Dossier Optimisation et Recherche Opérationnelle

Thomas Duvignau / Chloé Boissavy

Université Lyon 2
M1 Informatique 2018 / 2019

Table des matières

I. Introduction

Nous avons écrit ce dossier afin de répondre à la demande d'une évaluation en contrôle continu de nos connaissances sur le cours d'Optimisation et de Recherche Opérationnelle, supervisé par le professeur Julien Ah-Pine au premier semestre de notre M1 Informatique.

Vous pourrez trouver ci-dessous différents algorithmes avec leurs utilités, leurs buts, leurs codes ainsi que des explications détaillées sur leurs fonctionnement et sur pourquoi ils ont été implémenter ainsi.

Nous nous sommes intéressés en premier sur l'algorithme de Welsh-Powell, c'est-à-dire la coloration d'un graphe non orienté. Puis nous avons effectué la détermination d'un flot maximal dans un réseau avec capacités par Ford-Fulkerson. Et pour finir, nous avons fait des chemins de longueur p avec fermeture transitive et connexité.

II. Algorithme 1 : Coloration d'un graphe non orienté avec Welsh-Powell

1. Objet de l'algorithme

Ici, nous traitons des graphes non orienté et non pondéré. Le but de cet algorithme est d'attribuer différentes couleurs aux sommets du graphe tout en respectant la règle qui stipule que deux sommets liés par une arrête ne peuvent avoir la même couleur. Il est tout de même préférable d'avoir le moins de couleurs possible, ainsi attribuer une couleur à chaque sommet est trop simple et peu utile si des solutions meilleures sont disponibles.

Ce genre d'algorithme est très utilisé dans le domaine de la télécommunication ou pour résoudre des problèmes d'incompatibilités. Par exemple, si vous avez des produits chimiques, vous devez les ranger dans une armoire à plusieurs étages. Sachant que certain de ces produits chimiques peuvent exploser s'ils sont à côté sur la même étagère. Il suffit de faire un graphe avec les incompatibilités de chacun puis d'utiliser Welsh-Powell afin de trouver une solution rapide et efficace qui permettra de ne rien faire exploser.

Il faut savoir tout de même que l'algorithme de Welsh-Powell ne permet pas nécessairement d'obtenir le nombre de coloration minimale d'un graphe G = [X, U]. On appelle cela, un algorithme heuristique.

2. Pseudo-code de l'algorithme

Voici ci-dessous l'algorithme que l'on a étudié en cours :

Input : A (matrice d'adjacence de G)

1. Ranger les sommets par ordre de degrés non croissant
2. k = 0
3. B = liste des sommets rangés par ordre de degrés non croissant
4. Tant que toutes les lignes de B ne sont pas colorées faire
5. k = k + 1
6. Tant que B ≠ Ø faire
7. Colorer dans A par la couleur c_k la 1ère ligne non colorée dans B ainsi que la colonne correspondante
8. B = liste des lignes non colorées ayant un zéro dans toutes les colonnes de A de couleur c_k
9. Fin Tant que
10. B = liste des sommets non colorés rangés par ordre de degrés non croissant
11. Fin Tant que
12. **Output :** k-coloration de G

3. Code R de l'algorithme

```r
Welsh_Powell2 = function(X,A){
  n = length(X)   #On cherche le nombre de sommets
  d=rowSums(A)    #somme de chaque lignes de A (les degres)
  s=sort(d,decreasing=TRUE,index.return=TRUE)
          #Permet de trier la liste d
  k = 0   #Permet de dire sur quel couleur on est
  B = s$ix #Liste des sommets triés
  lignenoncoloree = B
  c=list()

  while (length(lignenoncoloree)>0){ #Tant qu'il ya des sommets pas coloriés
    k=k+1
    temp=c()

    while(length(B)>0){
      temp[length(temp)+1]=B[1] #On prend le premier non colorié
      lignenoncoloree = setdiff(lignenoncoloree,B[1])
      #On l'enlève des sommets non colorié

      if (is.matrix(A[,temp])){ #Si on a une matrice
        B=which(rowSums(A[,temp])==0)
        #B=liste des lignes non colorié ayant un zéro dans toutes les
        #colonnes de A de couleur c[k]
      }
      else{
        B=which(A[,temp]==0)
        #Pareil que precedemment mais si on a pas de matrice
      }

      B=intersect(lignenoncoloree,B)
      #On enlève les nouvelles lignes coloriés
      c[[k]]=temp #On met les sommet dans la couleur qui correspond
    }

    B = lignenoncoloree
  }
  return(c)
}
```

