Université de Nantes — UFR Sciences et Techniques
Master informatique parcours “optimisation en recherche opérationnelle (ORO)”
Année académique 2018-2019

Projet Graphes II et Réseaux

Marie Humbert–Ropers[1] – Arthur Gontier[2]

14 décembre 2018

Table des matières

1 Lancement du Code / Compilation

La compilation se réalise via la commande suivante :

```
javac GrandMain.java
```

Il y a trois arguments à rajouter. Le premier concerne la version, c'est-à-dire la méthode d'implémentation choisie, : matrice(0) ou tableau de Noeuds(1). Le deuxième concerne le nom du fichier, qui doit absolument être mis dans le dossier jeu_donnees au préalable s'il n'y est pas déjà. Le troisième concerne la verbosité : si l'on veut avoir le détail des étapes : verbose sinon une suite de caractères quelconques. La commande pour lancer le code est donc :

```
java GrandMain <version: 0 ou 1> <nom du fichier> <verbose ou autre>
```

exemple :

```
java GrandMain 1 exemple1.txt verbose
```

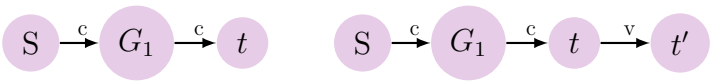
Il existe 5 fichiers de jeu de données, nommés exemple(1 à 5).txt

2 Réductions

2.1 Etape 1

Flot fixé v "R admet un flot de valeur v si et seulement si R' admet un flot maximum de valeur v". Si le réseau R admet un flot maximum de valeur v ou plus, il est toujours possible de trouver, avec des valeurs entières, un flot tel que R admet un flot de valeur égal exactement à v. Dans le cas contraire, si le flot maximum est inférieur à v, il sera impossible de trouver un flot de valeur égal à v.
En partant sur ce principe, il suffit de partir du réseau R, et d'appliquer sur le puits t de ce réseau la contrainte suivante : de ne pas avoir un flot de plus de v, c'est-à-dire une capacité de v sur le puits t. Ainsi, selon le résultat, on peut distinguer les deux cas précédents. Afin d'utiliser l'algorithme des préflots sur ce réseau, il faut le modifier légèrement. Le sommet t est suivi d'un nouveau sommet t', t' devenant le puits. L'arc entre les deux aura une capacité de v.

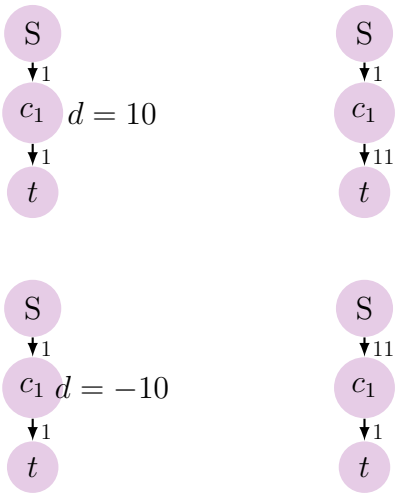
Exemple 1 *Etape 1 : Ajout d'un puits*



2.2 Etape 2

Modification du réseau pour réduire le problème à un problème de circulation Dans ce problème, il faut satisfaire une demande à chaque noeud d'un réseau R. Une nouvelle source S et un nouveau puits T sont créés. Pour ceci, les sommets sont séparés en deux catégories : ceux avec des demandes positives et ceux avec des demandes négatives. Dans le premier cas, un arc est ajouté entre le sommet à demande positive vers T. La capacité de cet arc est celle de la demande du sommet. Dans le second cas, les demandes sont négatives donc, un arc est ajouté à partir de la source S vers le sommet de demande négative, avec une capacité correspondant à la valeur absolue de la demande du sommet. Ensuite,il reste à chercher le flot maximal de ce nouveau graphe R'. Si l'on ne parvient pas à satisfaire les demandes, alors le problème est impossible et le flot est inférieur à la valeur fixée précédemment. On note que la source et l'ancienne source sont reliées et de même pour l'ancien puits et le nouveau puits, grâce aux demandes positionnées sur les anciens source et puits.

