

Algorithmique 2 - LU3IN003

COMPTE RENDU PROJET

Année 2020/2021

Responsables de l'UE : Olivier Spanjaard et Fanny Pascual
Responsables de TD : Anne-Elisabeth Falq et Nadjet Bourdache

Introduction

Le projet consistait à étudier et à résoudre un problème de tomographie discrète, représenté par le jeu du “Picross” ou aussi appelé “Nonogramme”. Pour répondre à ce problème, nous avons choisi d’élaborer nos fonctions en *Python*.

Dans un premier temps, nous allons effectuer une brève présentation des décisions que nous avons prises pour la conception du projet. Ensuite, nous vous présenterons un algorithme qui amènerait une solution partielle (ou complète pour certains cas). Enfin, nous vous proposerons des fonctions implémentant une solution complète de la grille.

1) Présentation

Tout d’abord, nous avons choisi d’écrire en *Python* pour différentes raisons. En tenant compte de la flexibilité de l’implémentation de l’interface graphique (Tkinter), de la lisibilité et de la propreté, *Python* semblait être le langage informatique correspondant au mieux à nos besoins. Indépendamment de cela, après avoir utilisé C comme langage durant la L2, ce projet paraissait être la situation idéale pour se refamiliariser avec le *Python*. Enfin, ce projet nécessite l’utilisation de la programmation dynamique. Or, le *Python* est un langage à évaluation paresseuse donc une valeur est calculée seulement si nous avons besoin d’elle. On peut donc en conclure que les étoiles semblent alignées pour le bon déroulement de ce projet avec le *Python*.

Pour bien structurer notre code, nous avons écrit plusieurs fichiers qui correspondent aux fonctionnalités distinctes de notre projet. Le fichier “*Nonogramme*” peut s’apparenter au fichier *main* dans le C. Il permet d’afficher la grille avec l’interface graphique et les valeurs obtenues pour le temps de résolution. Une fonction menu a été implémentée pour faciliter l’usage de nos fonctions par l’utilisateur.

D’autre part, comme son nom l’indique, le fichier “*parser*” récupère et analyse les données des instances que vous avez fournies. Grâce à ce dernier, nous connaissons le nombre de lignes *N*, le nombre de colonnes *M*, les séquences des lignes *Seq_lignes* et les séquences des colonnes *Seq_colonnes* de l’instance que nous voulons résoudre. Ces variables sont définies comme globales dans le fichier “*config*” pour préserver la lisibilité du code et s’abstenir de les mettre en paramètre de toutes nos fonctions du projet.

Maintenant que nous avons récupéré les valeurs nécessaires à la représentation du problème, nous pouvons initialiser la grille avec une fonction se situant dans le fichier “*jeu*”. Ce dernier contient toutes les fonctions d’initialisation et auxiliaires comme *change_couleur(i,j,cl,G)* qui change la couleur de la case (*i,j*) de la grille *G* avec la couleur *cl* ou bien *affichage_grille_terminal(G)* qui affiche la grille *G* dans le terminal.

Concernant la représentation des couleurs d'une case, nous avons opté pour que la valeur -1 corresponde à une case vide, la valeur 0 à une case blanche et enfin la valeur 1 à une case noire.

Pour mieux visualiser notre avancement et pour avoir un résultat concret, nous avons choisi de créer une interface graphique avec Tkinter. Celle-ci représente la grille du jeu à résoudre. Une case noire est coloré en noir, une case blanche est symbolisée par une croix et une case vide est coloré en blanc, pour conserver les conventions du jeu.

Quant à la partie algorithmique, nous avons créé trois fichiers différents pour mieux gérer les méthodes de résolution que nous vous présenterons en détails dans la suite de ce rapport.

