Mini-Projet d'Algorithmique
Tomographie discrète

3I003

Licence d'Informatique L3

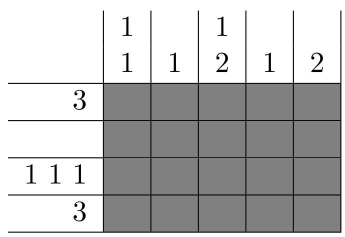
Université Pierre et Marie Curie

Année universitaire 2017-2018

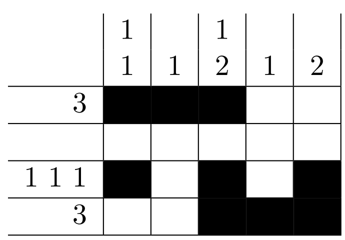
1 Introduction

La tomographie est une technique utilisée en imagerie (médecine, géophysique) qui permet de reconstruire un objet à partir d'une série de mesures effectuées par tranches depuis l'extérieur de cet objet : c'est par exemple le cas de la technique IRM employée en médecine (Imagerie par Résonance Magnétique). La tomographie est dite discrète si l'objet est représenté par des pixels ou *cases* qui composent alors un objet matriciel. Dans ce projet, nous nous intéressons à la reconstruction par tomographie discrète d'un objet en 2 dimensions composé de cases *noires* ou *blanches*.

Etant donnée une grille de n lignes numérotées de 0 à $n-1$ et m colonnes numérotées de 0 à $m-1$, on désire colorier chacune des nm cases en blanc ou en noir. A chaque ligne est associée une séquence d'entiers représentant les longueurs des blocs de cases noires de la ligne. De même, à chaque colonne est associée une séquence d'entiers représentant les longueurs des blocs de cases noires de la colonne.

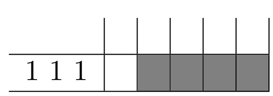


Dans l'exemple ci-dessus, le 3 de la première ligne signifie que celle-ci doit contenir un bloc de trois cases noires consécutives. La séquence $(1,2)$ de la troisième colonne signifie qu'elle doit contenir deux blocs : le premier bloc composé d'une unique case noire et le deuxième bloc de deux cases noires. Les blocs doivent être séparés d'au moins une case blanche. Un coloriage possible pour cet exemple est :



Etant donnée une grille ainsi que des séquences associées à ses lignes/colonnes, le problème de tomographie discrète consiste à déterminer s'il existe un coloriage respectant les contraintes et, le cas échéant, à en produire un.

Dans l'exemple de la matrice $1 \times m$ (appelée vecteur) ci-dessous, si l'on impose que la première case soit coloriée en blanc, alors on peut remarquer qu'il n'existe pas de coloriage possible : en effet, la case blanche de la colonne 0 empêche de placer ensuite trois blocs d'une case noire séparés d'au moins une case blanche.



Le travail se divise en une **partie théorique** et une **partie expérimentale**. La partie théorique permet d'établir et d'analyser plusieurs algorithmes ainsi que leurs complexités respectives. La partie expérimentale porte sur la mise en œuvre de ces algorithmes et la vérification expérimentale de leurs complexités respectives.

2 Partie théorique

Une instance du problème de tomographie discrète est donnée par
- n le nombre de lignes.
- m le nombre de colonnes.
- $(L_1^i, \ldots, L_{l_i}^i)$ une séquence de l_i nombres entiers correspondant à des blocs à colorier sur la ligne $i \in \{0, \ldots, n-1\}$.
- $(C_1^j, \ldots, C_{c_j}^j)$ une séquence de c_j nombres entiers correspondant à des blocs à colorier sur la colonne $j \in \{0, \ldots, m-1\}$.
- une matrice M de taille $n \times m$ telle qu'une case $M[i][j] \in \{0, 1, 2\}$ où 1 désigne "blanc", 2 "noir" et 0 "libre". Le statut "libre" désigne le fait que la case n'est pas coloriée au départ.

Si toutes les cases de la matrice M sont à valeur 0, on dit que la grille est libre (c'est le cas le plus fréquent). Dans l'exemple du bas de la page précedente, la séquence est (L_1^0, L_2^0, L_3^0) avec $L_1^0 = L_1^0 = L_1^0 = 1$.

