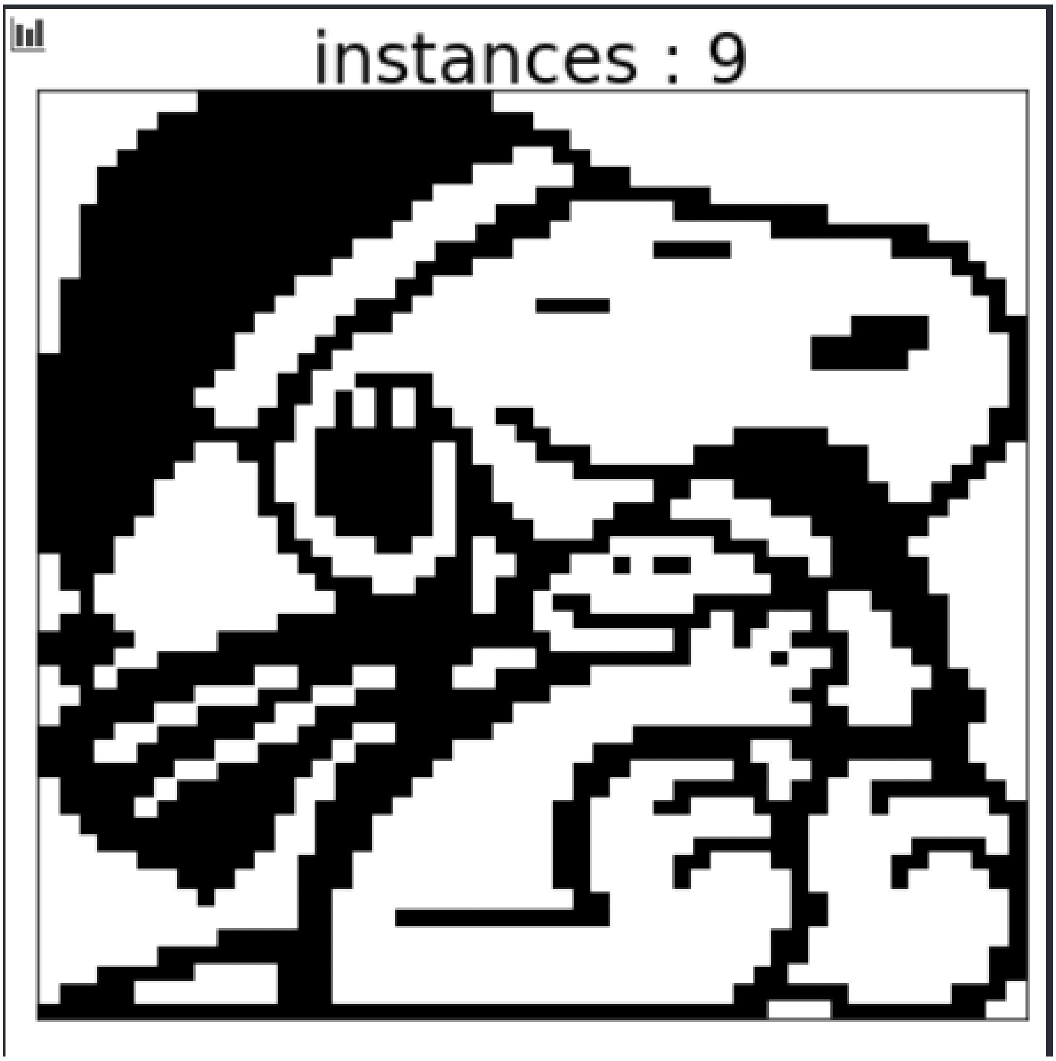
Projet : LU3IN003

Un problème de tomographie discrète



Ung Thierry

Table des matières

I Méthode incomplète de résolution

A) Première étape

Q1

Si pour tout T(j, l), où j allant de 0 à m-1 et l allant de 1 à k, ont été calculé, il est possible de savoir si une ligne l(i) peut être coloriée avec sa séquence entière.
Pour cela, il faut récupérer la valeur de T(m-1, k) (c'est-à-dire j = m-1 où m-1 est la dernière case de la sous ligne l(i) et l = sk où sk est le dernier bloc de la séquence de la ligne l(i)) qui représente une ligne entière avec sa séquence entière :

• si T(m-1, k), retourne True : la ligne peut être coloriée.