Exemple 2 *Etape 2 : Transformation des demandes*



Existence d'un algorithme polynomial Soit $R(S, A)$
D'après la construction précédente, il a été nécessaire de créer une nouvelle source et un nouveau puits. En plus de ces deux actions, la construction nous permet de dire que pour chaque sommet dans le graphe R, il est nécessaire d'ajouter un arc ayant chacun comme capacité la valeur absolue de la demande. Ensuite, deux cas sont distingués : l'ajout d'un arc d'un sommet vers le nouveau puits (dans le cas de demande positive) ou bien l'ajout d'un arc de la source vers le sommet pour une demande négative. Il y a donc $|S|$ tests suivis au pire de $|S|$ affectations (ajout d'un arc). Cela conclut la transformation de R vers R'. On note donc que le nombre d'étapes est en $\mathcal{O}(|S|)$.

2.3 Etape 3

Réduction du problème de l'arc-circulation vers celui de circulation Dans le problème d'arc-circulation, une capacité minimum est à satisfaire sur chaque arc. L'objectif est de transformer cet arc-circulation en circulation, c'est-à-dire avec seulement des capacités (maximum) sur les arcs et des demandes sur les sommets. Nous cherchons donc à transformer les minima sortants d'un sommet en une demande sur ce sommet. Les nouvelles capacités sont calculées à partir des changements suivants :

— Les nouvelles capacités sont, pour tout i :

$$C'_{ij} = C_{ij} - min_{ij}$$

— Les demandes sur un sommet sont :

$$D_i = \sum_{k \in VS} min_{ik} - \sum_{k \in VE} min_{ik}$$

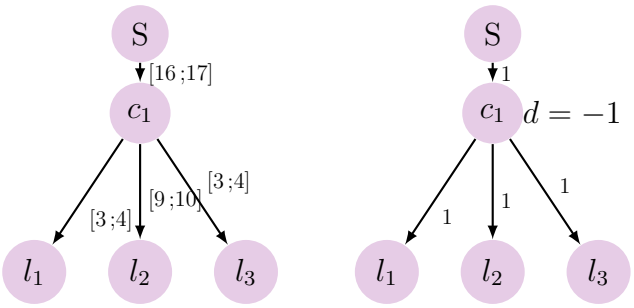
pour un sommet i, son voisinage sortant VS et son voisinage entrant VE

Exemple 3 *Comme les arrondis sont dans un intervalle de 1, on a :*

$$\forall i, j : C'_{ij} = C_{ij} - min_{ij} = 1$$

et

$$D_i = \sum_{k \in VS} min_{ik} - \sum_{k \in VE} min_{ik} = 3 + 9 + 3 - 16 = -1$$

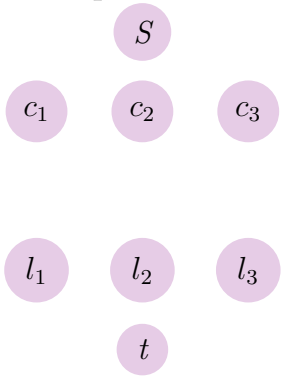


Existence d'un algorithme polynomial En nous basant sur l'idée de la construction précédente, nous partons d'un arc-circulation sur lequel nous modifions tous les arcs et ajoutons des demandes sur chaque sommet. Sur chaque arc, seul une affectation est effectuée, mais sur chaque sommet, une affectation est précédée de la somme des capacités des arcs vers les voisins. Soit $|S|$ au pire des cas, si le graphe est complet, c'est-à-dire que chaque sommet est relié à tous les autres sommets directement. Dans ce cas, le nombre d'opérations sera $|S|^2$. Puisqu'un arc n'est changé qu'une seule fois, le nombre de calculs est donc en $\mathcal{O}(|S|^2 + |A|)$. Ainsi, nous obtenons le réseau R' et la réduction est polynomiale.