4. Illustration sur un exemple

Pour illustrer cet exemple, nous allons utiliser la matrice d'adjacence :

A =

	1	2	3	4	5	6	7
1	0	1	1	0	1	0	1
2	1	0	1	1	0	1	0
3	1	1	0	1	1	0	1
4	0	1	1	0	1	1	1
5	1	0	1	1	0	1	1
6	0	1	0	1	1	0	1
7	1	0	1	1	1	1	0

Et on obtient comme résultat :

- X = c(1,2,3,4,5,6,7)
- A = cbind(c(0,1,1,0,1,0,1), c(1,0,1,1,0,1,0), c(1,1,0,1,1,0,1), c(0,1,1,0,1,1,1), c(1,0,1,1,0,1,1), c(0,1,0,1,1,0,1), c(1,0,1,1,1,1,0))
- Welsh_Powell2(X,A)

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

On a donc 4 couleurs pour le graphe, la couleur 1 est pour les sommets 3 et 6, la couleur 2 pour les sommets 4 et 1, la couleur 3 pour les sommets 5 et 2 et la couleur 4 pour le sommet 7.

III. Détermination d'un flot maximal dans un réseau avec capacités par Ford-Fulkerson

1. Objet de l'algorithme

Les problèmes de flots constituent un très important domaine de la théorie des graphes. Sous leur forme la plus simple, ils consistent à organiser de façon optimale, sous diverses contraintes, les mouvements de certaines quantités d'un bien dans un réseau.

L'algorithme prend en paramètre la matrice de capacité du graphe (A), l'ensemble des sommets du graphe (X), la source et le puits (s et p). Et il retourne une matrice (phi) qui contient la valeur du flot maximal.

2. Pseudo-code de l'algorithme

Voici ci-dessous le pseudo code de l'algorithme que l'on a étudié en cours :

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**

3. Code R de l'algorithme

```
Ford_Fulkerson = function (X,A,s,p)
{
  phi = matrix(0,nrow=length(X),ncol=length(X)) #initialisation de la matrice qui va contenir le flot
  V_phi = 0 #initialisation de la valeur du flot
  m=matrix(0,nrow=length(X),ncol=3) #initialisation de la matrice qui va servir pour le marquage
  while(1)
  {
    m[s,2] = Inf  #marquage de la source
    m[s,3] = 1    #marquage de la source
    S=c(s)  #On mets dans S la source
    Sb=setdiff(X,S) # S barre egale a l'ensemble des noeuds sauf ceux contenu dans S
    R1 = A-phi > 0 #Matrice qui contient TRUE si entre i et j il est possible de faire passer le flot sinon FALSE
    R2 = t(phi) > 0 #Matrice qui contient TRUE si le flot est superieure a 0 sinon non
    C = R1 | R2  # Matrice qui contient TRUE si le flot peut aller de i a j ou de j a i(càd flot[i,j] > 0)

    while(length(which(matrix(C[S,Sb]==TRUE,nrow=length(S),ncol=length(Sb)),arr.ind=TRUE)) > 0  )
    #tant qu'on trouve un ou plusieurs arc (i,j) où le flot peut passer
    {
          i = S[ which(matrix(C[S,Sb]==TRUE,nrow=length(S),ncol=length(Sb)),arr.ind=TRUE)[1,1] ] #premier i trouvé
          j = Sb[ which(matrix(C[S,Sb]==TRUE,nrow=length(S),ncol=length(Sb)),arr.ind=TRUE)[1,2] ]#premier j trouvé

          if( A[i,j] - phi[i,j] > 0) #test pour savoir si le flot passe de i à j
          {
            m[j,1] = i                                #On note le prédécésseur de j
            m[j,2] = min( m[i,2] , A[i,j] - phi[i,j] )  #On prend la valeur qui limite le plus le passage du flot
            m[j,3] = 1                                #On note que le flot passe dans le sens de l'arc (i,j)
          }
          else if( phi[j,i] > 0 ) # ou si il passe de j à i
          {
            m[j,1] = i                                #On note le prédécesseur de j
            m[j,2] = min( m[i,2] , phi[j,i] )           #On prend la valeur qui limite le plus le passage du flot
            m[j,3] = -1                               #On note que le flot passe dans le sens contraite de (i,j)
          }
          S[length(S)+1]=j # On marque j dans S
          Sb=setdiff(X,S) # On actualise Sb car S a changé
          if(j == p) # test pour savoir si on est arrivé au puit
          {
            V_phi =+ m[p,2] #si c'est le cas on augmente la valeur du flot et on vas en 47
            break
          }
      }
      if( is.element(p,S) ) # on test si p a été marqué
      {
        while( j != s ) # si il a été marqué on remonte le chemin parcourue jusqu'a s
        {
          if( m[j,3] == 1 ) # on test si on traverse l'arc dans le bon sens
            phi[ m[j,1] ,  j] = phi[ m[j,1] ,  j] + m[p,2]

          else if( m[j,3] == -1) # ou si on traverse l'arc dans le sens contraire
            phi[  j , m[j,1] ] =phi[  j , m[j,1] ]- m[p,2]

          j = m[j,1]
        }
      }
      else # si il n'as pas été marqué cela veut dire qu'il n'existe plus de chaine augmentante entre s et p
        return(phi) # donc on renvoie la matrice phi qui contient le flot maximal
  }
}
```

