Projet Comptes ronds

Solène Catella, Hugo Le Baher

Jeudi 13 Décembre

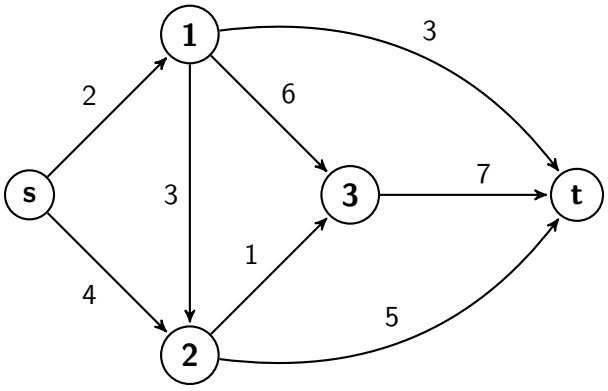
1 Introduction

Ce projet porte sur la résolution du problème Arrondis-2D. Le problème Arrondis-2D, défini pour un tableau à deux dimensions, s'attache à déterminer l'arrondi supérieur ou inférieur de chacune des valeurs du tableau, telle que la somme des valeurs arrondies sur chaque ligne (respectivement colonne) soit égale à l'arrondi de la somme des valeurs en ligne (respectivement colonne). L'implémentation ici fournie (programme Java) résout ce problème à base de flots, en quatre étapes distinctes.

2 Formulation du problème à l'aide de flots

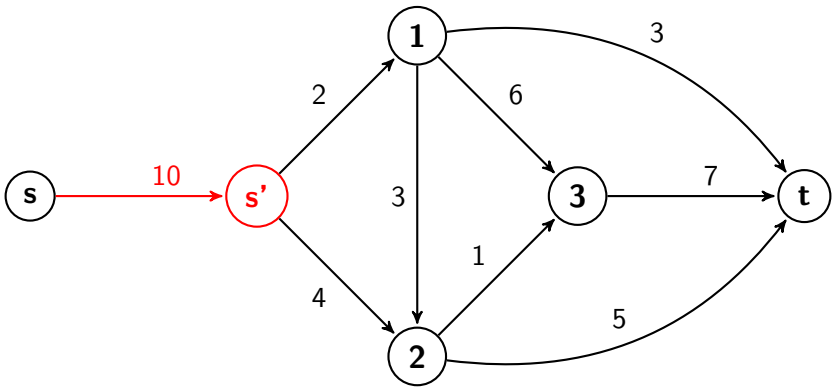
2.1 Etape 1

Soit R le réseau de transport tel que présenté ci-contre, de source s et de puits t, dont les arcs sont de capacité entière; soit v une valeur entière. On veut savoir si ce réseau de transport admet un flot de valeur exactement v.



Réseau R

L'idée est ici de transformer notre réseau R de flot de valeur v en un réseau R' de flot de valeur maximum. Le passage de R à R' se fait en substituant le sommet source s par un sommet intermédiaire s', successeur de s, telle que la capacité de l'arc allant de s à s' soit bornée par la valeur v, ici fixée à 10 par exemple :



Réseau R'

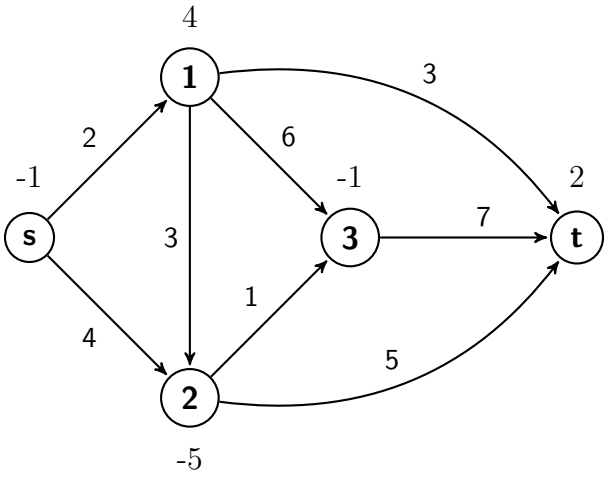
En limitant la capacité de l'arc joignant s à s' au flot de valeur exactement v, le flot circulant dans R' sera nécessairement un flot de valeur maximum. Réciproquement, si R' admet un flot maximum de valeur v, alors le flot de R sera nécessairement un flot de valeur exactement v.