2) Méthode incomplète de résolution

Dans cette première partie, nous avons implémenté des fonctions qui résolvent partiellement la grille. A présent, nous allons répondre de façon détaillée aux questions posées dans le sujet.

a) Première étape

Question 1 :

Il est possible de colorier la ligne I_i entière avec la séquence entière si et seulement si:

$$T(M - 1, k) = \text{True}$$

où $k = \text{nombre de séquences de la ligne } I_i$

$M = \text{nombre de colonnes de la grille}$

Question 2 :

- a. Pour le cas de base 1 quand $l = 0$ (pas de bloc) , $T(j, l)$ prend la valeur Vrai pour tout $j \in \{0, \dots, M - 1\}$
- b. Pour le cas de base 2 quand $l \geq 1$,
- c. $j < s_l - 1$, $T(j, l)$ prend la valeur Faux
- d. $j = s_l - 1$, $T(j, l)$ prend la valeur Vrai si $l = 1$, sinon Faux

Question 3 :

Si la case (i, j) est blanche, alors on passe à la case suivante sans changer le bloc de la liste de séquences lignes.

Si la case (i, j) est noire, alors d'une part on enlève le dernier bloc de la liste de séquences. D'autres part, on enlève à j la longueur du dernier bloc et la case blanche qui est censée suivre.

On obtient ainsi l'équation suivante :

$$T(j,l) = T(j - s_i - 1, l - 1) \text{ ou } T(j - 1, l)$$

Question 4 :

On définit `sequence_valide(i, j, ListeSeq, l)` avec i la ligne de grille, j la colonne de la grille, `ListeSeq` la liste des séquences des lignes, l l'indice du bloc de la séquence courante.

La fonction rend vraie s'il est possible de colorier les $j+1$ premières cases de la ligne l , avec la sous-séquence (s_1, \dots, s_l) des l premiers blocs de la ligne l . Dans un premier lieu, on obtient le pseudo-code suivant :

```
1  fonction sequence_valide( i, j, ListeSeq, l ) :
2      si l = 0 alors retourner Vrai
3
4      si j < ListeSeq(i,l-1) - 1 alors retourner Faux
5
6      si j = ListeSeq(i,l-1) - 1 alors
7
8          si l = 1 alors retourner Vrai
9
10     retourner sequence_valide(i, j - ListeSeq(i,l-1) - 1, ListeSeq , l-1)
11     |   |   OR sequence_valide(i, j - 1 , ListeSeq, l)
```

Après réflexion sur l'efficacité de la récursivité, nous avons décidé de mettre l'accent sur la complexité temporelle vis à vis de la complexité mémoire. Ainsi, nous avons choisi d'utiliser la programmation dynamique, qui permettrait d'éviter les appels de fonctions redondants et qui permettrait aussi de passer d'une complexité exponentielle en une complexité polynomiale. Nous avons fait le choix de ne pas montrer le tableau T qui stocke les différents appels de fonctions dans le pseudo-code ci-dessus mais nous vous invitons à aller voir le code sur cette partie.

b) Généralisation

Question 5 :

Après la modification de l'algorithme précédent afin qu'il prenne en compte les cases déjà coloriées, nous obtenons les deux fonctions ci-dessous. Nous sommes partis initialement des mêmes cas de base que la première méthode, cependant un certain nombre de cas ont dû être rajoutés. On considère dans un premier temps, que si la case actuelle (i,j) est une case blanche, alors on passe directement à la case suivante.

Dans un second temps, si la case actuelle (i,j) est noire, alors on regarde d'abord s'il y a une case blanche de $j-1$ à $j-s_l+1$, si c'est le cas, retourner False, sinon regarder si la case $(i,j-s_l)$ est noire, c'est-à-dire la case juste après avoir placé le bloc. Elle doit être nécessairement blanche pour respecter les règles du jeu. Si ce n'est pas le cas, retourner False.

