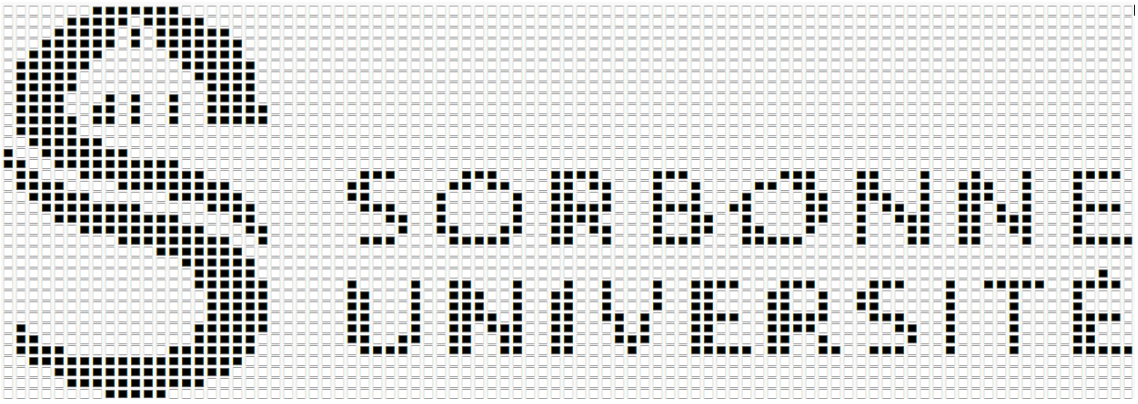
Projet LU3IN003

Un problème de tomographie discrète

CHEMALI Maïssa et DAHER Sarah



Novembre 2023

Table des matières

1 Méthode incomplète de résolution

1.1 Première étape

(Q1) Pour savoir si la ligne l_i peut être coloriée entièrement avec la séquence $(s_1, ..., s_k)$, après avoir calculé tous les `T(j,l)` il suffit juste de vérifier que `T(M-1,k)` est vrai ce qui par définition garantit l'existence d'un tel coloriage.

(Q2) On peut distinguer 4 cas différents :

Cas 1 : si `l = 0` (pas de bloc)
Un coloriage des `j+1` premières cases est possible, en effet, il suffit que les cases soient coloriées en blanc. `T(j,l)` retourne vrai $\forall j \in \{1, ...M-1\}$.

Cas 2 : si `l` $\geq$ `1` (au moins un bloc)
On sait qu'un coloriage est possible uniquement si les `j+1` premières cases contiennent au moins le nombre de cases correspondant aux blocs $(s_1, ...s_l)$ avec une case coloriée en blanc entre chaque bloc (donc `l-1` cases blanches), cela correspond à la formule (*) suivante :

$$j + 1 \geq l - 1 + \sum_{i=1}^{l} s_i$$

Cas 2a : si `j` $< s_l$ `- 1`
Alors, `j + 1` $< s_l$, or on sait d'après (*) que si `T(j,l)` était vrai on aurait j + 1 $\geq s_l$, contradiction. Donc ce cas retourne faux $\forall\ l \geq 1$.

Cas 2b : si `j` $= s_l$ `- 1`
Alors, `j + 1` $= s_l$. L'inégalité (*) se réécrit donc : $s_l \geq l - 1 + \sum_{i=1}^{l} s_i$.
On distingue 2 cas :
- Si `l = 1`, alors $l - 1 + \sum_{i=1}^{l} s_i = 1 - 1 + s_1 = s_1 \leq s_1$, donc on retourne vrai.
- Si `l` $>$ `1`, alors $l - 1 > 0$, donc $l - 1 + \sum_{i=1}^{l} s_i > s_l$, donc l'inégalité (*) n'est pas vérifiée et ce cas retourne faux.
Ainsi, pour `j` $= s_l$ `- 1`, `T(j,l)` retourne vrai uniquement si l = 1.