2.4 Etape 4 : Formulation du problème Arrondis-2D

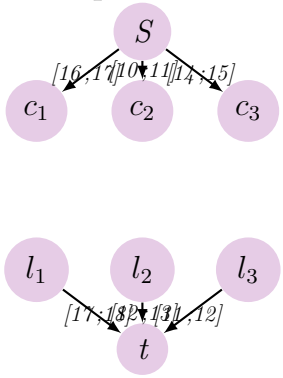
Pour réduire le problème Arrondis2D de taille $3 * 3$, on créé 6 sommets plus une source s et un puits t. Les cases de la dernière colonne (resp. ligne) correspondent chacune à la somme des cases d'une ligne (resp. colonne). Ces cases deviennent des sommets dans le graphe. On les nomme c1, c2 et c3 pour ceux issus des colonnes, puis l1, l2, et l3 pour les lignes.

Exemple 4 *Sommets*



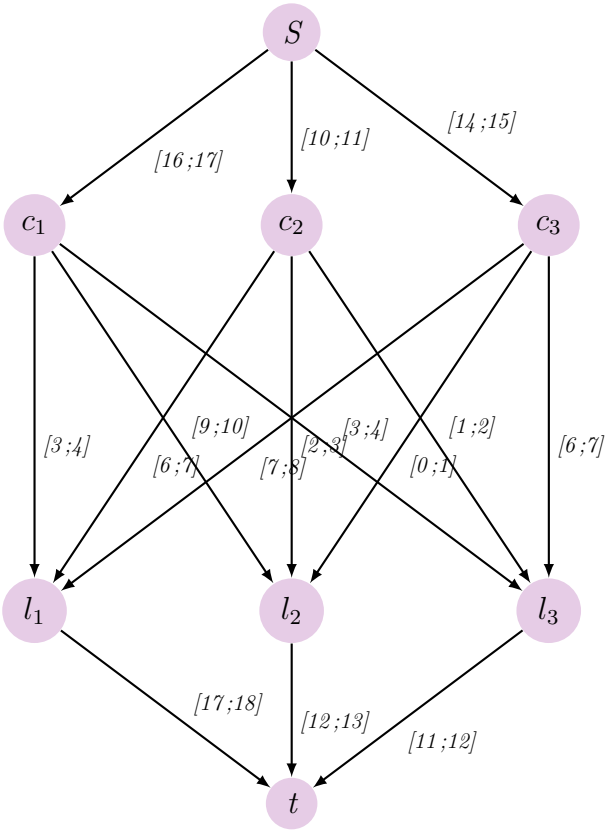
De manière générale, une source et un puits sont créés ainsi que n+m sommets, avec n et m ,respectivement, le nombre de colonnes et le nombre de lignes de la matrice. Nous les appelons donc c_i et l_j par la suite. On note que $i \in 1, .., n$ et $j \in 1, .., m$. Ensuite, la source est reliée aux sommets c_i, avec des arcs comportant un minimum et une capacité pour que ce soit une arc-circulation. Le minimum est la borne inférieure de l'arrondi de la somme de la ligne i et la capacité correspond à la borne supérieure. Les sommets l_j sont reliés au puits avec des arcs qui portent de la même manière que précédemment : le minimum s'obtient par la borne inférieure de la somme de la colonne donnée et la capacité par la borne supérieure de cette même somme.