• si T(m-1, k), retourne False : la ligne ne peut pas être colorié avec cette séquence s.

Q2

Pour l = 0, T(j, l) retourne True, car si l = 0 cela veut dire que la ligne a une séquence vide, et comme on considère la grille non coloriée, entre 0 et j, on constate qu'il y a aucune case coloriée, ainsi l = 0 est vérifié.

Pour l ≥ 1 et j < s(l) - 1, T(j, l) retourne False, avec l ≥ 1, cela signifie qu'on a au moins un bloc à traiter dans la sous séquence et on sait, avec la condition j < s(l) -1, que dans la sous ligne l(i) allant de 0 à j, il n'y a pas assez de case pour insérer le bloc entier s(l) représentant le dernier bloc actuel de la séquence s.

Pour j = s(l) - 1, nous avons deux sous-conditions possibles :

• Soit l = 1, donc T(j, l) retourne True car il y a de place pour seulement un bloc, ici la condition j == s(l) - 1 signifie que le bloc s(l) fait la taille de la sous ligne l(i) et l=1 annonce qu'il n'y a qu'un seul bloc à traiter dans la sous ligne.

• Soit l > 1, donc T(j, l) retourne False car il n' y a pas assez de place pour plus d'un bloc. Plus précisément, la condition l ≥ 1 nous dit que la sous-séquence contient plus d'un bloc et le fait que le bloc s(l) vérifie que j = sl-1 assure celui-ci occupe toute les cases de la sous-ligne donc, il n'est plus possible d'insérer le prochain bloc s(l-1).

Q3

Pour l ≥ 1 et j > s(l) -1, on a deux cas possibles dans cette partie du projet :

• Soit la case (i,j) est blanche, dans ce cas, on appel par récurrence le T(j-1, l) pour constater si on peut placer le même bloc s(l) pour un j allant de 0 à j-1 sachant que la case (i,j) ne peut être coloriée, car celle-ci est blanche et doit donc le rester.

• Soit la case (i,j) est noir, dans ce cas, on fait un appel par récurrence du T(j-s(l)-1, l-1), dans cette partie, on rappel que seule la case (i,j) est coloriée, ainsi les cases allant (i, j=(0, …, j-1)) sont vides. Donc, si la case(i,j) est noir, et qu'il y a assez de la place pour insérer le bloc s(l), c'est la condition j > s(l)-1 qui nous l'affirme. Alors on peut directement faire un appel récursif en décrémentant le j de la valeur du bloc s(l) et en faisant bien attention d'enlever -1 pour la case blanche qui doit séparer deux blocs, sans oublier de décrémenter l de 1 pour passer au prochain bloc, de la séquence de la ligne l(i).

Q4

```
def T_without_color(j, l, ligne, sequence):
    """ Entree : j          = indice de la j+1 premieres cases.
                 l          = indice bloc des li premiers blocs.
                 ligne      =
                 sequence   = Sequence de la ligne li (s1,...,sl).
        Sortie : Retourne True si il y a un coloriage possible d'une ligne non coloriée avec une sequence
                 donnée sinon retourne False.
    """
    # Sequence vide :
    if l == 0:
        return True
    # Au moins un bloc dans sequence de la ligne l(i)
    if l >= 1:
        if j < sequence[l-1]-1 :
            return False
        # Deux conditions pour le cas j == sequence[l-1]-1 :
        if j == sequence[l-1]-1 and l == 1:
            return True
        if j == sequence[l-1]-1 and l > 1:
            return False
        # On considère la ligne l(i) vide :
        if j > sequence[l-1] -1:
            return  T_without_color(j-1, l, ligne, sequence) or T_without_color(j-(sequence[l-1])-1, l-1, ligne, sequence)
```

B) Généralisation

Q5

Pour l = 0 :

• Si l = 0, cela signifie que les cases (i, j=(0,…,j)) doivent être toutes blanches ou vides.