Cas 2c : `j` $> s_l$ `- 1`

(Q3) `T(j,l)` = `T(j - 1, l)` ou `T(j -` s_l `- 1, l - 1)`.
En effet, si `T(j-1,l)` est vrai, alors on peut colorier les j premières cases avec la séquence entière $(s_1, ..., s_l)$, cela garantit que les `j+1` premières cases peuvent être coloriées avec cette séquence en coloriant la case `(i,j)` en blanc, ainsi `T(j,l)` est vrai. Sinon, si `T(j -` s_l `- 1, l - 1)` est vrai, cela signifie qu'on peut colorier les `j -` s_l premières cases avec la sous-séquence $(s_1, ..., s_{l-1})$, ainsi, on fait entrer le bloc s_l dans les $s_l + 1$ cases suivantes et la case `(i,j)` (dernière case du bloc s_l) est noire (car le bloc s_l est au moins séparé par une case blanche du bloc s_{l-1}).

(Q4) C'est la fonction *ColoriagePossibleLigneOuCol* de notre code.

1.2 Généralisation

(Q5) On distingue 4 cas :

Cas 1 : si `l = 0` (pas de bloc)
T(j, l) retourne vrai uniquement si aucune des j+1 premières cases n'est coloriée en noir.

Cas 2 : si `l` $\geq$ `1` (au moins un bloc)

Cas 2a : si `j` $< s_l$ `- 1`
Était faux, donc reste faux.

Cas 2b : si `j` $= s_l$ `- 1`
Si `l = 1` : alors `j =` s_1 `- 1`. Dans ce cas, on retourne vrai uniquement si aucune des j+1 premières cases n'est coloriée en blanc. En effet, on veut que les s_1 (= j + 1) premières cases soient noires mais s'il y a au moins une case blanche, on aurait donc moins de s_1 cases incolores ou déjà noires ; il serait alors impossible de faire rentrer le bloc s_1 dans les j + 1 premières cases. Au contraire, si on n'a aucune case blanche, on aura exactement s_1 cases déjà noires ou incolores qu'on pourra colorier en noir.
Sinon `l` > 1 : on retourne toujours faux.

Cas 2c : si `j` $> s_l$ `- 1`
- Si la case `(i,j)` n'est pas noire : on regarde si on peut faire entrer la séquence $(s_1, ..., s_l)$ dans les `j` premières cases, c'est-à-dire si `T(j-1, l)` est vrai. Dans ce cas, `(i,j)` doit être blanche.
- Si la case `(i,j)` n'est pas blanche : on vérifie qu'il n'y a aucune case blanche dans les s_l dernières cases (vu qu'on doit positionner le dernier bloc à la fin de la ligne) et que la case j - s_l + 1 n'est pas noire (cette case doit être blanche sinon le dernier bloc sera de longueur s_l + 1). Ces deux conditions validées, on vérifie ensuite si la sous-séquence $(s_1, ..., s_{l-1})$ peut entrer dans les j - s_l premières cases, c'est-à-dire si `T(j-`s_l`-1, l-1)` est vrai. Dans ce cas, `(i,j)` doit être noire.

(Q6) On remarque que le nombre de valeurs `T(j,l)` à calculer est majoré par M $\times$ l. On constate qu'une séquence est composée d'au plus $l = \lceil \frac{M}{2} \rceil$ blocs (en prenant chaque bloc de taille 1 ; entre chaque bloc il y a forcément une case blanche). Alors, le nombre de `T(j,l)` est en $\mathcal{O}(M \times \lceil \frac{M}{2} \rceil)$.
D'autre part, afin de calculer chaque valeur de `T(j,l)` (hors cas de base), on fait appel à la fonction *NonPresenceValeur* qui est en $\mathcal{O}(M)$, en effet, dans le pire des cas on parcourt toutes les cases de la ligne.
Par conséquent, la complexité totale de l'algorithme est en $\mathcal{O}(M \times M \times \lceil \frac{M}{2} \rceil)$, c'est-à-dire en $\boxed{\mathcal{O}(M^3)}$.

(Q7) Ce sont les fonctions *NonPresenceValeur* et *ColoriagePossibleLigneOuColPreremplie* de notre code.

1.3 Propagation

(Q8) On considère l'algorithme proposé. Étudions sa complexité.
Analysons tout d'abord la complexité de *ColoreLig* :
Dans cette fonction, on fait une boucle sur la ligne `i` en $\mathcal{O}(M)$ pour appliquer sur chaque case de la ligne `i` la fonction de la (Q6) qui est en $\mathcal{O}(M^3)$. Ainsi la complexité de cette boucle est en $\mathcal{O}(M^4)$.