2.2 Etape 2

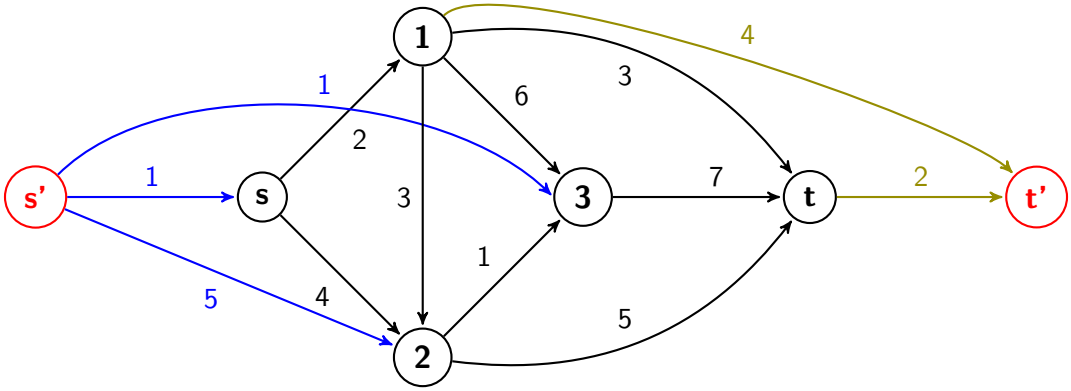
Dans cette deuxième étape, chaque sommet i (incluant s et t) formule une demande entière d_i. On cherche ici à transformer notre réseau R initial en réseau R' tel que R admette un flot satisfaisant les demandes de tous les sommets (appelé aussi circulation) si et seulement si R' admet un flot de valeur fixé v.

Passer de R à R' revient à ajouter un nouveau sommet s' (respectivement t') adjacent au sommet s (respectivement t) tel que s' (respectivement t') soit le prédécesseur (respectivement successeur) des sommets de demande négative (respectivement positive). Les valuations sur les nouveaux arcs correspondent aux valeurs absolues des demandes des sommets associés.

En considérant le réseau R tel que défini en 2.1, augmenté de demandes (positives ou négatives) sur chacun de ses sommets, on obtient le réseau R' suivant :



Réseau R (avec ajout des demandes)



Réseau R' (demandes négatives en bleu ; demandes positives en vert)

Dans le réseau R', toutes les demandes sont nécessairement satisfaites puisque les capacités des nouveaux arcs sont fixées par les demandes des sommets respectifs auxquels ils sont adjacents. Les demandes positives ont été substituées par des arcs avant valués vers le sommet t'; les demandes négatives redirigées en partance du sommet s', puisque ces demandes traduisent un besoin/manque devant être satisfait.

La résolution de ce problème revient à : (1) ajouter deux sommets s' et t' ; (2) parcourir chaque sommet du réseau; (3) vérifier la valeur de sa demande; (4) ajouter un arc orienté de s' vers le sommet traité si la demande est négative; (4bis) ajouter un arc orienté du sommet traité vers t' si la demande est positive.

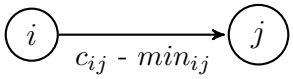
Il peut donc être résolu en temps polynomial : **complexité en O(n)**, où n représente le nombre de sommets.

2.3 Etape 3

Dans cette troisième étape, les flots sont désormais bornés par un minimum et un maximum (la capacité). Ainsi, plus formellement, sur tout arc ij, le flot circulant est nécessairement compris entre min_{ij} et c_{ij}, la capacité de l'arc.

La nouvelle configuration du réseau R' est telle que le flot circulant dans R' satisfait les contraintes de tous les arcs si et seulement si R admet une circulation.

Chaque arc ij se voit attribuer une nouvelle capacité $c'_{ij} = c_{ij}$ - min_{ij}. On a alors :



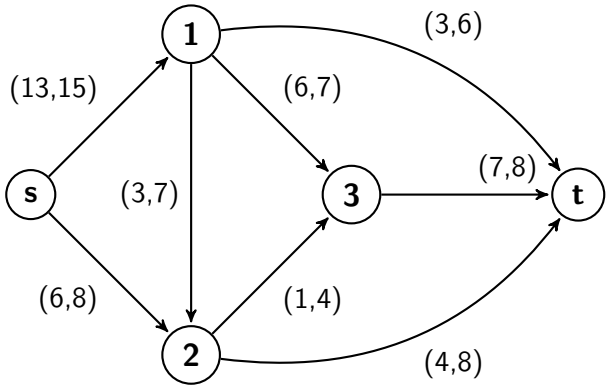
Les demandes des sommets (hormis s et t) sont ajustées par rapport aux capacités minimum des arcs. Ainsi, la demande d'un tel sommet i est telle que : $d_i = \sum min\ sortants\ i$ - $\sum min\ entrants\ i$.