Pour l ≥ 1 et j < s(l) - 1 :

• Il n'y a aucune modification. C'est à dire, si les deux conditions sont vérifiés alors le T(j, l) retourne False.

Pour l == 1 et j = s(l) - 1 :

• On vérifie si tout les cases allant (i, j=(0,…, j)) sont, soit noire, soit vide. En effet, s'il y avait une case blanche dans cette intervalle de cases, alors on ne pourrait pas placer entièrement la séquence s(l).

Pour l > 1 et j = s(l) - 1 :

• Il n'y a aucune modification. On retourne False, car si on a plus d'une case à placer et que l'une des cases occupe l'ensemble de la sous ligne, alors il n'est pas possible d'insérer le second bloc de la séquence l(i).

Pour l ≥ 1 et j > s(l) - 1 avec une case blanche :

• On regarde dans la mémoire cache qui représente le stockage des valeurs lors d'un appel T(j,l), et on récupère la valeur du T(j-1, l).

Pour l ≥ 1 et j > s(l) - 1 avec une case noir :

• On vérifie si les cases (i, j-s(l)+1 à j) sont de couleurs noir et vide, afin de s'assurer de pouvoir insérer le bloc s(l) actuel. Si cela est valide, il se présente deux cas :

  • Soit il nous reste qu'un bloc à vérifier, donc on doit observer s'il n'y a aucune case de de couleur noir avant le bloc insérer, pour vérifier les contraintes des séquences. C'est-à-dire, après avoir insérer le dernier bloc s(l) de la séquence de l(i), on ne doit pas se retrouver avec une ligne qui contient plus de bloc de couleur noir que la séquence nous suggère.

• Soit il nous reste plusieurs blocs à insérer, et dans ce cas nous devons vérifier que la case, précédent le bloc s(l) à stocker dans la ligne, doit comporter une case blanche ou vide.

• Si l'une des deux conditions est valide alors T(j,l) est True sinon False.

Q7

```python
def Top_Down(ligne, sequence, memoire_cache):
    j, l = len(ligne), len(sequence)
    # On regarde si la clef (j,l) a deja ete calcule:
    if (j,l) in memoire_cache :
        return memoire_cache[(j,l)]
    # Si la clef n'a pas ete calcule ulterieurement:
    if l == 0:
        memoire_cache[(j,l)] = np.all(ligne <= 0)
    elif j < sequence[-1]:
        memoire_cache[(j,l)] =  False
    elif l==1 and j == sequence[-1]:
        memoire_cache[(j,l)] =  np.all(ligne >= 0)
    elif l > 1 and j == sequence[-1]:
        memoire_cache[(j,l)] =  False
    elif j > sequence[-1]:
        if ligne[j - 1] == CASE_WHITE:
            memoire_cache[(j,l)] =  Top_Down(ligne[:j - 1],sequence,memoire_cache)
        else :
            if ligne[j-1] == CASE_EMPTY and (j-1,l) in memoire_cache :
                memoire_cache[(j,l)] = memoire_cache[(j-1,l)]
            else :
                # Decomposition de la condition en plusieur variable :
                insert_bloc = np.all(ligne[j-sequence[-1]:] >= 0)
                one_bloc = l==1 and np.all(ligne[:j-sequence[-1]] <=0)
                several_bloc = (ligne[j-sequence[-1]-1] <= 0) and \
                            Top_Down(ligne[:j - sequence[-1] - 1],sequence[:-1],memoire_cache)
                # Condition final pour validation :
                condition = insert_bloc and (one_bloc or several_bloc)
                if ligne[j - 1] == CASE_BLACK:
                    memoire_cache[(j,l)] = condition
                elif ligne[j - 1] == CASE_EMPTY:
                    memoire_cache[(j,l)] =  condition or Top_Down(ligne[:j - 1],sequence,memoire_cache)
    return memoire_cache[(j,l)]

def Top_Down_Init(ligne, sequence):
    return Top_Down(ligne, sequence,{})
```