Un coloriage de la matrice M d'une instance consiste à donner la couleur 1 (blanc) ou 2 (noir) aux cases libres de l'instance, de façon que chaque ligne $i \in \{0, \ldots, n-1\}$ (resp. chaque colonne $j \in \{0, \ldots, m-1\}$) comporte un premier bloc de L_1^i (resp. C_1^j) cases noires consécutives, puis un deuxième de L_2^i (resp. C_2^j) cases noires, etc. Il n'y a pas d'autres cases noires dans la ligne/colonne et les blocs doivent être séparés d'au moins une case blanche. Avant le premier ou après le dernier bloc, il peut y avoir aucune, une ou plusieurs cases blanches.

Noter que $l_i = 0$ (resp. $c_j = 0$) correspond au fait que la ligne i (resp. colonne j) peut être librement coloriée.

Le problème de tomographie discrète est en fait divisé en deux problèmes :
Décision : déterminer si une instance possède ou non un coloriage.
Coloriage : s'il en existe un, fournir un coloriage.

Ces deux problèmes sont NP-difficiles, ce qui implique que l'on ne connaît pas d'algorithme de complexité polynomiale en n et m pour les résoudre quelle que soit l'instance considérée. Pour le problème Coloriage, nous allons étudier un algorithme d'énumération. Pour le problème Décision, nous allons étudier un algorithme de programmation dynamique pour le cas particulier du vecteur. Enfin nous nous intéresserons à une méthode approchée afin de colorier rapidement un maximum de cases d'une instance.

2.1 Enumération des solutions

On suppose que l'on dispose des fonctions :
- `Compare_seq_ligne(i)` en $O(m)$

- Compare_seq_col(j) en $O(n)$
qui, étant donnée une ligne i (respectivement colonne j) entièrement coloriée de M, vérifie si le coloriage respecte bien la séquence L^i (respectivement C^j) de la ligne (respectivement colonne).

Considérons la fonction récursive `Enumeration_Rec` suivante. On admet que l'appel [`Enumeration_Rec`$(0,1)$ ou `Enumeration_Rec`$(0,2)$] retourne vrai ou faux en réponse au problème Décision ; et le cas échéant, répond au problème Coloriage en remplissant la matrice M d'un coloriage.
Dans ce pseudo-code :

- on suppose que les données de l'instance sont des variables globales.
- le paramètre $k \in \{0, \ldots, nm-1\}$ représente la numérotation de la $k^{\text{ième}}$ case dans une lecture de gauche à droite, de haut en bas.
- le paramètre c représente la couleur testée (1 ou 2) pour la case k.

```
Algorithme Enumeration_Rec(k,c) : booléen
 ok: booléen                                    -- passe à faux si non coloriable
 raz: booléen                                   -- demande la remise à 0 de la case
 i, j: entier
 i ← ⌊k/m⌋                                      -- ligne de la case k
 j ← k mod m                                    -- colonne de la case k
 Si M[i][j] = 0 Alors                           -- Si case libre dans l'instance
       M[i][j] ← c                              -- On tente la couleur c
       raz ← vrai
 Sinon                                          -- Si case déjà coloriée dans l'instance
       Si M[i][j] ≠ c Alors Retourner faux      -- Si déjà colorié différement
       Sinon raz ← faux
 FinSi
 ok ← vrai
 Si i = n − 1 Alors ok ← compare_seq_col(j)
 Si ok et j = m − 1 Alors ok ← compare_seq_ligne(i)
 Si ok Alors
       Si i = n − 1 et j = m−1 Alors Retourner vrai
       ok← Enumeration_Rec(k + 1,1) ou Enumeration_Rec(k + 1,2)
 FinSi
 Si (non ok) et raz Alors M[i][j] ← 0
 Retourner ok
```

Question 1
Pour la grille libre suivante, dessiner l'arbre des appels récursifs générés par l'appel [`Enumeration_Rec`$(0,1)$ ou `Enumeration_Rec`$(0,2)$] en indiquant la grille obtenue à chaque appel. Préciser (en une phrase) comment cet arbre est exploré.

	2	
1		
1		

Question 2
Analyser la complexité de l'appel [`Enumeration_Rec`$(0,1)$ ou `Enumeration_Rec`$(0,2)$] en fonction du nombre p de cases libres de M.