La demande du sommet s est alors égale à $\sum demandes\ négatives$.

La demande du sommet t est alors égale à $\sum demandes\ positives$.

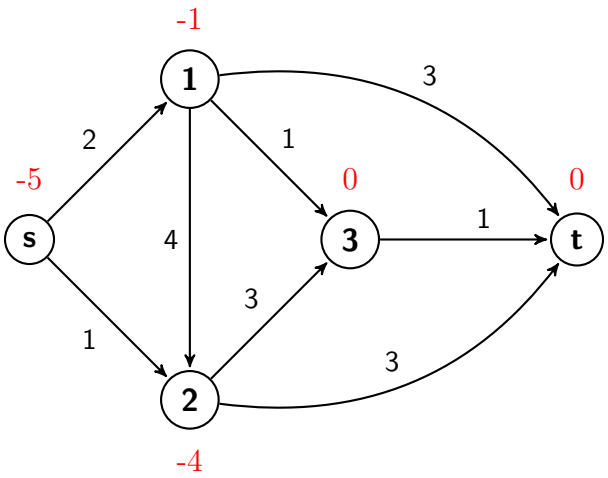
En reportant les minimums sur les demandes, les demandes sont nécessairement satisfaites (flot circulation) : on retrouve la configuration de notre réseau R' de l'étape 2, intégrant en plus une contrainte sur le flot circulant qui ne peut pas descendre en dessous de la borne inférieure (prise en compte dans la demande), ni dépasser la capacité maximum (nouvelle capacité bornée).

Soit pour un réseau R (les capacités ont ici été modifiées pour répondre au besoin du problème) :



Réseau R (chaque arc ij est valué par un ensemble (x,y) où x = min_{ij} et y = c_{ij})

On obtient le réseau R' suivant :



Réseau R'

La complexité de cette étape est en **O(n+m)**, où n représente le nombre de sommets et m le nombre d'arcs, puisque chaque sommet se voit attribuer une nouvelle demande, fonction des valeurs minimum des arcs auxquels ce sommet est adjacent. Il faut donc, pour chaque sommet, ajuster la valeur de sa demande en parcourant ses différents arcs, et modifier les capacités de ces derniers.

2.4 Etape 4

Cette quatrième étape vise à formuler le problème Arrondis-2D en un problème d'arc-circulation. Notre entrée est ici une matrice M à n lignes et m colonnes, à valeur réelles.

Une première étape consiste tout d'abord à transformer M, telles que ses valeurs soient comprises entre une borne inférieure et supérieure. Dans l'exemple qui nous est fourni, on obtient la matrice M' donnée ci-contre, où la dernière ligne (respectivement colonne) représente les totaux des valeurs sur la ligne (respectivement colonne) traitée.

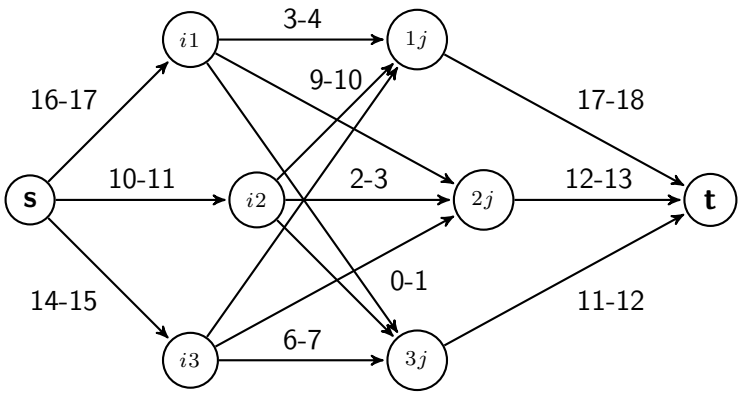
3-4	6-7	7-8	17-18
9-10	2-3	0-1	12-13
3-4	1-2	6-7	11-12
16-17	10-11	14-15	

Matrice M'

Dans la mesure où la valeur du flot entrant est nécessairement égale à la valeur du flot sortant, l'astuce consiste ici à mapper le sommet s (source) aux sommets correspondant aux sommes en colonnes (dernière ligne) et le sommet t (puits) aux sommets correspondant aux sommes en lignes (dernière colonne), de telle sorte que la valeur du flot sortant de s soit égale à la valeur du flot arrivant à t.