Exemple 5 *Arcs de la source et du puits*



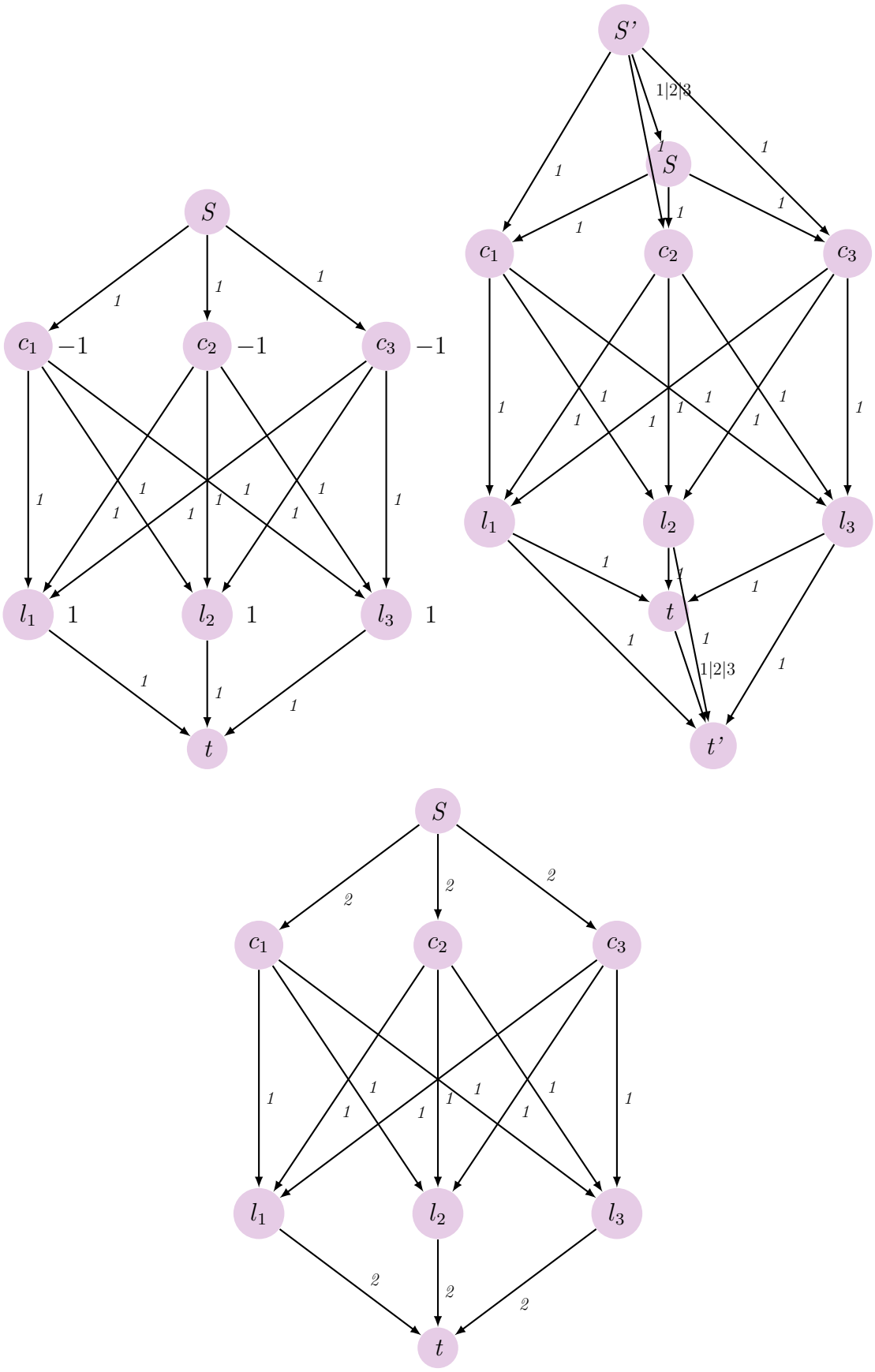
Ensuite, les sommets c_i sont reliés aux sommets l_j par des arcs pondérés. A partir de chaque case du tableau, un arc est créé de la manière suivante : la case sur la colonne i et sur la ligne j donnera un arc entre c_i et l_j dont le poids minimum sera l'arrondi inférieur et la capacité, l'arrondi supérieur. On obtient alors un arc-circulation.