2.2 Problème Décision : cas particulier du vecteur

Dans le cas particulier d'une matrice à une seule dimension (un vecteur ligne ou colonne), nous allons prouver que le problème Décision peut être résolu en temps polynomial grâce à un algorithme de programmation dynamique.

Notons V un vecteur de m cases indicées de 0 à $m-1$ et $L = (L_1, \ldots, L_k)$ une séquence de blocs à colorier sur la ligne (les colonnes ont des séquences vides). Etant donnés $j \in \{0, \ldots, m-1\}$ et $l \in \{0, \ldots k\}$, on note $T(j,l)$ la valeur booléenne indiquant s'il est possible de réaliser un coloriage de $V[0 \ldots j]$ en respectant la séquence $(L_1, \ldots, L_l)$ (si $l = 0$, la séquence est vide).

2.2.1 Cas du vecteur libre

Regardons tout d'abord le problème Décision pour un vecteur libre (toutes les cases sont libres). Bien entendu, ce cas très simple peut être résolu beaucoup plus efficacement, mais cette approche par programmation dynamique sera étendue dans la section suivante au cas du vecteur non libre.

Question 3
Si on connaît toutes les valeurs $T(j,l)$ pour $j \in \{0, \ldots, m-1\}$ et $l \in \{0, \ldots k\}$, comment déterminer si le vecteur V peut être colorié en respectant L ?

Question 4
Montrer que $T(j,0)$=vrai pour tout $j \in \{0, \ldots, m-1\}$.

Question 5
Montrer que
- $T(L_1 - 1, 1)$=vrai.
- $T(j,l)$=faux pour tout $l \geq 1$ et $j < L_l - 1$.
- $T(j,l)$=faux pour tout $l \geq 2$ et $j \leq L_l - 1$.
Notez que, pour ce dernier cas, $T(j,l)$=faux même si $j \leq 0$.

Question 6
Si à présent $l \geq 1$ et $j \geq L_l - 1$, donner les expressions de $T(j,l)$ si on fixe $V[j]$ à 1 ou si on fixe $V[j]$ à 0.

Question 7
En déduire une formule de récurrence pour $T(j,l)$.

2.2.2 Cas général d'un vecteur non libre

Nous allons adapter les principes vus précédemment pour le cas où le vecteur est déjà partiellement colorié.

On suppose que l'on possède une fonction `TestSiAucun`(V,j_1, j_2, val) qui renvoie vrai si la valeur val n'apparaît dans aucune des cases d'un vecteur V entre les cases $j1$ et $j2$ (comprises).

On considère l'algorithme suivant où le tableau TT a été défini comme un tableau de tableaux dont chaque case peut prendre trois valeurs {non visité, vrai, faux}. Toutes les cases de ce tableau ont été préalablement initialisées à la valeur "non visité". Au terme de cet algorithme, chaque case $TT[j][l]$ visitée contiendra la valeur booléenne $T(j, l)$ définie dans la section précédente.

On admet que cet algorithme répond bien au problème Décision pour un vecteur libre ou non libre.

```
Algorithme TestVecteur_Rec(V : vecteur, j : entier, l : entier, TT : tableau) : booléen
 c1: booléen                -- Cas où la case j est blanche
 c2: booléen                -- Cas où la case j est noire

 Si (l = 0) Alors Retourner TestSiAucun(V, 0, j, 2)
 Si (l = 1) et (j = L_l − 1) Alors Retourner TestSiAucun(V, 0, j, 1)
 Si (j ≤ L_l − 1) Alors Retourner faux
 Si (TT[j][l] ≠ ‘‘non visité’’) Alors Retourner TT[j][l]
 Si (V[j] = 2) Alors
       c1 ←faux;
 Sinon
       c1 ←TestVecteur_Rec(V, j − 1, l, TT)
 FinSi
 Si (non TestSiAucun(V, j − (L_l − 1), j, 1)) Alors
       c2 ←faux
 Sinon
       Si (V[j − L_l]=2) Alors
             c2 ←faux
       Sinon
             c2 ←TestVecteur_Rec(V, j − L_l − 1, l − 1, TT)
       FinSi
 FinSi
 TT[j][l] ← c1 ou c2
 Return TT[j][l]
```

Question 8

A quoi sert la ligne `Si` ($TT[j][l] \neq$ `‘‘non visité’’`) `Retourne` $TT[j][l]$?

