Un problème de tomographie discrète
LU3IN003 : Projet

Antoine Horreard, Ida Rogie

Semptembre - Novembre 2023

Table des matières

1 Introduction **3**

2 Méthode incomplète de résolution **4**
2.1 Première étape 4
2.2 Généralisation 4
2.3 Propagation 5
2.4 Tests 5

3 Méthode complète de résolution **7**
3.1 Implantation et tests 7

1 Introduction

Ce projet se concentre sur la résolution d'un problème de coloriage de grilles rectangulaires, composées de N lignes numérotées de 0 à $N-1$ et M colonnes numérotées de 0 à $M-1$. Chaque cellule de cette grille doit être coloriée en blanc ou en noir et chaque ligne ou colonne est associée à une séquence d'entiers définissant les longueurs des blocs de cases noires.

On cherche ici à développer une solution capable de construire, le cas échéant, un coloriage respectant ces contraintes. Plus précisément, chaque ligne ou colonne se voit attribuer une séquence (potentiellement vide) de nombres entiers strictement positifs $(s_1, s_2, ..., s_k)$, qui définissent la disposition des blocs de cases noires. Le coloriage doit commencer par un premier bloc de s_1 cases noires consécutives, suivi d'un second bloc de s_2 cases noires, et ainsi de suite. Les blocs doivent être séparés d'au moins une case blanche, et il est possible d'avoir des cases blanches avant le premier bloc et/ou après le dernier.

L'objectif de ce projet est de développer une solution répondant à ces contraintes. Pour ce faire, nous avons adopté une approche en deux étapes : une méthode incomplète de résolution, suivie d'une méthode complète permettant d'obtenir une solution finalisée.

2 Méthode incomplète de résolution

Dans cette section, nous appliquons cette méthode pour colorier partiellement des grilles en généralisant ce raisonnement afin de déterminer la couleur de certaines cases.

2.1 Première étape

Ici, nous allons exposer la manière de déterminer si, pour une séquence donnée, il est possible de colorier une ligne tout en respectant cette séquence. C'est le rôle de la fonction `T0`. L'appel `T0(j, l)` renvoie True s'il est possible de colorier les $j+1$ premières cases $(i,0),(i,1),...,(i,j)$ de la ligne l_i avec la sous-séquence $(s_1,...,s_l)$ des l premiers blocs de la ligne l_i et False sinon.

La ligne l_i peut être coloriée en entier si prenant l égal à la taille de la ligne et s la liste de tous les blocs dans l'appel `T0(j, l)`.

Dans le but de coder cette fonction récursivement, il faut distinguer plusieurs cas :

1. Si $l = 0$ (pas de bloc), $j \in \{0,...,M-1\}$: La fonction renvoie True.
2. Si $l \geq 1$ (au moins un bloc) :
   (a) Cas où $j < s_l - 1$: La fonction retourne False, il n'y a pas assez de place pour mettre le bloc.
   (b) Cas où $j = s_l - 1$: La fonction retourne True s'il ne reste qu'un seul bloc et False sinon.
   (c) Cas où $j > s_l - 1$: On fait un appel récursif en supposant dans un premier temps que la case j de la ligne l_i est blanche donc on passe à la prochaine case (`T0(j - 1, l)`). Si on suppose que la case est noire, on fait un appel récursif où on pose le bloc : `T0(j - s[l-1] - 1, l - 1)`.

2.2 Généralisation

Ensuite, nous allons généraliser cette approche au cas d'une ligne dont la couleur de certaines cases est déjà fixée. Cela se fait l'aide de la fonction `T( j, l, s, li,dict)` avec `j` et `l` les indices respectifs de `s` (la liste de la longueur des blocs) et `li` (la ligne avec les cases à remplir). La variable `dict` est un dictionnaire de mémoïsation qui permet de stocker les résultat des appels précédent, pour éviter les appels inutiles. C'est un des principes de la programmation dynamique.

On considère M la longueur de la liste `li` placée en paramètre et l le nombre de blocs. Nous avons pris la liberté d'implémenter deux fonctions supplémentaires : `is_not` et `put_block`. La premiere fonction verifie que la couleur des cases 0 à n soit pas blanc ou noir selon le parametre passé en argument. La deuxieme fonction quant à elle verifie si on peut bien poser un block en verifant que les n prochaines cases soit pas blanches et que la n+1 case soit pas noire. Leur complexité est en $O(M)$.

