Rapport de graphes

Binarisation d’image

Table des matières

[Préambule 2](#_Toc26546621)

[Le projet 2](#_Toc26546622)

[Questions de l’exercice 2](#_Toc26546623)

[Q1. 2](#_Toc26546624)

[Q2. 3](#_Toc26546625)

[Q3. 3](#_Toc26546626)

[Q4. 3](#_Toc26546627)

[ResoudreBinIm 4](#_Toc26546628)

[LireEtDecomposerFichierTexte 4](#_Toc26546629)

[ConstructionReseau 5](#_Toc26546630)

[CalculCoupeMin 5](#_Toc26546631)

[Le programme Java 7](#_Toc26546632)

[Compilation 7](#_Toc26546633)

[Exécution 7](#_Toc26546634)

[Quelques fichiers utilisables 7](#_Toc26546635)

# Préambule

Il peut être intéressant de pouvoir distinguer deux parties dans une image (par exemple : l’arrière-plan du premier plan, la partie sanguine de la partie muqueuse, etc…). Utilisée dans de nombreux domaines, du traitement d’images en photographie ou en médecine ou en reconstitution d’images bruitées, la binarisation d’images est une technique bien répandue.

De nombreuses recherches ont été réalisées dans ce domaine et les résultats nous poussent à étudier ces techniques à travers l’utilisation de graphes. De nombreux algorithmes ont été utilisés pour résoudre ce problème, nous verront l’utilisation de l’un d’eux dans ce rapport : l’algorithme de recherche de flot maximum de Ford-Fulkerson.

# Le projet

Le projet donné ici se focalise sur le traitement donné à une image prétraitée. En effet, on suppose « l’image » en tant que fichier texte dont on a décomposé les pixels et les liens entre les pixels pour qu’ils aient des valeurs spécifiques au travail. Parmi ces valeurs, on va retrouver un « score » de premier-plan et de second-plan, et un coût de séparation entre 2 pixels. En simplifiant, on peut voir la première valeur comme étant sa luminosité ou son obscurité, et la seconde valeur comme l’écart de luminosité entre un pixel et ses voisins.

Ici le fichier que nous utilisons nous est fourni au préalable comme étant un fichier où ces valeurs ont été calculés par un traitement informatique indépendant.

Le résultat est l’apparition d’une image en uniquement 2 couleurs (noir et blanc par exemple) qui nous permet de distinguer directement l’objet de l’image initiale que nous souhaitons observer.

# Questions de l’exercice

## Q1.

En notant et en le faisant intervenir dans les calculs, montrer que maximiser équivaut à minimiser :

Réponse :

En effet, si

Nous remarquons donc que , or est « constant » donc maximiser revient à minimiser .

## Q2.

Construire un réseau de transport R (graphe oriente) représentant les pixels et leurs voisinages. Ajouter d’autres arcs selon les besoins et définir des capacités sur les arcs de sorte que toute coupe dans le réseau ait une capacité donnée par une formule de type (2) [].

Nous sommes dans un problème de flot, donc nous avons besoin d’une source et d’un puit, de sommets et d’arcs avec des capacités.

Nous supposerons le réseau défini comme cela : Chaque « pixel » de l’image est représenté par un sommet (i,j) relié avec au maximum 4 voisins qui sont ses pixels voisins (uniquement 2 pour un pixel du coin, ou 3 pour un pixel d’un bord de l’image). La valeur des arcs reliant deux pixels est la valeur avec (i,j) et (k,l) les deux pixels. Nous ajoutons ensuite une source S, sommet virtuel, reliée à chaque pixel par un arc orienté S→(i,j) dont la capacité est la valeur du pixel, et un puit T, sommet virtuel également, dont chaque pixel sera relié par un arc orienté (i,j)→T dont la capacité est la valeur du pixel.

Nous chercherons donc à trouver les valeurs spécifiques d’un graphe de flots (capacité maximum, coupe minimum, …) sur ce graphe avec la source S et le puit T.

## Q3.

Justifier l’affirmation : le problème de binarisation d’image a une solution si et seulement si dans le réseau R la paire est une coupe de capacite minimum.

Réponse :

La construction du graphe étant réalisée en suivant la formule de , la paire indique la meilleure division du réseau. Or si une paire P’ de capacité inférieure peut être trouvée, cela signifie que la solution n’est pas optimale.

On peut se retrouver dans le cas où aucune paire P n’est trouvée, c’est uniquement le cas avec un graphe avec des valeurs irrationnelles.