4. Illustration sur un exemple

Le graphe :

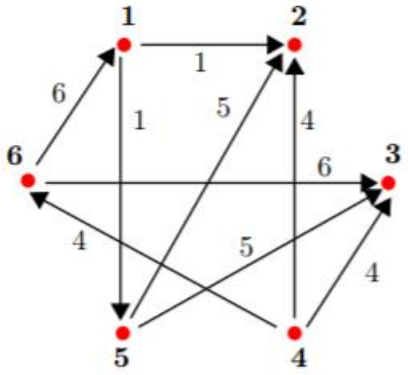

FIGURE 1 – Réseau avc capacités.

Sa matrice de capacité :

	[,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	4	0	0	4
[5,]	0	5	5	0	0	0
[6,]	6	0	0	0	0	0

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

```
> 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(6,0,0,0,0,0))
> phi = Ford_Fulkerson(1:6,B,4,2)
> phi
     [,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
```

IV. Chemins de longueur p, fermeture transitive et connexité

1. Objet de l'algorithme

Cette partie se compose de deux algorithmes, le premier qui permet de calculer la puissance booléenne d'une matrice et le second qui permet de trouver la fermeture transitive d'un graphe à partir de sa matrice d'adjacence.

Le calcul de la puissance booléenne d'une matrice revient finalement à calculer sa puissance normale et à remplacer les éléments de la matrice supérieure à 1 par un 1.

En effet si l'on prend la définition de la somme booléenne tel quel :

$$x \oplus y = \begin{cases} 1 \text{ si } x = 1 \text{ OU } y = 1 \\ 0 \text{ sinon} \end{cases}$$

Alors on remarque que x+y est égale à un nombre différent de 0 si x et y sont différents de zéro, donc si le résultat de leur somme est supérieur à 1 il suffit de le ramener à 1 pour avoir le résultat de la somme booléenne.

De même si l'on prend la définition du produit booléen :

$$x \otimes y = \begin{cases} 1 \text{ si } x = 1 \text{ ET } y = 1 \\ 0 \text{ sinon} \end{cases}$$

On remarque que x*y est égale à un nombre différent de zéro si x et y sont différents de zéro. Donc si le résultat de leur produit est supérieur à 1 il suffit de le ramener à 1 pour avoir le résultat du produit booléen.

Donc dans les algorithmes suivants il y a la présence de boucles for imbriqués pour tester chaque élément de la matrice et les mettre à 1 si ils sont supérieur à 1.