Question 9

On veut prouver que la complexité de la fonction `TestVecteur_Rec` est polynomiale. Proposer pour cela une réécriture de la fonction `TestVecteur_Rec` de manière à ce que les appels récursifs au sein de la nouvelle fonction ne soient effectués que si la case $TT[j][l]$ est égal à "non visité". Cette fonction doit avoir la même complexité que la fonction initiale.

Conclure alors sur leur complexité en fonction de la taille m du vecteur.

3 Partie expérimentale

Dans cette section, nous allons mettre en œuvre les algorithmes proposées dans la partie théorique et proposer une méthode efficace pour déterminer un coloriage partiel.

Structures de données

Il est important de noter qu'il est possible et tout à fait efficace d'utiliser uniquement des tableaux pour les données de l'instance ou les éléments des algorithmes. En effet, toutes ces structures occuperont un espace mémoire en $O(nm)$.

Par exemple, les tailles des séquences L et C peuvent être des tableaux et les séquences L et C peuvent être des tableaux de tableaux (de tailles différentes). De même, le tableau TT de la fonction de programmation dynamique peut être elle-même un tableau de tableaux de la même taille que L ou C.

Instances

Il vous est fourni deux lots d'instances :

- des instances de grilles libres à deux dimensions numérotées : de 0.tom à 16.tom,
- des instances de vecteur-lignes vec_X.tom où X correspond à la taille du vecteur.

Les deux lots d'instances respectent le même format. La première ligne contient les nombres n et m. Elle est suivie de n lignes qui indiquent les séquences correspondantes aux lignes, puis m lignes correspondant aux séquences des colonnes. Ce format est illustré par le fichier 0.tom suivant pour l'instance donnée dans l'introduction.

```
4 5             n=4 et m=5
1   3           séquence d'un seul bloc de 3 cases noires en ligne 1
0               pas de séquences en ligne 2
3   1 1 1       séquence de 3 blocs d'une case noire chacuns en ligne 3
1   3

2   1 1
1   1
2   1 2         un bloc de 1 case noire et un bloc de 2 cases noires en colonne 3
1   1
1   2
```

3.1 Enumeration

Question 10

Implémenter les fonctions Compare_seq_ligne(i) en $O(m)$ et Compare_seq_col(j) en $O(n)$ spécifiées dans la partie théorique.

Implémenter la fonction `Enumeration` décrite dans la partie théorique.

Testez-la sur les instances 0 à 16 (en autorisant une durée maximale d'exécution de 5min). Que constatez-vous ?

3.2 Cas du vecteur

Question 11
En suivant le principe de programmation dynamique décrit dans la fonction `TestVecteur_Rec` de la partie théorique, implémenter les fonctions `TestVecteurLigne_Rec` et `TestVecteurColonne_Rec` pour tester s'il existe un coloriage possible pour une ligne ou une colonne de la matrice (même si elle est partiellement coloriée).

Question 12
Tester votre fonction `TestVecteurLigne_Rec` sur des petites instances réalisables et non réalisables. Pour obtenir des instances non réalisables, fixez une couleur à une ou plusieurs cases d'un vecteur avant d'appeler la fonction.

Question 13
Tester vos fonctions `Enumeration` et `TestVecteurLigne_Rec` sur le lot d'instances de vecteurs en arrêtant l'exécution après 30 minutes si l'exécution est trop longue. Pour chacune des fonctions (sur deux graphiques distincts), tracer une courbe permettant d'évaluer le temps d'exécution CPU en fonction de la taille du vecteur (attention le temps CPU peut être différent du temps d'exécution). Générer ces courbes à l'aide d'un tableur ou d'un outil graphique tel que `xgraphic` ou `gnuplot` (une documentation est en ligne sur le site du module). Vous pourrez ainsi analyser de façon critique la valeur expérimentale obtenue en fonction des complexités théoriques et aussi comparer les performances des algorithmes.

3.3 Propagation

On veut à présent implémenter une méthode de complexité polynomiale pour colorier le maximum de cases avant de lancer à nouveau la méthode `Enumération` sur les cases restantes.