Autrement dit, dans notre exemple, s pointe vers trois sommets distincts : si_1, si_2 et si_3, et trois autres sommets distincts pointent vers t : s_{1j}, s_{2j} et s_{3j}. Dans cette nouvelle

configuration, chacun des sommets du premier ensemble pointe vers chacun des sommets du deuxième ensemble, tel que présenté ci-contre :



NB: Pour des raisons de lisibilité, toutes les capacités n'ont pas été représentées.

Chaque case du tableau est ainsi représentée par un arc valué, dont la valuation est bornée entre deux arrondis (inférieur et supérieur). Dans la mesure où chaque case est mise en correspondance avec une ligne et une colonne données, et par extension entre la source (dernière ligne) et le puits (dernière colonne), la condition de conservation du flot est nécessairement respectée.

3 Implémentation

L'implémentation du projet devant être réalisée en Java, nous avons décidé d'utiliser une approche objet, dont les apports et répercussions sur les algorithmes développés seront ici présentés.

Pour compiler le projet, le script shell **compile.sh** doit être éxecuté à la racine du projet. Un jar précompilé **projet.jar** est fourni.

Pour lancer le programme il faut utiliser cette commande : *java -jar projet.jar path [-debug].* path étant le chemin vers le fichier de tableau et [-debug] facultatif (affichages intermédiaires). Plusieurs fichiers d'exemple sont disponibles dans le dossier *data*. De plus un fichier python permet de générer aléatoirement un nouveau tableau : *python generateData.py 20 15*. Cette commande permet de générer un tableau 20x15 dans la sortie standard, redirigeable vers un fichier.

3.1 Structure de données

3.1.1 Classes de base

Pour stocker notre graphe, nous avons utilisé une conception objet assez classique, dont voici le diagramme UML non exhaustif :

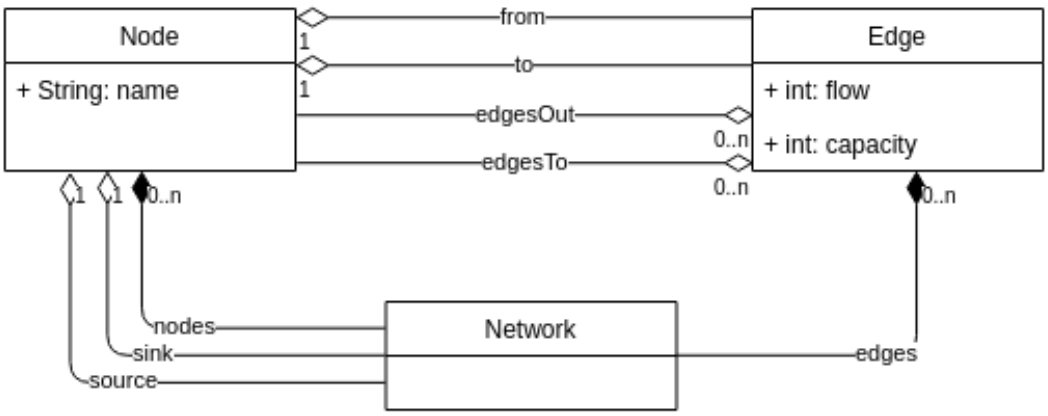


Diagramme UML non exhaustif de Node, Edge et Network

Une classe a été créée pour chaque type d'élément : Noeud (Node), Arc (Edge) et Réseau (Network). Tout les arcs et les noeuds sont stockés dans le réseau. Le réseau contient aussi les références vers les noeuds *source* et *puits*.

Cependant, il existe aussi des liens directs entre les noeuds et leurs arcs associés et vice-versa. Il faut noter que les listes d'objets sont implémentées ici à travers des Maps, qui associent une clé à une valeur. Par exemple pour la liste *nodes* dans Network, la clé est le nom du noeud. Dans Node, les listes d'arcs sortants et d'arcs sortants ont pour clé le nom du noeud opposé. Ce système de clés permet d'optimiser certains algorithmes et nous sera utile, comme expliqué par la suite.

Toutes ces références entre les objets peuvent sembler lourdes mais elles ont, à notre avis, plus d'avantages que d'inconvénients :

D'une part, si l'on veut reconstruire un graphe identique, il faut reconstruire toutes les références. De plus, cette implémentation requiert légèrement plus de mémoire. Ce désavantage est grandement limité puisque qu'aucun objet n'est dupliqué, uniquement leurs références.