Exemple 6 *Construction*



Remarques : Nous avons développé deux variantes lors de l'étape 2. Nous avons remarqué que les arcs créés à l'étape 2 sont parfois double entre deux sommets du graphe. Il y a alors deux possibilités : créer une nouvelle source et un nouveau puit pour éviter ces doublons ou bien fusionner ces arcs doublons. Les deux représentations sont similaires, à ceci près que la première approche permet d'agir sur la demande de l'ancienne source, afin de pouvoir réguler le flot et permettre d'avoir toutes les demandes comblées. Ne sachant pas quelle est la meilleure approche, nous avons essayé les deux. Elles sont expliquées plus en détails dans la partie implémentation, avec la régulation du flot dans la version du Tableau de Noeuds et la version fusionnée dans la représentation matricielle.

Exemple 7 *Réductions*



Existence d'un algorithme polynomial Nous partons du tableau arrondi-2D de n cases * m cases. Chaque case du tableau correspond à un arc du graphe. Le nombre de noeuds est la

somme du nombre de lignes et de colonnes. Cela représente donc $n+m$ noeuds. Il faut rajouter à cela 2 noeuds : la source et le puits. Ensuite, les différents noeuds sont reliés par des arcs. Ces arcs proviennent de chaque case du tableau, et il y en a $n*m$. Les minima et capacités sont obtenues directement du tableau : respectivement, l'entier inférieur et supérieur du réel de la case du tableau. Enfin, il faut ensuite ajouter les arcs qui vont de la source vers les sommets c_i , soit m arcs puis n arcs allant des sommets l_j vers le puits. Le minimum pour un c_i (resp. colonne) s'obtient par la borne inférieure de la somme de la colonne (resp. l_j) donnée et la capacité par la borne supérieure de cette même somme. Cela correspond à $n+m$ arcs à créer, sachant qu'il faut faire les sommes au préalable soit $2*n*m$ opérations de temps constant. Ainsi, on compte donc un nombre de $4*n*m$ opérations de temps constant. Cela reste donc un temps polynomial en la taille des données, soit $\mathcal{O}(n*m)$.

3 Implémentation

L'implémentation de graphes dans le langage java encourage à utiliser le paradigme objet pour représenter les différents graphes des réductions et de programmer un préflot utilisant cette structure. Mais en réfléchissant aux réductions, il est possible de passer par une représentation matricielle du graphe généré par le problème initial. Ne sachant pas trop quelle représentation serait la plus simple à utiliser, nous avons décidé d'essayer d'implémenter les deux et de les comparer. Nous avons donc la première implémentation utilisant le paradigme orienté objet avec un tableau de Noeuds et la seconde implémentation se basant uniquement sur une matrice. Notons aussi que la méthode des préflots utilisée est celle d'Elever vers l'Avant/Décharger.

3.1 Vue globale de la première approche

Structure des données
Dans cette première approche, nous utilisons comme élément de base la classe de Noeuds. Le graphe issu de la matrice des données correspond à un tableau de Noeuds. Chaque Noeud possède une hauteur, une charge e et une demande, qui sont à 0 par défaut. Pour faciliter l'utilisation de la méthode des préflots, chaque élément de la classe Noeuds contient une liste d'entiers permettant d'identifier les prédecesseurs et successeurs du noeud. Chaque noeud contient les informations des arcs sortants, c'est-à-dire les capacités, les minima ainsi que les flots, par défaut à 0. Le graphe se construit à partir de la matrice de base, le constructeur de Graphes faisant appel à l'étape 4 de la construction.

Pseudo-code en annexe Algorithmes 1 à 5

3.2 Vue globale de la deuxième approche

Structure des données
Pour cette deuxième version, on utilise une matrice d'adjacence liée à la structure de notre graphe. En effet, si l'on ne construit pas de nouvelle source ou de nouveau puits (qui sont fusionnés à la source et au puits pour calculer le flot max), le graphe est alors un graphe presque biparti avec les sommets c_i qui sont reliés à la source mais pas entre eux et les sommets

l_j qui sont reliés au puits mais pas entre eux. On a donc pour cette version, une implémentation du préflot qui est spécifique au problème.