Enfin si la case (i,j) est vide, alors on regarde s'il y a une case blanche de j-sl à j-sl+1. Si tel est le cas, alors on passe à la case suivante. Sinon on regarde s'il y a une case noire en j-sl (après le bloc), si oui alors retourner False, sinon retourner ce qu'on a obtenu à la question 2.c). Pour plus de détails concernant les différents cas, nous vous conseillons vivement d'aller voir les commentaires du code.

```

1  fonction sequence_valide_lignes_rec ( i , j , ListeSeq, l , G ):
2      si tab(l,j) !=Vide alors retourner tab(l,j) # si on a déjà vu T(j,l) on retourne la valeur trouvée
3      si l = 0 alors # (cas 1) si la séquence est vide on vérifie s'il y a des cases noires entre 0 et (i,j)
4          pour n < j+1 faire
5              si (i,n) = noir alors
6                  tab(l,j) = Faux
7                  retourner Faux
8              si j < sl - 1 alors # (cas 2a)
9                  tab (l,j) = Faux
10                 retourner Faux
11             si j = sl - 1 alors # (cas 2b)
12                 pour n < j faire
13                     if (i,j) = blanc
14                         tab(l,j) = Faux
15                     sinon tab(l,j) = Vrai
16             si tab(l,j) = 1 alors
17                 si l = 1 alors tab(l,j) = Vrai
18                 sinon tab( l,j ) = Faux
19             retourner tab(l,j)
20         # (cas 2c)
21         si (i,j) = blanc alors
22             tab (l,j) = sequence_valide_lignes_rec ( i , j - 1 , ListeSeq , l , G )
23             retourner tab(l,j)
24
25         si (i,j) = noir # si (i,j) noire on vérifie s'il y a des cases blanches de j - 1 à j - sl + 1
26             pour 0 < n < sl faire
27                 si (i, j-n) alors
28                     tab(l,j) = Faux
29                 sinon tab(l,j) = Vrai
30             si tab(l,j) = Faux :
31                 retourner tab(l,j)
32             si (i, j-sl) = noir # si la case juste après la séquence est noire, alors T(j,l) = Faux
33                 tab(l,j) = Faux
34                 retourner Faux
35             # on passe à la valeur suivante du T
36             tab(l,j) = sequence_valide_lignes_rec( i , j - sl - 1 , ListeSeq , l - 1 , G )
37             retourner tab(l,j)
38         # si (i,j) vide
39         pour 1 < n < sl faire
40             si (i, j-n) = noir alors
41                 tab(l,j) = sequence_valide_lignes_rec(i, j-1, ListeSeq, l , G)
42                 retourner tab(l,j)
43
44         si (i, j-sl) == noire: # si la case juste après la séquence est noire, alors T(j,l) = Faux
45             tab(l,j) = sequence_valide_lignes_rec( i , j - 1 , ListeSeq , l , G )
46             retourner tab(l,j)
47         tab(l,j) = sequence_valide_lignes_rec( i , j - sl - 1 , ListeSeq , l - 1 , G )
48             or sequence_valide_lignes_rec( i , j - 1 , ListeSeq , l , G )
49         retourner tab(l,j)

```

Pour simplifier l'appel de la fonction récursive et pour initialiser le tableau dynamique, nous utilisons la méthode ci-dessous.

```

1  fonction sequence_valide_complet_lignes( i , j , G ):
2      init_T_lignes(l)
3      retourner sequence_valide_lignes_rec ( i , Seq_lignes , M - 1 , l , G )

```

Question 6 :

Analysons la complexité de l'algorithme en fonction de M :

On commence par déterminer le nombre de valeurs de $T(j,l)$ à calculer.

Nous rappelons que pour la programmation dynamique, nous récupérons tous les appels de fonctions dans un tableau *tab* de dimension $k * M$ avec k le nombre de blocs de la séquence de la ligne i . Comme nous décrémentons à chaque appel récursif les paramètres j et l (initialement $j=M-1$ et $l=k$) et nous allons stocker les valeurs obtenues dans les cases du tableau dynamique. Il existe donc un certain rang n où la fonction retournera soit True soit False en fonction de nos cas de bases (soit $l=0$, soit $j < sl - 1$ ou soit $j=sl - 1$).

Dans le pire cas, k est égale à $M/2$ blocs de longueur 1 car il faut placer une case blanche entre chaque bloc noir. Ainsi, nous pouvons majorer le nombre de $T(j,l)$ appelé par $M*M/2$, soit le nombre de cases de *tab*.

Concernant la complexité de calcul de chaque valeur de $T(j,l)$, nous parcourons une boucle allant de 0 à sl ce qui revient, dans le pire cas, à faire M opérations à l'intérieur cette boucle en $O(1)$, auxquelles nous pouvons rajouter 12 opérations en $O(1)$ et l'initialisation du tableau dynamique en $O(M*M/2)$. Les opérations en $O(1)$ sont soit des affectations soit des comparaisons.