D'autre part, nous avons accès à différentes représentations d'un graphe en machine dans la même structure de données. Dans Network, nous avons une liste des noeuds et des arcs.

Dans Node, une liste des prédécesseurs et des successeurs. Enfin dans Edge, nous avons accès aux noeuds entrants et sortants.

3.1.2 Classes dérivées

D'une étape à l'autre, le type de graphe peut changer, c'est-à-dire qu'il peut porter des informations supplémentaires sur ses noeuds ou ses arcs. Pour représenter ceci, et garder un code plutôt factorisé, chaque nouveau type est un héritier de la classe de base. Les types de graphes sont donc représentés à travers différentes classes. Les méthodes de conversion d'une étape à l'autre sont donc codées dans les classes associées, soit en *static*, accessible partout, soit en méthode de classe.

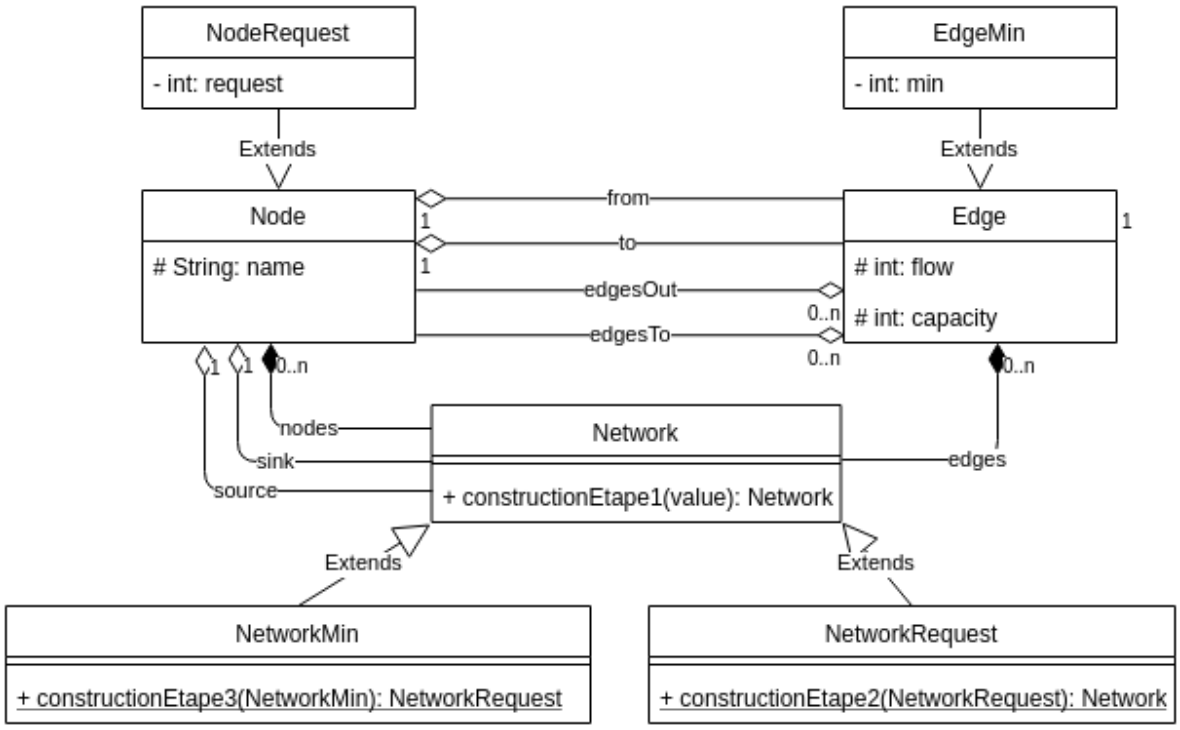


Diagramme UML non exhaustif des classes dérivées

3.2 Algorithmes et complexité

3.2.1 Algorithme général

L'algorithme général consiste à exécuter chaque étape consécutivement, du chargement du fichier jusqu'au préflot sur le graphe final. Il faut cependant pour l'étape 1 tester plusieurs valeurs de flot afin d'obtenir des totaux satisfaisants.