C) Propagation

Q9

```
def Coloration(grille,sequences_lignes, sequences_colonnes, indiceAVoir) :
    indiceAajouter = set()
    for i in indiceAVoir:
        # Récupere seulement les case à qui sont non coloriée :
        # la fonction where, nous donne l'indice de ces cases non coloriées :
        for j in np.where(grille[i] == CASE_EMPTY)[0]:
            # Recupere les sequences d'une ligne et une colonne pour la case (i,j)
            lig = sequences_lignes[i]
            col = sequences_colonnes[j]
            # Test avec une case (i,j) BLACK :
            grille[i][j] = CASE_BLACK
            test_black = Top_Down_Init(grille[i],lig) and Top_Down_Init(np.transpose(grille)[j], col)
            # Test avec une case (i,j) White :
            grille[i][j] = CASE_WHITE
            test_white =  Top_Down_Init(grille[i],lig) and Top_Down_Init(np.transpose(grille)[j], col)
            if not test_black and not test_white:
                # Leve une exception en cas de grille non resolvable:
                # Cette exception est rattraper dans le main
                raise GrilleNonResolvable("Grille non resolvable pour la ligne "+i+",la colonne "+j)
            elif test_black and test_white:
                # Impossible de decider de la couleur de la case :
                grille[i][j] = CASE_EMPTY
                indiceAajouter.add(j)
            elif test_black and not test_white:
                grille[i][j] = CASE_BLACK
                indiceAajouter.add(j)
            else:
                grille[i][j] = CASE_WHITE
                indiceAajouter.add(j)
    return indiceAajouter
```

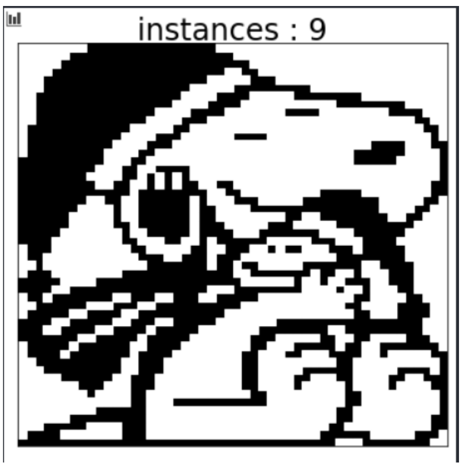
```
def propagation (sequences_lignes, sequences_colonnes) :
    # Récupere la taille des contraintes sur les lignes et colonnes :
    N, M = len(sequences_lignes), len(sequences_colonnes)
    # Initialisation de la grille avec une valeur de type CASE_EMPTY = 0 :
    grille = np.full((N, M), CASE_EMPTY)
    # Initialisation des lignes et colonnes a voir :
    # On choisit de prendre la fonction set() afin de ne pas avoir de doublons
    lignesAVoir, colonnesAVoir = set(range(N)), set()
    # Debut d'Algorithme :
    while lignesAVoir or colonnesAVoir:
        # Test toute les lignes et retourne les colonnes a voir,
        #c'est à dire les colonnes ou les case (i,j) d'une ligne ont été modifié:
        colonnesAVoir = Coloration(grille, sequences_lignes, sequences_colonnes,lignesAVoir)
        # Les lignes étant toute parcourut, on met l'ensemble à vide
        lignesAVoir = set()
        # Test toute les colonnes:
        #c'est à dire les lignes ou les case (i,j) d'une colonnes ont été modifié:
        lignesAVoir = Coloration(np.transpose(grille), sequences_colonnes, sequences_lignes, colonnesAVoir)
        colonnesAVoir = set()
    return grille
```