Nous pouvons donc conclure que la fonction *sequence_valide_complet_lignes* est en $O((M*M/2)*M + 12 + M*M/2)$ donc en $O(M^3)$.

c) Propagation

Pour implémenter la fonction COLORATION donnée dans l'annexe du projet, nous avons besoin de créer les fonctions COLORELIG et COLORECOL qui gère la propagation des couleurs sur respectivement une ligne et une colonne, comme il est expliqué dans le sujet. Nous avons pris le soin d'écrire le pseudo-code de cette méthode :

```
1  fonction COLORELIG ( G , i ) :
2      Nouveau = set()
3      max = 0
4      case_noir = 0
5      l = len(Seq_lignes[i])
6      pour 0 < i < l
7          max += Seq_lignes(i,k)
8      retourner COLORELIG_REC ( G , i , M-1 , l , Nouveau, max , case_noir)
9 
```

La fonction COLORELIG fait appel à la fonction COLORELIG_REC.

```

1  fonction COLORELIG_REC ( G , i , j , l , Nouveau , max , case_noir ) :
2      si j < 0 retourner ( True , G , Nouveau )
3
4      # if somme des cases noirs de la sequence de la ligne i == nombre de cases noires posées
5      si max = case_noir alors
6          si sequence_valide_complet_lignes ( i , M - 1 , G ) :
7              pour k <= j faire
8                  if (i ,j) == vide :
9                      change_couleur( i , k , blanc , G )
10                     Nouveau = Nouveau U { k }
11                     retourner ( Vrai , G , Nouveau)
12                     retourner (Faux, [], {})
13
14     si case(i,j) est noire alors
15         case_noir += 1
16
17     # else case vide
18     sinon si cas(i,j) est vide
19         noir = Faux
20         blanc = Faux
21
22         change_couleur( i , j , blanc , G )
23         blanc = sequence_valide_complet_lignes( i , j , G )
24         change_couleur(i, j, noir, G)
25         noir = sequence_valide_complet_lignes( i , j , G )
26
27         si non blanc et non noir alors
28             retourner (Faux , G , {} )
29         sinon si noir et non blanc alors
30             Nouveau = Nouveau U { j }
31             case_noir +=1
32         sinon si blanc et non noir alors
33             change_couleur(i,j, blanc,G)
34             Nouveau = Nouveau U { j }
35         else :
36             change_couleur(i,j, vide,G)
37
38     retourner COLORELIG_REC ( G , i , j - 1 , l , Nouveau , max , case_noir)
--
```

Question 8 :

Montrons que l'algorithme COLORATION est de complexité polynomiale en N et M :

Commençons par la complexité des boucles de COLORATION. Dans un premier temps, nous réalisons une copie de la grille N*M. Cette opération est en O(N*M). Ensuite, LignesAVoir et ColonnesAVoir sont respectivement des ensembles initialisés avec toutes les valeurs de 0 à N-1 et 0 à M-1 donc nous sommes pour l'instant en O(N+M-2) car nous ajoutons un nouvel élément à chaque ensemble (en O(1)) Analysons maintenant la complexité de la boucle *while* et des boucles *for*. Dans le pire cas, nous partons d'une grille vide et nous colorions un certain nombre de cases dans la boucle *while*. Au vu de la condition de sortie qui est un *or*, il faut prendre le *max* du nombre de tours de boucles de chaque *for*. En ce qui concerne le *for* des LignesAVoir, nous considérons dans le pire cas que nous parcourons toutes les N lignes de la grille à chaque tour du *while*. Ce pire cas peut se reproduire au maximum M fois. Sans pertes de généralité, nous pouvons répéter le même raisonnement pour la deuxième *for*. Ainsi, nous obtenons un nombre de tour de boucle totale de 2^*N*M .