La matrice découle directement du problème arrondi-2D. On a donc une matrice de taille $m * n$ où la sous matrice $n - 1 * m - 1$ contient les capacités des arcs qui sont entre les noeuds c_i et les noeuds l_j. La dernière ligne de la matrice contient les capacités des arcs qui partent de la source vers les noeuds c_i. Et la dernière colonne contiens les capacités des arc qui partent des noeuds l_j vers le puits.

Pseudo-code en annexe Algorithmes 6 à 10

4 Conclusion

D'après les différentes étapes, le problème Arrondi2D est réductible à un problème de préflot en un temps polynomial en la taille des données. On note qu'il est aussi possible de récupérer le résultat de la méthode du préflot et de retrouver la solution du problème initial en un temps polynomial en la taille de données. De plus, la méthode du préflot est une méthode en $Card(S^3)$, donc la résolution du problème Arrondi2D se fait en un temps polynomial en la taille des données. Nous constatons aussi que l'implémentation utilisant uniquement la matrice est très économe en mémoire et en temps. Cependant, il est plus complexe de réguler le flot. Le contrôle du flot se réalise plus facilement pour la version orientée objet, qui trouve plusieurs solutions dont la plus contrainte possible qui satisfait toutes les demandes.

A Jeux de données

```
3
3
3,14 6,8 7,3
9,6 2,4 0,7
3,6 1,2 6,5
.
```

(a) exemple1

```
3
3
2,9 4,9 5,5
6,5 3,0 1,1
1,1 6,0 10,1
.
```

(b) exemple2

```
2
2
0,9 0,01
0,9 0,01
.
```

(c) exemple5

```
5
6
2,4 8,9 17,8 2,1 4,6 28,1
4,1 5,2 4,6 7,9 2,3 4,1
10,4 9,1 5,4 3,3 2,0 4,5
7,8 8,9 5,6 5,4 3,2 12,6
4,2 5,1 6,6 7,7 10,9 24,3
.
```

(d) exemple3

```
6
5
2,4 8,9 17,8 2,1 4,6
4,1 5,2 4,6 7,9 2,3
10,4 9,1 5,4 3,3 2,0
7,8 8,9 5,6 5,4 3,2
4,9 5,1 6,6 7,7 10,9
28,9 4,1 4,5 12,6 24,3
.
```

(e) exemple4

B Pseudo-code

B.1 Implémentation objet

Notons que dans cette section, m correspond à la taille des lignes du Tableau et n correspond à la taille des colonnes du Tableau. Par la suite, le nombre de sommets est noté $Card(S)$ et, par construction de l'étape 4, $Card(S) = n + m$. Les procédures rajoutent 3 noeuds, soit un nombre constant de noeuds donc, l'ordre restera de $Card(S)$.

Algorithme 1 : Procédure : ConstructionEtape1 $\mathcal{O}(1)$

Entrées : G, un graphe

```
1 T ← dernier Noeuds du Tableau de Noeuds de G
2 P ← new Noeuds                                  /* Création du nouveau Puits */
3 Ajout de P à la fin du Tableau de Noeuds de G
4 Ajout de l'arc de T à P
```

Dans la procédure ConstructionEtape2, l'opération "Ajout de l'arc de T à P" correspond à l'ajout de la capacité (somme des capacités des prédecesseurs) et de la présence de P dans la liste des successeurs de T.