Concernant *ColoreCol*, on effectue d'abord une extraction de la colonne `j` de la matrice en $\mathcal{O}(N)$. Ensuite, par symétrie par rapport à *ColoreLig*, on obtient que la complexité totale de *ColoreCol* est en $\mathcal{O}(N^4 + N) = \mathcal{O}(N^4)$.

Revenons à l'étude de la complexité de la fonction *Coloration*.
Initialement, une copie de la grille A, passée en paramètres, est réalisée en $\mathcal{O}(NM)$. Ensuite, pour former les ensembles `LignesAVoir` et `ColonnesAVoir`, des boucles sont effectuées respectivement en $\mathcal{O}(N)$ et $\mathcal{O}(M)$.

Par la suite, une boucle `tant que` est exécutée à la ligne `4` du pseudo-code. Elle s'arrête quand `LignesAVoir` ou `ColonnesAVoir` est vide, c'est-à-dire dès qu'il n'y a plus de cases qui peuvent être coloriées. Étant donné qu'il y a N × M cases, et qu'une fois qu'une case est modifiée, elle ne peut plus l'être à nouveau, la boucle `tant que` de la ligne `4` se répète au plus N × M fois, elle est donc un $\mathcal{O}(N \times M)$.
Cette boucle contient deux boucles non imbriquées, l'une sur les lignes et l'autre sur les colonnes (lignes `5` et `11`), qui se répètent respectivement N et M fois. Les opérations étant symétriques, examinons la complexité sur la première boucle.
1. On fait appel à la fonction *ColoreLig* dont la complexité est en $\mathcal{O}(M^4)$
2. On fait un test en $\mathcal{O}(1)$ sur la valeur de retour de *ColoreLig* :

> **Cas 2.1** Si elle retourne faux, on retourne la matrice vide, ce qui se fait en $\mathcal{O}(NM)$ et on termine.
> **Cas 2.2** Sinon, on continue à exécuter les instructions de la boucle, qui consistent en :
>
> > - une union d'un ensemble (`nouveaux`) de taille en $\mathcal{O}(M)$ avec un autre ensemble (`ColonnesAVoir`) en $\mathcal{O}(M)$. Cette opération est en $\mathcal{O}(M + M) = \mathcal{O}(M)$,
> > - une suppression d'un élément d'un ensemble (`LignesAVoir`), la complexité de cette opération est en $\mathcal{O}(1)$.

Ainsi, on constate que la complexité de cette boucle `pour` de la ligne `5` est en $\mathcal{O}(N) \times \mathcal{O}(M^4 + M) = \mathcal{O}(NM^4)$.
Symétriquement, on a que la complexité de la boucle `pour` de la ligne `11` est en $\mathcal{O}(MN^4)$.

Par conséquent, la complexité de la boucle globale de la ligne `4` est en $\mathcal{O}(NM(MN^4 + NM^4)) = \mathcal{O}(N^5M^2 + M^5N^2)$.

Par ailleurs, on constate qu'en dehors de cette boucle, on effectue un parcours en $\mathcal{O}(NM)$ qui est moins coûteux que la boucle précédente. De même pour les opérations d'initialisation (lignes `2` et `3`) déjà étayées plus haut.

Finalement, on a montré que la complexité de l'algorithme est en $\boxed{\mathcal{O}(N^5M^2 + M^5N^2)}$. Elle est bien polynômiale en N et M.

(Q9) Ce sont les fonctions *ColoreLig, ColoreCol, Coloration, fichierToGrille, propagation* et *AfficherMatriceCouleur*.