D) Tests

Q10

Instances	Temps(en secondes)
0	0.004000425338745117
1	0.004000186920166016
2	0.8549966812133789
3	0.3049774169921875
4	0.6979987621307373
5	0.7569994926452637
6	2.288024425506592
7	0.823979377746582
8	1.6709904670715332
9	27.20965576171875
10	31.39819645881653

Instance 9 :



Q11

On remarque, pour la grille de l'instance 11, que chaque case (i,j) de la grille est non colorier. On observe un temps de 0 secondes.
Ce temps s'explique par le fait que lorsqu'on applique la fonction Coloration sur les lignes à voir, celle-ci nous retourne un ensemble de colonne à voir vide, et c'est normal car elle n'a pas réussi à déterminer une seule case qui pourrait prendre la couleur noir ou blanche . Donc le programme se termine sans avoir résolu une seule case (i,j) de la grille.

II Méthode complète de résolution

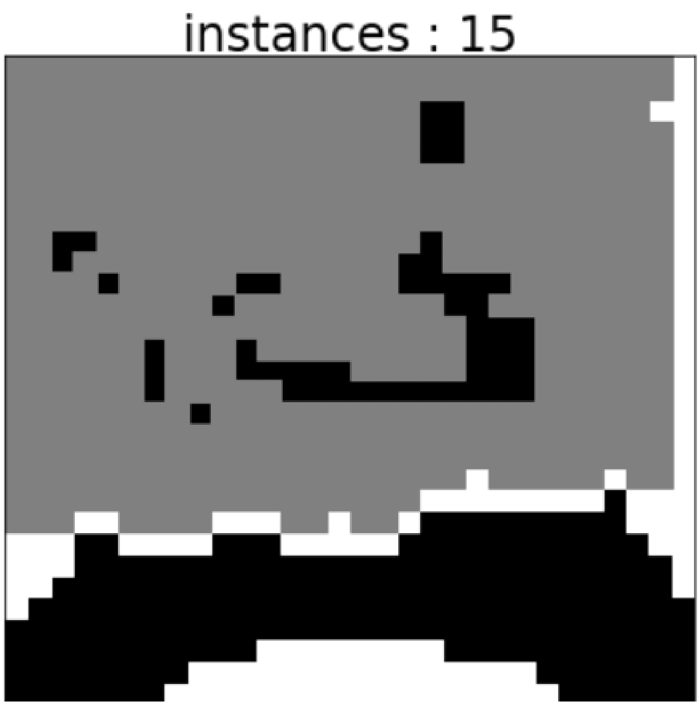
Q14

Instances	Méthode section 1 Temps (en secondes)	Méthode section 2 Temps (en secondes
0	0.004000425338745117	
1	0.004000186920166016	
2	0.8549966812133789	
3	0.3049774169921875	
4	0.6979987621307373	
5	0.7569994926452637	
6	2.288024425506592	
7	0.823979377746582	
8	1.6709904670715332	
9	27.20965576171875	
10	31.39819645881653	
11	0.0	

12	1.9909992218017578	
13	2.283006429672241	
14	1.1730234622955322	
15	0.8249659538269043	
16	2.1930007934570312	

On devrait remarquer, dans la méthode section 2 que toutes les instances sont résolus, c'est à dire qu'elles ne contiennent pas de case non décidée. Contrairement à la version programmation dynamique, où on avait les instances de 11 à 16 non résolus.

Instance 15 (méthode section 1)



III Conclusion

Au cours de ce projet, on prend conscience de l’importance de la programmation dynamique qui permet d’éviter des appels récursifs redondants et très coûteux, par le biais de la mémorisation des résultats de sous-problèmes afin de ne pas les recalculer plusieurs fois le même T(j, l). Mais la programmation dynamique à elle seule, ne permettait pas de résoudre toute sorte de grille, comme celles des instances 11 à 16 qui avaient toutes plusieurs cases qui étaient non décidées, c’est-à-dire, on pouvait soit mettre une case blanche ou une case noire.