La fonction `T(j, l, s, li,dict)` utilise un dictonaire `tab` de dimensions de taille maximale $M \times l$ pour stocker les résultats intermédiaires. La complexité totale est donc $O(M^2 \times l)$. L'ajout de la complexité linéaire des fonctions auxiliaires affecte la complexité globale. Soit l le nombre de blocs. S'il y a plus de blocs que de cases dans la ligne, la complexité est en $\Omega(1)$, sinon on a $l < M$. La complexité peut être approximée par $O(M^3)$.

2.3 Propagation

Nous mettons en lumière comment exploiter cet algorithme pour identifier des cases qui doivent nécessairement être blanches ou noires dans une ligne ou une colonne, en procédant ensuite par propagation pour colorier partiellement la grille.

Cet algorithme s'appuie sur deux fonctions : `colorlig` et `colorcol`. Elles permettent de colorier ligne par ligne et colonne par colonne en vérifiant qu'une unique couleur peut être placée sur une case de la grille (si la case n'est pas déjà coloriée). C'est ensuite dans la fonction `coloration` qu'on parcourt les lignes et les colonnes de la grille (à plusieurs reprises si une case a été modifiée). Cette fonction permet aussi de vérifier si la grille est solvable ou non et renvoie donc la grille (partiellement) complétée avec une chaîne de caractères qui indique si la grille a été entièrement coloriée (`"T"`), partiellement coloriée (`"N"`) ou n'est pas solvable (`"F"`).

Ces fonctions parcourent toutes les lignes ou toutes les colonnes et font chacune deux appels à `T`. Leur complexité respective est donc en $O(N^4)$ et $O(M^4)$.

Dans l'algorithme `coloration`, on itère jusqu'à ce que toutes les cases de la grille soient coloriées ou qu'une impossibilité soit détectée. On parcourt une nouvelle fois une ligne ou une colonne à chaque fois qu'une de leurs cases est modifiée. Parcourir les lignes et les colonnes se fait donc en une complexité de $O(NM)$.
Pour modifier chaque ligne, il faut parcourir dans le pire cas l'ensemble des lignes donc en $O(N)$ puis appeler la fonction `colorlig` et ajouter aux colonnes à voir celles qui ont été modifiée (ce qui est en $O(N + M)$). La complexité du parcours de ces lignes est en $O(N(M^4 + M + N)) \equiv O(NM^4 + N^2)$.
Symétriquement, on obtient la complexité du parcours des colonnes en $O(MN^4 + M^2)$.

Ainsi, on obtient une complexité totale en :

$$O(NM(NM^4+N^2+MN^4+M^2)) \equiv O(N^2M^5+N^3M+N^5M^2+NM^3) \equiv O(N^2M^5+N^5M^2)$$

C'est une complexité en temps polynomiale.

Grace à une fonction d'affichage, on obtient le résultat suivant avec l'instance `0.txt`.

2.4 Tests

Suite aux tests des instances `1.txt` à `10.txt`, nous avons pu établir le tableau suivant de la résolution en fonction du temps :

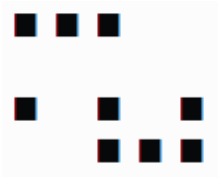

FIGURE 1 – Grille de l'instance `0.txt` avec la résolution incomplète

instances	1	2	3	4	5	6	7	8	9	10
temps (en secondes)	0.0	0.07	0.06	0.09	0.13	0.39	0.16	0.32	3.04	3.63

Voici ce qu'affiche le programme pour l'instance `9.txt`.

FIGURE 2 – Grille de l'instance `9.txt` avec la résolution incomplète

Concernant l'instance `11.txt`, l'image n'est pas complète. En effet, les cases de la grille ont une croix, ce qui veut dire que l'algorithme n'a pas réussi à colorier la grille dans son entièreté.

FIGURE 3 – Grille de l'instance `11.txt` avec la résolution incomplète

3 Méthode complète de résolution

Comme l'illustre la grille de l'instance `11.txt`, la méthode exposée dans la section précédente présente une limitation, elle ne permet pas la résolution complète de certaines grilles. Nous avons donc implémenté un autre algorithme récursif qui se base sur la résolution partielle du premier algorithme. Chaque embranchement dans l'arbre des appels récursifs (arbre d'énumération) représente le choix de colorier en blanc ou en noir une case de la grille, suivant une numérotation de gauche à droite et de haut en bas, de 0 à NM - 1. Afin de limiter la taille de l'arbre d'énumération exploré, à chaque décision de colorier une case en blanc ou en noir, une propagation est effectuée depuis cette case. Cette propagation vise à étendre le coloriage, ou éventuellement à détecter une impossibilité par ce choix de coloration.