Note : On observe que la valeur de flot valide ne dépasse jamais la plus grande des dimensions du tableau, et s'en rapproche même. Il vaut donc mieux partir de la valeur maximum et s'arrêter quand un résultat valide est trouvé pour obtenir une meilleure complexité au mieux.

```
Main( String cheminGrille )
    grille = ChargerGrille(cheminGrille)
    G1 = ConstructionEtape4(grille)
    G2 = ConstructionEtape3(G1)
    Flot flot = max(nbColonnes, nbLignes)
    Tant que (VerifierTotaux(G3)) et (f >= 0) faire :
            G3 = ConstructionEtape1(ConstructionEtape2(G2, flot))
            Preflot(G3)
            flot = flot + 1
    Fin
Fin
```

3.2.2 Construction de l'étape 4

```
ConstructionEtape4( Grille g ): GrapheMinimums
    G = Graphe
    G.ajouterNoeud(source)
    G.ajouterNoeud(puit)
    Pour chaque total ti en ligne, i de 0 a numLignes:
        G.ajouterNoeud(ti)
        G.ajouterArc(source->ti, min=round(ti), capacite=round(ti)+1)
    Fin
    Pour chaque total tj en ligne, j de 0 a numColonnes:
        G.ajouterNoeud(tj)
        G.ajouterArc(tj->puit, min=round(tj), capacite=round(tj)+1)
    Fin
    Pour chaque case ij, i de 0 a numLignes, j de 0 a numColonnes:
        G.ajouterArc(ti->tj, min=round(ij), capacite=round(ij)+1)
    Fin
    retourner(G)
Fin
```

Complexité : O(l+h+lh)=O(lh) où l : taille en ligne, h : taille en hauteur

3.2.3 Construction de l'étape 3

```
ConstructionEtape3( GrapheMinimums G ): GrapheDemandes
    G' = Graphe
    Pour chaque Noeud n de G:
```

```
        G'.ajouterNoeud(n, demande=0)
    Fin
    Pour chaque NoeudDemande u de G:
        Pour chaque NoeudDemande v successeur de u:
            G'.ajouterArc(u -> v, capacite=capacite(u,v)-min(u,v))
            demande(u) = demande(u)+min(u,v)
            demande(v) = demande(v)-min(u,v)
        Fin
    Fin
    retourner(G')
Fin
```

Reconstruction du graphe : complexité en **O(n+ns)=O(n+e)** où n : noeuds, s : successeurs de n, e : arcs.

Note : Ici, la source et le puits sont initialisés à des valeurs $-\infty$ et $+\infty$, contrairement à la description précédente. Le résultat est le même et permet une implémentation plus légère.

3.2.4 Construction de l'étape 2

```
ConstructionEtape2( GrapheDemandes G ): Graphe
    G' = Graphe
    s', t' = Noeud
    G'.definirSource(s'), G'.definirPuit(t')
    Pour chaque NoeudDemande n de G:
        G'.ajouterNoeud(n)
        Si demande(n)<0 Alors G'.ajouterArc(s' -> n, capacite=-demande(n))
        Sinon G'.ajouterArc(n -> t', capacite=demande(n))
    Fin
    Pour chaque Noeud u de G:
        Pour chaque Noeud v successeur de u:
            G'.ajouterArc(u -> v, capacite=capacite(u,v))
        Fin
    Fin
    retourner(G')
Fin
```

Reconstruction du graphe : **O(n+ns)=O(n+e)**. Même complexité que l'étape 3.

3.2.5 Construction de l'étape 1

```
ConstructionEtape1( Graphe G, int valeur ): Graphe
    G.changerArc(s' -> s, capacite)
    retourner(G)
Fin
```

Cette fonction consiste juste à changer la capacité de l'arc s' vers s. La complexité est donc constante, en **O(1)**.

3.2.6 Avantages et possibles améliorations de l'implémentation

Nous avons déjà vu que notre modèle objet permet une meilleure complexité. En effet en cumulant les structures de données, nous sacrifions un peu d'espace mémoire au profit d'optimisations.

La complexité générale est la somme des complexités de toutes les étapes. Cependant les étapes 2, 1, le préflot (O(n+e)) et la récupération du résultat (O(lh+l+h) = O(lh)) sont répétées au maximum max(lignes, colonnes). La complexité générale reste donc polynomiale en O(1 + lh + 2(n+e) + max(l,h)(n+e + 1 + lh)). Elle pourrait sans doute être améliorée en factorisant du code entre les étapes, et surtout en changeant la méthode pour récupérer le résultat sans réinitialiser tout le tableau à chaque itération.

Enfin, même si notre implémentation est opérationnelle pour la quasi-totalité des exemples, il peut arriver que certains totaux en ligne et en colonnes ne soient pas suffisants (uniquement dans le cas de tableaux dont les tailles en ligne et en colonne sont très grandes et diffèrent).