Dans un second temps, nous avons un appel de COLORELIG dans la fonction COLORATION qui initialise le paramètre j à M-1. Cette dernière s'appelle récursivement une seule fois et dans ce cas là, elle décrémente la taille de la ligne à

colorier donc nous avons M appels de fonctions. En ce qui concerne les opérations à l'intérieur de cette fonction, dans le pire cas, la case est vide. Si la case est vide, alors nous appelons 2 fois `séquence_valide_complet_lignes`. Nous avons vu plus haut que cette fonction est en $O(M^3)$. Or, `COLORELIG_REC` est appelé M fois et effectue 12 opérations dans le pire cas donc elle est en $O(M^4)$. Par conséquent, `COLORELIG` est aussi en $O(M^4)$. En suivant le même raisonnement, nous trouvons que `COLORECOL` est en $O(N^4)$.

Enfin, dans `COLORATION`, la première boucle `for LignesAVoir` fait M tours de boucle pour N lignes. Avec l'appel de `COLORELIG`, sa complexité est de $O(NM^5)$. La deuxième boucle `for ColonnesAVoir` fait N tours de boucles pour M colonnes. Avec l'appel de `COLORECOL`, sa complexité est de $O(N^5M)$.

Pour conclure, `COLORATION` est en $O(N^*M+N+M+\max(N^5M+NM^5))$ donc en $O(\max(N^5M+NM^5))$.

d) Tests

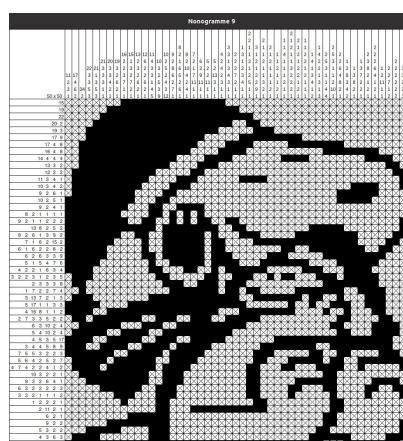
Question 10 :

Après avoir appliqué la fonction `COLORATION` sur les instances 0 à 10, voici un tableau des temps de résolutions de chaque instances pour 50 tests différents :

Instances	0	1	2	3	4	5	6	7	8	9	10
Dimensions	4x5	5x5	20x20	13x37	25x25	25x27	30x30	31x34	40x35	50x50	99x99
Résolution	True										
Temps (ms)	0.424	0.865	75	50	110	122	359	158	284	3092	3075

Les dix premières instances semblent se résoudre complètement et permettent de conclure qu'il existe un coloriage à ces grilles. Nous verrons plus tard que cela n'est pas toujours le cas.

En analysant le tableau obtenu, comme nous pouvons s'en douter, plus les dimensions augmentent, plus le temps de résolution augmente. A cette échelle, il est difficile d'observer que la méthode `COLORATION` est en complexité temporelle polynomiale de N et M mais avec l'imagination du lecteur, tout est possible...



Voici ce que nous avons obtenu pour l'instance 9. Vous pouvez retrouver en annexe toutes les autres instances ainsi qu'un bonus.

Question 11 :

Après avoir appliqué la fonction COLORATION à l'instance 11.txt, nous remarquons que la grille est encore vide. Pas une seule case n'a été colorée, et c'est bien normal! Toutes les cases de la grille peuvent être noires ou blanches. Ainsi, le programme se retrouve donc dans l'incapacité de colorier une seule case, d'où la sortie *None* qui signifie qu'elle ne peut pas conclure sur l'existence ou non d'un coloriage.

Pour pallier cette incapacité, nous avons implémenté une nouvelle méthode qui permet de résoudre entièrement le jeu.

3) Méthode complète de résolution

En suivant les instructions du projet, nous avons donc implémenté cette nouvelle méthode complète ENUMERATION. Celle-ci est basée sur un algorithme où il est question d'énumération des coloriages possibles. Nous pouvons représenter cette énumération par des arbres binaires, où chaque branchement correspond au choix de colorier en blanc ou en noir la case actuelle. Plus on plonge dans l'arbre de ENUMERATION, plus on colorie des cases. Dès qu'on trouve un coloriage possible, on remonte sans parcourir les autres branchements. Notre méthode renvoie donc s'il existe un coloriage possible mais non pas s'il existe plusieurs coloriages possibles. Cette question pourra être étudiée dans un futur projet.