(Q10)

Nom de l'instance	Temps de résolution par le programme 1 (en secondes)
1.txt	0.00200
2.txt	0.17001
3.txt	0.10901
4.txt	0.30702
5.txt	0.26602
6.txt	0.59204
7.txt	0.35403
8.txt	0.52722
9.txt	6.04062
10.txt	6.14445

Figure 1 : tableau des temps d'exécution de l'algorithme de propagation

La grille obtenue pour l'instance *9.txt* est la suivante :

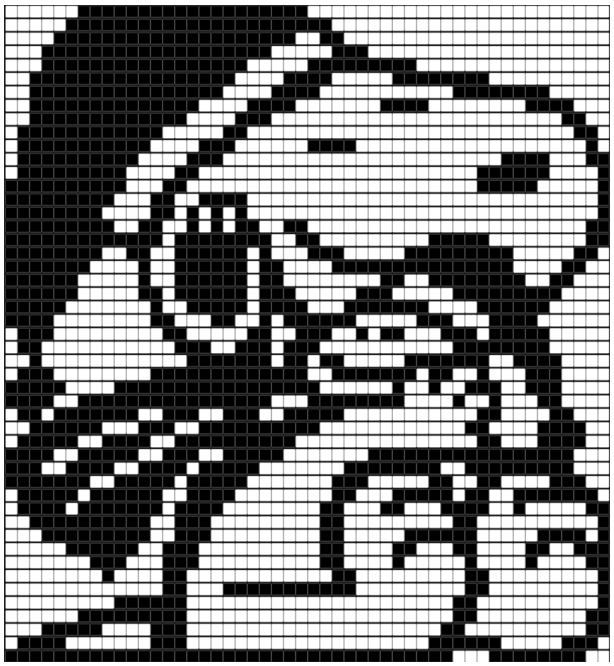


Figure 2 : grille obtenue après application de l'algorithme de propagation sur l'instance 9.txt

(Q11) On exécute le programme sur l'instance `11.txt` fournie. On remarque que la grille retournée est celle dont aucune case n'a été coloriée, et que la fonction retourne `NESAITPAS`. Ceci montre que l'algorithme n'est pas capable d'identifier des cases dans la grille dont la couleur est obligatoire (elles peuvent être blanches et noires), et donc n'a pas pu proposer une solution. En effet, pour mettre un bloc de 2 dans une ligne de 4 cases, on a plusieurs configurations possibles, on ne peut donc rien en déduire (soit les deux premières sont noires, soit les deux centrales, soit les deux dernières). De même pour placer deux blocs d'une case sur une ligne de 4 cases, ou un bloc d'une case sur une colonne de deux cases. Finalement, le programme ne peut pas décider quelles cases doivent être coloriées.

2 Méthode complète de résolution

(Q12) Dans la fonction *EnumRec*, hors cas de base, on fait 2 appels récursifs sur chaque case indéterminée (une fois en la coloriant en blanc et la seconde fois en la coloriant en noir). On a au plus N × M cases à déterminer, donc on a $\mathcal{O}(2^{MN})$ nœuds dans l'arbre des appels récursifs de la fonction. En chaque noeud, on fait appel à la fonction *ColorierEtPropager* qui a la même complexité que *Coloration*. Les complexités des autres opérations sont négligeables en comparaison. Donc la complexité en chaque noeud est en $\mathcal{O}(N^5M^2 + M^5N^2)$. Ainsi la complexité de *EnumRec* est en $\mathcal{O}(2^{MN} \times (N^5M^2 + M^5N^2))$.
La fonction *Enumération* est de même complexité que *EnumRec*, en effet, l'appel à *Coloration* dans *Enumération* est néligeable par rapport à la compléxite exponentielle en N et M.
Par conséquent, la complexité de la fonction *Enumération* est en $\boxed{\mathcal{O}(2^{MN} \times (N^5M^2 + M^5N^2))}$.
Elle est bien exponentielle en N et M.

2.1 Implantation et tests

(Q13) Ce sont les fonctions *colorierEtPropager, trouverProchaineCaseIndeterminee, enumRec, enumeration* et *resolutionComplete*.