Cet algorithme parcourt les cases non-déterminées, jusqu'à ce qu'elles soient toutes coloriées. La complexité temporelle totale dépend du nombre d'appels récursifs effectués. Chaque appel récursif peut engendrer deux appels supplémentaires, chacun explorant une branche différente (blanc ou noir). On considère $T(NM)$ la suite qui représente la complexité de `enum_rec` sur une grille de taille $N \times M$. On obtient la récurrence suivante :

$$T(NM) = O(N^2M^5 + N^5M^2) + 2T(NM - 1)$$

On obtient alors une complexité en :

$$O((N^2M^5 + N^5M^2)2^{NM})$$

C'est une complexité exponentielle.

3.1 Implantation et tests

Nous avons implémenté une fonction `colorierEtPropager` qui utilise le même principe que la fonction `colorier` sauf que la grille est déjà partiellement coloriée en entrée, donc les lignes et colonnes à traiter sont différentes.
C'est dans la fonction `enum_rec` qu'on construit l'arbre avec les différents appels récursifs. En effet, on colorie une case en noir ou en blanc et on vérifie si la modification n'amène pas à une grille irrésolvable à l'aide de la fonction `colorierEtPropager`.
La fonction finale `enum` construit la grille avec la coloration partielle. Si cette coloration s'avère être complète, `enum` renvoie la grille. Sinon, elle appelle à `enum_rec` pour construire un arbre et enfin obtenir une résolution complète.

On obtient alors la grille suivant pour l'instance `11.txt` :

FIGURE 4 – Grille de l'instance `11.txt` avec la résolution complète

Voici le tableau contenant les temps (en secondes) pour les méthodes de résolution des instances :

Instances	Temps pour la résolution incomplète	Temps pour la résolution complète
1	0.0	0.0
2	0.07	0.07
3	0.06	0.06
4	0.09	0.09
5	0.13	0.13
6	0.39	0.39
7	0.16	0.16
8	0.32	0.32
9	3.04	3.04
10	3.63	3.63
11	0.008	0.00
12	0.23	0.27
13	0.36	0.36
14	0.23	0.26
15	0.08	0.19
16	0.23	11.07

On remarque que plus la taille de l'instance est grande, plus la différence de temps entre les deux méthodes est grande. En effet, la première résolution a un temps en complexité polynomiale alors que pour la seconde la complexité est exponentielle.

FIGURE 5 – Grille de l'instance `15.txt` avec la résolution incomplète

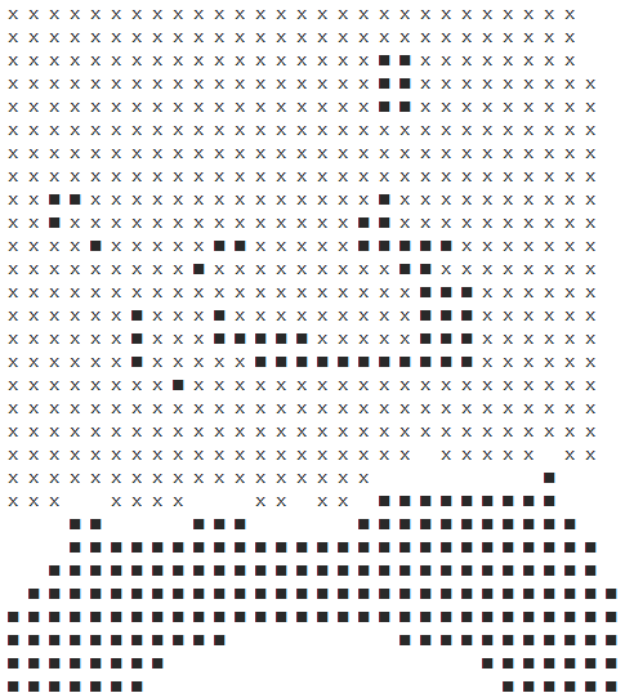

FIGURE 6 – Grille de l'instance `15.txt` avec la résolution incomplète