2. Pseudo-code de l'algorithme

Pseudo code de l'algorithme de la puissance :

Input : A

1. result ← A
2. **Pour** p **de** 1 **à** (p-1)
3. result ← result * A
4. **Fin Pour**
5. **Pour** i **de** 1 **a** N
6. **Pour** j **de** 1 **à** N
7. **Si** result[i][j] > 1
8. result[i][j] ← 1
9. **Fin Si**
10. **Fin Pour**
11. **Fin Pour**
12. **Output** : result

Pseudo code de l'algorithme de la fermeture transitive :

Input : A
$\widehat{A} \leftarrow A$
Pour p de 2 à N **faire**
    Calculer A^p
    $\widehat{A} \leftarrow \widehat{A} \oplus A^p$
Fin Pour
Output : $\widehat{A}$

3. Code R de l'algorithme

Code de l'algorithme de la puissance :

```
puis_bool = function (A,p)
{
  result = A

  for(i in 1:(p-1))
  {
    print(i)
    result = result %*% A
  }

  for(i in 1:sqrt(length(A)))
    for(j in 1:sqrt(length(A)))
      if(result[i,j] >= 1)
        result[i,j]=1

  return(result)
}
```

Code de l'algorithme de la fermeture transitive :

```
ferm_trans = function (A)
{
  A_ = A
  N = sqrt(length(A))

  for(p in 2:N )
  {
    A_ = A_ + puis_bool(A,p)
  }

  for(i in 1:sqrt(length(A_)))
    for(j in 1:sqrt(length(A_)))
      if(A_[i,j] >= 1)
        A_[i,j]=1

  return(A_)
}
```

4. Illustration sur un exemple

Matrice d'adjacence de A :

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

Application de notre fonction puissance :

```
>   p = 2
>   A=rbind(c(0,1,0,0,0),c(1,0,1,0,0),c(0,0,0,1,1),c(0,0,0,0,1),c(1,0,0,0,0))
>
>   r=puis_bool(A,2)
[1] 1
>   r
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    1    0    0
[2,]    0    1    0    1    1
[3,]    1    0    0    0    1
[4,]    1    0    0    0    0
[5,]    0    1    0    0    0
```

Donc d'après ce résult il existe des chemins de longueur deux entre les couples de nœuds suivants : (1,1) (1,3) (2,2) (2,4) (2,5) (3,1) (3,5) (4,1) (5,2)

Si on reprend la matrice des adjacences de A on peut facilement retrouver ces chemins :

(1,1) = 1→2→1 (1,3) = 1→2→3 (2,2) = 2→1→2 (2,4) = 2→3→4

(2,5) = 2→3→5 (3,1) = 3→5→1 (3,5) = 3→4→5 (4,1) = 4→5→1

(5,2) = 5→1→2

Application de la fonction fermeture transitive sur A :

```
> A=rbind(c(0,1,0,0,0),c(1,0,1,0,0),c(0,0,0,1,1),c(0,0,0,0,1),c(1,0,0,0,0))
>   A_ = ferm_trans(A)
>   A_
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1    1    1
[2,]    1    1    1    1    1
[3,]    1    1    1    1    1
[4,]    1    1    1    1    1
[5,]    1    1    1    1    1
```

On remarque que la matrice de la fermeture transitive est composée uniquement de 1. Cela veut dire qu'il existe un chemin entre 1 et 1, 1 et 2, etc. Et donc cela veut dire qu'à partir de n'importe quel nœud du graphe on peut se rendre sur un autre nœud du graphe grâce à un chemin de taille inférieure ou égale à N.

V. Conclusion

Comme vous avez pu le constater nous avons donc étudié 3 algorithmes relativement différent. Nous savons donc maintenant faire la coloration d'un graphe grâce à Welsh-Powell, trouver un flot maximal grâce à Ford-Fulkerson puis effectuer la fermeture transitive d'un graphe.

Les seules difficultés ici, ont été de nous familiariser avec le langage R qui est un langage que mon binôme et moi-même utilisons pleinement que depuis cette année.

Pour conclure, ce projet reprend bien toutes les différentes fonctions que nous avons étudiés durant les cours et TP d'Optimisation et Recherche Opérationnelle.