar{S}, i \in S) : (c_{ij} - \varphi_{ij} > 0) \vee (\varphi_{ij} > 0)$ **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** $\varphi_{ij} > 0$ **faire**
6 $m_j \leftarrow (i, \alpha_j, -)$ avec $\alpha_j = min\{\alpha_i, \varphi_{ij}\}$
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**

Si $m_j(3) = +$ **faire**
$\varphi_{m_j(1)j} \leftarrow \varphi_{m_j(1)j} + \alpha_p$
Sinon Si $m_j(3) = -$ **faire**
$\varphi_{jm_j(1)} \leftarrow \varphi_{jm_j(1)} + \alpha_p$
Fin Si
$j \leftarrow m_j(1)$
Fin Tant que
Aller en 1
Sinon faire
Output : $V(\varphi)$
Fin Si

4.3 Code R

Notre algorithme prend en entrée l'ensemble des sommets X d'un graphe, la matrice d'adjacence A associée qui représente la capacité maximale de chaque arc, le sommet source s, le puits p, la matrice phi (φ) qui est la matrice des flots courants dans chaque arc à un instant t. Cette matrice est donc initialisé à 0. Ainsi que v_phi ($V(\varphi)$) qui représente la valeur du flot sortant de la source, initialisée elle aussi à 0.

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


  C1 <- (A-phi)>0 # 1ère condition
  C2 <- t(phi)>0 # 2ème condition

  C=C1|C2 # matrice de booléens qui vérifie C1 ou C2

  # on commence par mettre dans S la source s
  S = c(s)

  # on associe à une liste m_s les triplés qui caractérisent la source
  m_s = list(0,Inf,"+")

  # on crée une liste m qui va contenir les listes des marquages permanents
  m=list()
  m[[s]]=m_s # ajout de m_s dans m en position s


  Sb = setdiff(X,S) # ensemble des sommets X qui ne sont pas dans S

  # indices des lignes et colonnes pour lesquels les valeurs
  # de la matrice C[S,Sb] sont à TRUE
  ind=which(matrix(C[S,Sb],nrow=length(S),ncol = length(Sb)),arr.ind = TRUE)
```

```r
S_Sb=cbind(S[ind[,1]],Sb[ind[,2]])
# ind[,1] : 1ère colonne de ind correspond aux sommets de S
# ind[,2] : 2ème colonne de ind, correspond aux sommets de Sb

# Tant que la matrice S_Sb n'est pas vide
while (all(is.na(S_Sb))!=TRUE) {

  # on prend le premier couple (i,j) de la matrice S_Sb qui vérifient C
  # (on aurait pu prendre n'importe quel couple de cette matrice)
  i <- S_Sb[1,1]
  j <- S_Sb[1,2]

  alpha_i <- m[[i]][[2]]

  if ((A[i,j]-phi[i,j])>0){
    alpha_j <- min(alpha_i,(A[i,j]-phi[i,j]))
    m[[j]] <- list(i,alpha_j,'+')
  }

  else if (phi[j,i]>0){
    alpha_j <- min(alpha_i,phi[j,i])
    m[[j]] <- list(i,alpha_j,'-')
  }

  # on met a jour S, Sb et S_Sb avant de rentrer à nouveau dans la boucle
  S=c(S,j)
  Sb = setdiff(X,S)
  ind=which(matrix(C[S,Sb],nrow=length(S),ncol=length(Sb)),arr.ind = TRUE)
  S_Sb=cbind(S[ind[,1]],Sb[ind[,2]])

  # Si j correspond au puit
  if (j==p){
    v_phi=v_phi+alpha_j
    break
  }
}

# si le puit se trouve dans S..., lignes exécutées si il y a eu un break
if (p %in% S){
# tant que j est différent de la source
  while(j!=s){

    if (m[[j]][3]=='+'){
      phi[unlist(m[[j]][1]),j] <- phi[unlist(m[[j]][1]),j]+alpha_j
```

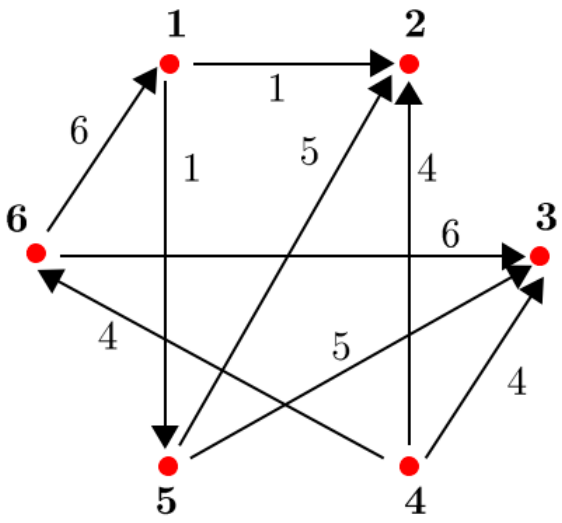
```
      }

      else if (m[[j]][3]=='-'){
        phi[j,unlist(m[[j]][1])] <- phi[j,(unlist(m[[j]][1]))]-alpha_j
      }

      j<- unlist(m[[j]][1])
    }
      # Aller en 1 du pseudo-code -> on rappelle la fonction avec X,A,s et p
      # qui restent inchangés et les nouvelles valeurs de phi et v_phi
      return (Ford_Fulkerson(X,A,s,p,phi,v_phi))
    }
  else{
      # Fin du programme, on retourne le flot maximal
      return(v_phi)
    }
}
```

4.4 Illustration

Nous allons tester notre fonction sur l'exemple représenté dans la figure ci-dessous, en prenant le sommet 4 comme source et le sommet 2 comme puits du réseau.




```
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=rep(1:nrow(A))
phi = matrix(0,length(X),length(X))
```

```r
v_phi=0

res=Ford_Fulkerson(X,A,4,2,phi,v_phi) # 4 = source, 2 = puit

print(paste("Le flot maximal du réseau est",res))

## [1] "Le flot maximal du réseau est 6"
```

En appliquant l'algorithme de Ford-Fulkerson sur ce graphe avec $s = 4$ et $p = 2$, on trouve donc que le flot maximal de ce réseau est de 6.

5 Conclusion

Dans ce travail nous avons donc implémenté trois algorithmes utilisés dans la résolution de problèmes d'optimisation : l'algorithme de Prim qui permet de trouver un arbre couvrant de poids minimal, l'algorithme de Ford-Bellman qui permet de calculer le plus court chemin entre des sommets d'un graphe et l'algorithme de Ford-Fulkerson qui permet de déterminer un flot maximal dans un réseau avec capacités.

Ce travail s'est avéré intéressant. Il nous a permis d'évaluer notre capacité à traduire des pseudo-codes d'algorithmes dans un langage de programmation et nous a permis de mieux comprendre et de mettre en pratique les notions vues en TP. Nous avons par la même occasion amélioré notre niveau de maîtrise du langage R et avons appris à rédiger un rapport de qualité dans un éditeur LaTeX avec la compilation des codes.

Nous dirions que la principale difficulté a été de programmer l'algorithme de Ford-Fulkerson. En effet nous avons eu du mal à traduire son pseudo-code en R et la gestion des boucles n'a pas été évidente du premier coup. Nous avons fait l'effort de programmer l'ensemble de nos fonctions en essayant de respecter au mieux les pseudo-codes fournis.