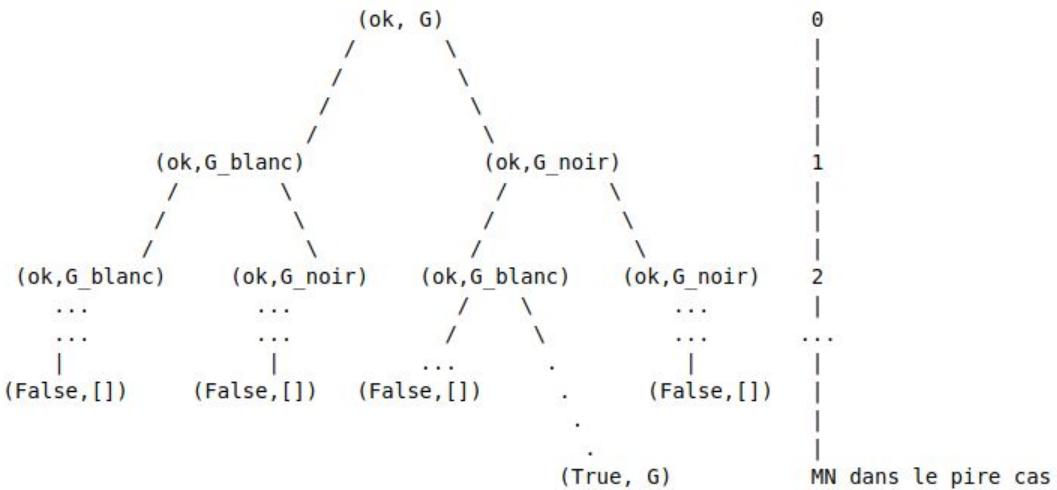
Question 12 :

Montrons que cette nouvelle méthode ENUMERATION est de complexité exponentielle en N et M :

Dans un premier temps, nous appelons la fonction COLORATION qui est en $O(\max(N^5M, NM^5))$. Dans le pire cas, celle-ci rend une grille vide. Nous devons donc trouver la prochaine case vide. Nous utilisons pour cela la fonction *prochaine_case_vide* qui parcourt toute la grille dans le pire cas donc qui est en $O(MN)$. A la fin de la fonction, nous effectuons deux appels récursifs de ENUM_REC dans le pire cas, qui correspondent aux deux branches de l'arbre des différents coloriages possibles.

Dans ENUM_REC, à chaque appel, en plus des 12 opérations effectuées en $O(1)$, nous appelons COLORIERETPROPAGER. Cette dernière est quasiment la même que COLORATION dans le pire cas, donc on peut considérer qu'elle est aussi en $O(\max(N^5M, NM^5))$.

Analysons maintenant le nombre d'appels de ENUM_REC. Il y aura une solution dans la branche d'hauteur maximale donc dans le pire cas, cette hauteur maximale sera N^5M , soit le nombre de cases totales en partant d'une grille vide. En utilisant les propriétés sur les arbres binaires, nous savons que $n < 2^h$, avec n le nombre de noeuds. Dans notre cas, $n < 2^{NM}$, avec n le nombre d'appels de la fonction. Comme pour chaque appel de ENUM_REC, nous appelons COLORIERETPROPAGER, nous obtenons une complexité $O(2^{NM} \cdot \max(N^5M, NM^5))$. Nous pouvons illustrer tout cela par le schéma ci dessous :



En dépit des quatre comparaisons réalisées dans le pire des cas, on peut conclure que pour la fonction ENUMERATION, nous obtenons une complexité en $O(\max(N^5 M, NM^5) + 2^{NM} \cdot \max(N^5 M, NM^5))$ donc en $O(2^{NM})$.