## Q4.

Ecrire un programme pour résoudre le problème BINIM utilisant l’approche proposée. Pour calculer la coupe de capacite minimum, utiliser le théorème MaxFlow-MinCut et implémenter un algorithme de flot maximum. Attention, la coupe de capacite minimum est facilement calculable si vous avez implémenté l’algorithme de Ford-Fulkerson (voir cours), mais l’algorithme de préflots est plus simple à implémenter.

Le programme implémentera, parmi d’autres méthodes, les méthodes suivantes :

1. ConstructionReseau qui construit le réseau de transport R adapté au problème BINIM.

2. CalculFlotMax qui calcule le flot maximum pour un réseau de transport donné.

3. CalculCoupeMin qui calcule la coupe de capacite minimum pour un réseau de transport donné.

4. ResoudreBinIm qui résout le problème BINIM.

### ResoudreBinIm

Nous avons donc développé un programme en Java répondant à ce problème. Voici son fonctionnement global :

#### Pseudo-code :

Def ResoudreBinIm :

Valeurs = LireEtDecomposerFichierTexte(cheminVersLeFichierTexte)

Graphe = ConstructionReseau(Valeurs)

FlotMaximum = CalculFlotMax(Graphe)

{CoupeX, CoupeY} = CalculCoupeMin(Graphe)

Affichage(FlotMaximum, GraphDivise)

Fin

Nous récupérons les informations utiles dans le fichier texte puis les transformons en données exploitables par le programme. Avec ces données, nous construisons un graphe de flots. Depuis ici, il est possible de calculer le flot maximum et de récupérer les deux parties de la coupe minimum. Il ne nous reste plus qu’à afficher les résultats à l’utilisateur.

#### Complexité :

On une complexité d’après les résultats vus dans les lignes suivantes.

Nous allons entrer plus dans les détails de chacune de ces fonctions :

### LireEtDecomposerFichierTexte

Premièrement, la fonction pour lire et décomposer le fichier texte en différentes valeurs du texte (les valeurs Aij (), Bij (), Phorizontal et Pvertical () selon le lien entre les pixels) :

#### Pseudo-code :

Def LireEtDecomposerFichierTexte(chemin) :

Lignes = ConvertirEnDecimal(SeparerChaqueMot(LireFichier(chemin)))

N = Lignes[1][1]

M = Lignes[1][2]

LigneCourante = 3

Pour i allant de 1 à N :

Pour j allant de 1 à M :

Aij[i][j] = Lignes[i + LigneCourante][j]

Bij[i][j] = Lignes[i + LigneCourante + N+1][j]

Fin Pour

Fin Pour

LigneCourante = 2 \* N + 5 # Ligne précédant les Phorizontaux

Pour i allant de 1 à N :

Pour j allant de 1 à M-1 :

Phorizontal[i][j] = Lignes[i + LigneCourante][j]

Fin Pour

Fin Pour

LigneCourante = 3 \* N + 6 # Ligne précédant les Pverticaux

Pour i allant de 1 à N - 1 :

Pour j allant de 1 à M :

Pvertical[i][j] = Lignes[i + LigneCourante][j]

Fin Pour

Fin Pour

Retourner {N, M, Aij, Bij, Phorizontal, Pvertical}

Fin

La première étape est de récupérer chaque ligne de notre fichier, puis de séparer chaque valeur (séparées par des espaces dans notre cas) puis de transformer chaque valeur en valeur décimale pour que notre programme puisse l’exploiter. Ensuite nous parcourons toutes les lignes pour lire les valeurs Aij, Bij, Phorizontales et Pverticales. Enfin, on renvoie ces valeurs.

#### Complexité :

On a donc une complexité

### ConstructionReseau

On passe à la fonction ConstructionReseau :

Def ConstructionReseau(valeurs) :

Graphe = CreationGrille(valeurs.N, valeurs.M)

Ajouter un sommet S à Graphe

Ajouter un sommet T à Graphe

Pour i allant de 1 à valeurs.N :

Pour j allant de 1 à valeurs.M :

Si j < valeurs.M :

ValeurArc( (i,j), (i, j+1) ) = valeurs.Phorizontal( (i,j) )

ValeurArc( (i,j+1), (i, j) ) = valeurs.Phorizontal( (i,j) )

Fin Si

Si I < valeurs.N :

ValeurArc( (i,j), (i+1, j) ) = valeurs.Pvertical( (i,j) )

ValeurArc( (i+1,j), (i, j) ) = valeurs.Pvertical( (i,j) )

Fin Si

ValeurArc( S, (i,j) ) = valeurs.Bij

ValeurArc( (i,j), T ) = valeurs.Aij

Fin Pour

Fin Pour

Retourner Graphe

Fin

On crée un graphe vide auquel on ajoute la source S et le puit T.