Question 14
Les annexes 1 et 2 donnent le pseudo-code de fonctions `PropagLigne` et `Propagation` qui décrivent comment utiliser les fonctions `TestVecteurLigne` et `TestVecteurColonne` pour déterminer des cases qui doivent nécessairement être coloriées en blanc ou en noir. Implémenter les fonctions `PropagLigne`, `PropagCol` et `Propagation` correspondantes.

Question 15
Tester la fonction `Propagation` en indiquant son temps d'exécution ainsi que le pourcentage de cases coloriées pour les différentes instances.

Question 16
Enchaîner les fonctions `Propagation` et `Enumeration` pour les instances fournies. Que constatez-vous ?

Cahier des charges

- Pour la mise en œuvre algorithmique, vous êtes libres de choisir le langage de programmation que vous utiliserez. Seuls les aspects purement algorithmiques seront abordés avec vos enseignants.
- Un rapport devra être remis au plus tard le **6 décembre 2017 à 23h59** à votre chargé de TD. Ce rapport doit contenir l'analyse théorique (dans laquelle vous prendrez soin de justifier toutes vos réponses), ainsi que l'analyse expérimentale de la complexité des algorithmes. Vous veillerez à commenter les courbes que vous aurez obtenues.
- Vous devez créer un répertoire ayant pour nom `nomBinome1_nomBinome2`, où `nomBinome1` et `nomBinome2` correspondent à vos noms de famille. Si vous êtes seul, donnez votre nom au répertoire. Ce répertoire contiendra votre rapport ainsi que les fichiers sources *commentés* de vos programmes. Compressez ce répertoire sous forme d'un fichier `tgz` (par `tar cvzf nomRepertoire.tgz nomRepertoire`). Envoyez ce fichier en pièce attachée d'un courrier électronique ayant pour sujet `3i003 MiniProjet`, et adressé à votre chargé de TD.
- Le projet se fait par binôme du même groupe.

Annexe 1 : Algorithme Propagation sur une ligne

```
PropagLigne(i : entier, marque : vecteur, nb : entier) : booléen
 j: entier
 c1,c2: booléen
 cptcolor: entier
 cptcolor← 0
 nb← 0
 Pour j de 0 à m-1 Faire
       Si M[i][j] = 0 Alors
            M[i][j] ← 1
            c1 ← TestVecteurLigne_Rec(i)
            M[i][j] ← 2
            c2 ← TestVecteurLigne_Rec(i)
            M[i][j] ← 0
            Si (non c1) et (non c2) Alors Retourner faux
            Si c1 et (non c2) Alors
                 M[i][j] ← 1
                 cptcolor← cptcolor+1
                 Si (non marqueC[j]) Alors
                      marqueC[j]← vrai
                      nb← nb+1
                 FinSi
            FinSi
            Si (non c1) et c2 Alors
                 M[i][j] ← 2
                 cptcolor← cptcolor+1
                 Si (non marqueC[j]) Alors
                      marqueC[j]← vrai
                      nb← nb+1
                 FinSi
            FinSi
       FinSi FinPour
 Retourner vrai
```

Annexe 2 : Algorithme Propagation

```
Propagation() : booléen
 marqueL, marqueC vecteur de booléens
 nbmL, nbmC, nb: entiers
 i,j: entiers
 ok: booléen
 ok← vrai
 Allouer marqueL de n cases à vrai
 Allouer marqueL de m cases à vrai
 nbmL← n
 nbmC← m
 Tant que ok et ((nbmL≠ 0) ou (nbmC≠ 0)) Faire
       i ← 0
       Tant que ok et i < n Faire
            Si marqueL[i] Alors
                 ok=PropagLigne(i,marqueC,nb)
                 nbmC← nbmC + nb
                 marqueL[i]← faux
                 nbmL← nbmL -1
            FinSi
            i ← i + 1
       FinTantQue
       Tant que ok et j < m Faire
            Si marqueC[j] Alors
                 ok=PropagCol(j,marqueL,nb)
                 nbmL← nbmL + nb
                 marqueC[j]← faux
                 nbmC← nbmL -1
            FinSi
            j ← j + 1
       FinTantQue
 FinTantQue
 Retourner ok
```