(Q14)

Nom de l'instance	Temps de résolution par le programme 1 (en secondes)	Temps de résolution par le programme 2 (en secondes)
1.txt	0.00200	0.00200
2.txt	0.17001	0.17001
3.txt	0.10901	0.12101
4.txt	0.30702	0.30402
5.txt	0.26602	0.25402
6.txt	0.59204	0.58904
7.txt	0.35403	0.35203
8.txt	0.52722	0.54704
9.txt	6.04062	5.83688
10.txt	6.14445	6.83751
11.txt	0.00100 (Pas de solution trouvée)	0.00100
12.txt	0.55304 (Seulement début d'une potentielle solution trouvée)	0.75906
13.txt	0.68805 (Seulement début d'une potentielle solution trouvée)	0.68105
14.txt	0.46903 (Seulement début d'une potentielle solution trouvée)	0.55204
15.txt	0.31702 (Seulement début d'une potentielle solution trouvée)	0.60904
16.txt	1.12909 (Seulement début d'une potentielle solution trouvée)	56.03517

Figure 3 : Tableau des temps obtenus par les deux programmes pour les instances données

Concernant le temps de résolution pris par les programmes, on remarque que les instances `1.txt` à `15.txt` ont le même ordre de grandeur, bien que les instances `11.txt` à `15`.txt ne soient pas complétement résolues par l'algorithme de la partie 1. Par contre, on constate une grande différence par rapport au temps pris par l'instance `16.txt`. Ceci est dû à la complexité exponentielle de l'algorithme d'énumération utilisé dans la partie 2.
D'autre part, on observe que pour les instances `12.txt` à `16.txt`, la méthode de propagation de la section 1 ne résout que partiellement le problème : il reste des cases non coloriées (celles-ci figurent en jaune dans nos grilles). En revanche, la méthode d'énumération résout bien le problème dans

ces cas là. Comme exemple, voici les grilles obtenues par les deux méthodes pour l'instance 15.txt.

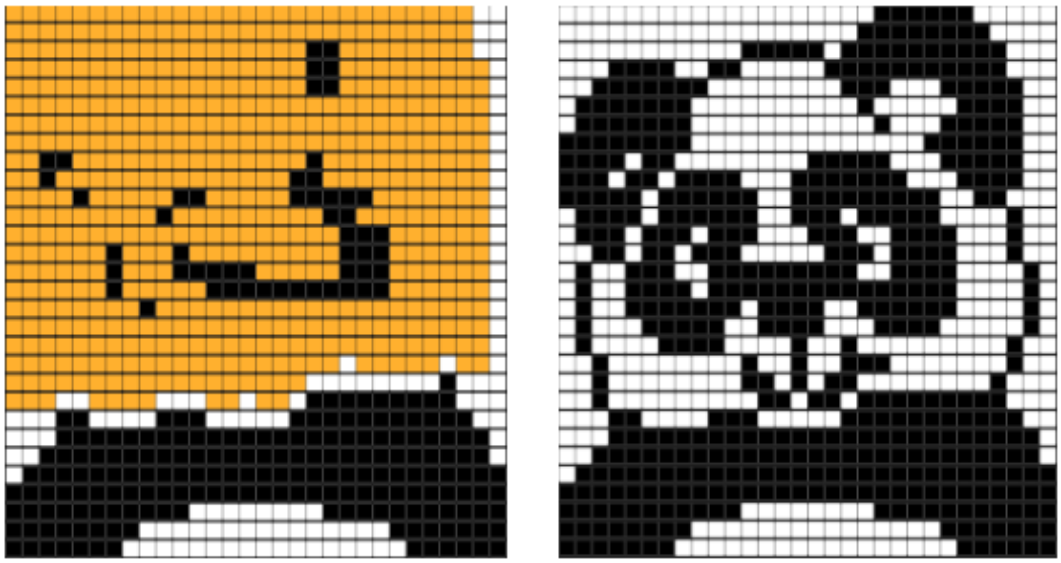


Figure 4 : Grilles obtenues par le programme 1 (à gauche) et par le programme 2 (à droite) pour l'instance 15.txt

Notons également que la fonction de coloration joue un rôle essentiel dans l'optimisation de la fonction d'énumération. Bien que la grille ne soit généralement pas entièrement coloriée après l'application de l'algorithme de coloration (comme le montre la figure ci-dessus), celui-ci a une meilleure complexité que l'algorithme d'énumération. Ainsi, en l'appliquant avant, cela permet de déterminer la couleur d'un certain nombre de cases, et donc d'éviter d'explorer des branches inutiles de l'arbre des appels récursifs, ce qui réduit significativement le nombre de noeuds dans l'arbre. Ainsi, le temps pris par la résolution complète est moins important.