Pour chaque sommet (pixel) à ajouter au graphe, on définit les arcs vers et depuis le sommet à sa droite (Phorizontal) et les arcs vers et depuis le sommet en dessous. De plus, on ajoute un arc de S à ce pixel (Bij) et du pixel vers T (Aij).

Enfin, on retourne ce nouveau graphe.

#### Complexité :

On a donc une complexité de

### CalculCoupeMin

Puis la fonction pour calculer la coupe minimum du graphe :

Def CalculCoupeMin(Graphe, S, T) :

NbSommets = NombreDeSommets(Graphe)

X = Bool[NbSommets]

Residuel = Graphe

Predecesseurs = ParcoursLargeurDepuis(S, Residuel)

Tant Que Predecesseurs a rencontré T :

Flot = +∞;

V = T

Tant que V != S

Flow = Min(flow, Residuel[ Predecesseurs[V] ][V]);

V = Predecesseur[V]

Fin Tant Que

V = T

Tant Que V != S

Residuel[Predecesseurs[V] ][V] -= Flow

Residuel[V][Predecesseurs[V]] += Flow

Fin Tant Que

Predecesseurs = ParcoursLargeurDepuis(S, Residuel)

Fin Tant Que

X = ParcoursLargeurDepuis(S, Residuel)

Y = !X

Retourner {X, Y}

Fin

Def ParcoursLargeurDepuis(Depart, Graphe) :

Pour tout S sommet de Graphe :

Marquer S « non-visité »

Fin Pour

Liste = []

Ajouter Depart à Liste

Marquer Depart « visité »

Predecesseur de Depart = NIL

Tant Que Liste n’est pas vide :

U = Premier élément de Liste

Supprimer le premier élément de Liste

Pour tout V successeur de U non-visité  :

Ajouter V à Liste

Enregistrer U comme prédécesseur de V

Marquer V « visité »

Fin Pour

Fin Tant Que

Retourner liste de prédécesseurs

Fin

Ici, nous allons utiliser la fonction ParcoursLargeurDepuis pour calculer la coupe minimale. En commençant depuis le sommet de départ qu’on marquera comme « visité » et que l’on entre dans la liste de sommets « en attente ». Ensuite, tant qu’il y a des sommets à traiter dans la liste, on considère le premier sommet de la liste comme le sommet courant. On ajoute ses successeurs dans la liste des sommets « en attente », on indique que leur prédécesseur est le sommet courant et on les marque comme « visités ». L’opération se répète tant que nous n’avons pas visité tous les sommets accessibles depuis le sommet de départ. Enfin on retourne la liste des prédécesseurs de chaque sommet visité.

Pour le calcul de la coupe minimum, On va tout d’abord récupérer la liste des prédécesseurs avec la fonction de parcours, et vérifier si le puit T est accessible depuis S. Tant que ça l’est, on trouve un chemin depuis T dont on calcule le flot, on réduit la capacité des arcs de ce chemin selon la capacité minimum trouvée. On réduit jusqu’à ce que T ne soit plus accessible depuis S et on renvoie la liste des successeurs de S, et le reste.

#### Complexité :

Le parcours en Largeur se fait avec une complexité avec S le nombre de sommets (soit ).

Le calcul de la coupe minimum a une complexité .

## Le programme Java

Le programme est disponible sous « VenteHartley.zip ».

### Compilation

Pour compiler le programme, se mettre à la racine du dossier, puis :

> cd bin

> javac -cp ../src/main/ResolutionGraphes.java ../src/main/\*.java -d

### Exécution

Pour exécuter le programme (toujours dans le dossier « bin »), la commande standard est :

> java ResolutionGraphes -v fichier.txt

Plusieurs options sont disponibles en ajoutant un mot clé avant le nom du fichier. La liste des options est disponible avec la commande :

> java ResolutionGraphes

Par exemple on peut utiliser l’option -image=monImage.png pour afficher le résultat du découpage.

### Quelques fichiers utilisables

Les fichiers sont disponibles à la racine du projet sous les noms de :

* exemple\_cours.txt, JeuDeDonneesDemo1.txt, JeuDeDonneesDemo2.txt (exemples disponibles sous Madoc)
* message.txt (intéressant pour essayer l’option « -image » du programme)
* graph\_mushroom.txt (un exemple de taille plutôt conséquente)