Algorithme 2 : Procédure : ConstructionEtape2 $\mathcal{O}(Card(S))$

Entrées : G, un graphe

```
1  T ← new Noeuds                                  /* Création du nouveau Puits */
2  S ← new Noeuds                                /* Création de la nouvelle Source */
3  Ajout de T à la fin du Tableau de Noeuds de G
4  Ajout de S au début du Tableau de Noeuds de G
5  for chaque sommet N du Tableau de Noeuds de G do
6  |  Si Demande(N) < 0 alors
7  |  |  Ajout d'un arc de S à N to
8  |  |  Capacité(S,N) ← -Demande(N)
9  |  |  Demande(N) ← 0
10 |  fin
11 |  Sinon si Demande(N) > 0 alors
12 |  |  Ajout d'un arc de N à T
13 |  |  Capacité(N,T) ← Demande(N)
14 |  |  Demande(N) ← 0
15 |  fin
16 end
```

La valeur v permet de contraindre le flot, plus précisément de restreindre le flot allant de l'ancienne source de l'étape 3 vers la nouvelle source de l'étape 2. L'algorithme s'arrête lorsque la vérification montre que le flot est trop restreint et que résultat ne répond plus au problème initial. Pour faciliter le contrôle du flot, la valeur v a été rajoutée dès l'étape 3.

Algorithme 3 : Procédure : ConstructionEtape3 $\mathcal{O}(Card(S))$

Entrées : G, un graphe, v, un entier

```
1  for chaque sommet N du Tableau de Noeuds de G do
2  |  for chaque successeur J de N do
3  |  |  Capacite(N,J) ← Capacite(N,J) - Minimum(N,J)
4  |  |  Demande(N) ← Demande(N) + Minimum(N,J)
5  |  |  Demande(J) ← Demande(j) - Minimum(N,J)
6  |  |  Minimum(N,J) ← 0
7  |  end
8  end
9  Demande(Source S) ← -(somme des capacités des arcs sortants) + v
10 Demande(Puits P) ← somme des capacités des arcs entrants
```

Algorithme 4 : Constructeur de Graphes : ConstructionEtape4 $\mathcal{O}(n * m)$

Entrées : M, une matrice de taille n*m

Sorties : G, un graphe

```
1  Table ← Tableau de Noeuds vide de taille (n + m + 2)
2  P ← Table[n+m+1]                                          /* le puits */
3  for k ← 0 to m − 1 do
4  |  N ← Table[k+1+n]
5  |  Ajout d'un arc entre N et P          /* Ajout de P à la liste des successeurs */
6  |  Capacite(N,J) ← ⌊M[k][n − 1]⌋
7  |  Minimum(N,J) ← ⌊M[k][n − 1]⌋ + 1
8  end
9  for j ← 0 to n − 1 do
10 |  N ← Table[j+1] for k ← 0 to m − 1 do
11 |  |  N ← Table[k+n]
12 |  |  Ajout d'un arc entre N et R        /* Ajout de R à la liste des successeurs */
13 |  |  Capacite(N,R) ← ⌊M[k][j]⌋
14 |  |  Minimum(N,R) ← ⌊M[k][j]⌋ + 1
15 |  end
16 end
17 S ← Table[0]                                               /* la source */
18 for k ← 0 to n − 1 do
19 |  N ← Table[k+1]
20 |  Ajout d'un arc de S vers N           /* Ajout de N à la liste des successeurs */
21 |  Capacite(S,N) ← ⌊M[m − 1][k]⌋
22 |  Minimum(S,N) ← ⌊M[m − 1][k]⌋ + 1
23 end
24 retourner G, un graphe
```

Dans l'algorithme 1, les méthodes suivantes sont des procédures appliquées sur le graphe G : ConstructionEtape3, ConstructionEtape2, ConstructionEtape1 et EleverVersLAvant. Le constructeur de la classe Graphes correspond à la ConstructionEtape4. Les autres méthodes sont des fonctions. GetResultats parcourt le graphe et retourne le résultat du préflot appliquée sur le graphe G. verifResultats vérifie ce résultat et retourne un booléen.

Algorithme 5 : main Objet

Entrées : M, une matrice
Sorties : G, un graphe

```
1  table ← readScanner()
2  table ← remplissageSum(table)          /* calculs des sommes colonnes et lignes */
3  v ← 0                    /* v la valeur qui contraint au fur et à mesure le flot */
4  while marche && v < nombre de noeuds touchée par la source - 1 do
5  |  G ← new Graphe (table)                                  /* ConstructionEtape4 */
6  |  G.ConstructionEtape3
7  |  G.ConstructionEtape2
8  |  G.ConstructionEtape1
9  |  G.EleverVersLAvant          /* Méthode des préflots appliquée sur le graphe G */
10 |  Tab ← G.GetResultats(M)
11 |  marche ← verifResultats(Tab)
12 |  v ← v + 1
13 end
```

B.2 Implémentation matricelle

Algorithme 6 : main Matrice

```
1  tab ← readScanner()
2  tab ← remplissageSum(tab)              /* calcul des sommes colonnes et lignes */
3  G4 ← ConstructionReseau(tab)
4  d ← ConstructionEtape3(G4)    /* calcul du vecteur des demandes et des nouveaux
     arc(tous a un dans notre cas) */
5  G2 ← ConstructionEtape2(G4,d)
6  G0 ← Preflot(G2)
7  demande ← ConstructionEtape1(G0,d)
8  G ← remplissageSum(constructSol(G0,G4))        /* calcul des sommes entières */
9  println("Demandes satisfaites ? "+demande)
10 println("Sommes correctes ? "+verifSum(G))
```

Algorithme 7 : ConstructionReseau $\mathcal{O}(Card(A))$

Entrées : *tab*, la matrice de départ
Sorties : $G4$, un vecteur des demandes

```
1  G4 ← new int[tab.length][tab[0].length]
2  for k ← 0 to tab.length do
3  |  for j ← 0 to tab[0].length do
4  |  |  G4[k][j] ← (int) tab[k][j]
5  |  end
6  end
7  retourner G4
```

Algorithme 8 : ConstructionEtape3 $\mathcal{O}(Card(A))$

Entrées : $G4$, un graphe
Sorties : d, un vecteur des demandes

```
1  d ← new int[G4.length + G4[0].length-2]
2  k ← 0
3  for i ← 0 to G4.length -1 do
4  |  s ← 0                                  /* calcul des demandes des noeuds colones */
5  |  for j ← 0 to G4[0].length -1 do
6  |  |  s ← s + G4[i][j]
7  |  end
8  |  d[k] ← s - G4[i][G4[0].length-1]
9  |  k++
10 end
11 for j ← 0 to G4[0].length -1 do
12 |  s ← 0                                   /* calcul des demandes des noeuds lignes */
13 |  for i ← 0 to G4.length -1 do
14 |  |  s ← s + G4[i][j]
15 |  end
16 |  d[k] ← s + G4[G4.length-1][j]
17 |  k++
18 end
19 retourner d
```

Algorithme 9 : ConstructionEtape2 $\mathcal{O}(Card(A))$

Entrées : $G3$, un graphe, d, un vecteur de demandes
Sorties : $flag$, un boolean

```
1  G2 ← new int[G3.length][G3[0].length]
2  for i ← 0 to G3.length-1 do
3  |  for j ← 0 to G3[0].length-1 do
4  |  |  G2[i][j] ← 1                          /* on met des 1 sur tous les arcs */
5  |  end
6  end
7  k ← 0
8  for i ← 0 to G3.length-1 do
9  |  G2[i][G3[0].length-1] ← 1 - d[k]  /* ajout des demandes qui viennent de la source */
10 |  k++
11 end
12 for j ← 0 to G3[0].length-1 do
13 |  G2[G3.length-1][j] ← 1 + d[k]      /* ajout des demandes qui vont vers le puits */
14 |  k++
15 end
16 G2[G3.length-1][G3[0].length-1] ← 0
17 retourner G2
```

Algorithme 10 : ConstructionEtape1 $\mathcal{O}(Card(A))$

Entrées : G, un graphe, d, un vecteur de demandes
Sorties : $flag$, un boolean

```
1 flag ← true                                   /* satisfiabilité des variables */
2 j ← 0
3 for i ← G.length − 1 to d.length do
4 |   flag ← flag & (d[i] ≤ G[G.length − 1][j])
5 |   j++
6 end
7 retourner flag
```