Question 14 :

Après avoir appliqué notre méthode ENUMERATION, voici le tableau des différents temps de résolutions que nous obtenons pour 20 tests différents :

Instances	0	1	2	3	4	5	6	7	8	9	10
Dimensions	4x5	5x5	20x20	13x37	25x25	25x27	30x30	31x34	40x35	50x50	99x99
Résolution Coloration	True										
Temps (ms)	0.424	0.865	75	50	110	122	359	158	284	3092	3075
Résolution Enumération	True										
Temps (ms)	0.400	0.785	75	59	125	124	377	165	286	3005	3022

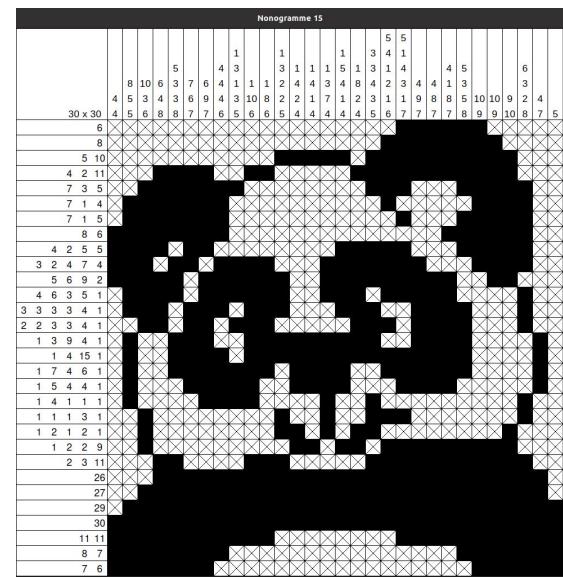
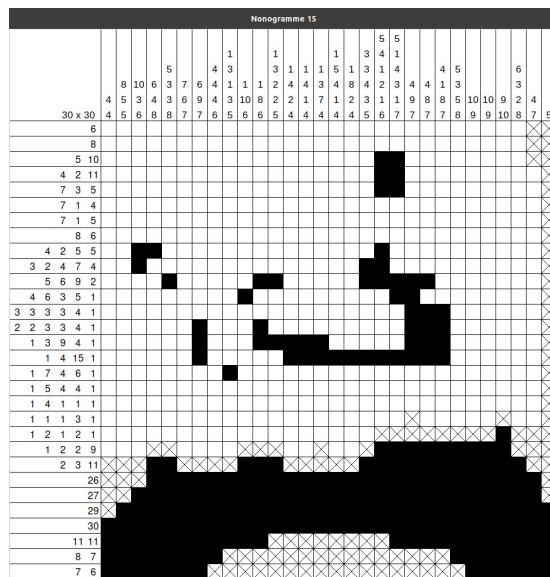
Instances	11	12	13	14	15	16	Total
Dimensions	2x4	33x28	45x45	30x38	30x30	35x50	
Résolution Coloration	None	None	None	None	None	None	None
Temps (ms)	0.209	252	293	215	113	333	8460
Résolution Enumération	True	True	True	True	True	True	True
Temps (ms)	0.337	279	380	243	277	21.51 s	28.47 s

Comme nous l'avions remarqué dans la première partie, les dix premières instances semblent se résoudre complètement avec la fonction COLORATION. Comme nous utilisons cette dernière dans ÉNUMÉRATION, il est donc normal que celle-ci résolve aussi toutes les instances de 0 à 10.

Cependant, contrairement à la première partie, COLORATION ne semble pas résoudre toutes les instances. Elle n'est pas capable de conclure sur les problèmes des instances 11 à 16. Nous voyons la limite de la méthode incomplète de résolution.

Néanmoins, la méthode ENUMERATION présente une conclusion différente. Cette dernière semble être capable de donner une réponse à toutes les instances grâce à l'énumération de tous les coloriages possibles, comme vous pouvez le voir sur l'instance 15 ci-dessous.

Enfin, nous pouvons souligner la complémentarité des deux méthodes. Pour l'instance sur la Joconde, il manque une dizaine de cases pour permettre à la fonction COLORATION de conclure. Ces cases sont très vite coloriées grâce à la seconde méthode. Mais il est important de relever que la méthode COLORATION reste indispensable à l'optimisation de la méthode ENUMERATION pour éviter de rentrer dans des branchements de l'arbre inutilement. Plus il y a de cases déjà posées dans la grille, moins il y aura d'appels récursifs, et donc plus le jeu sera résolu rapidement.



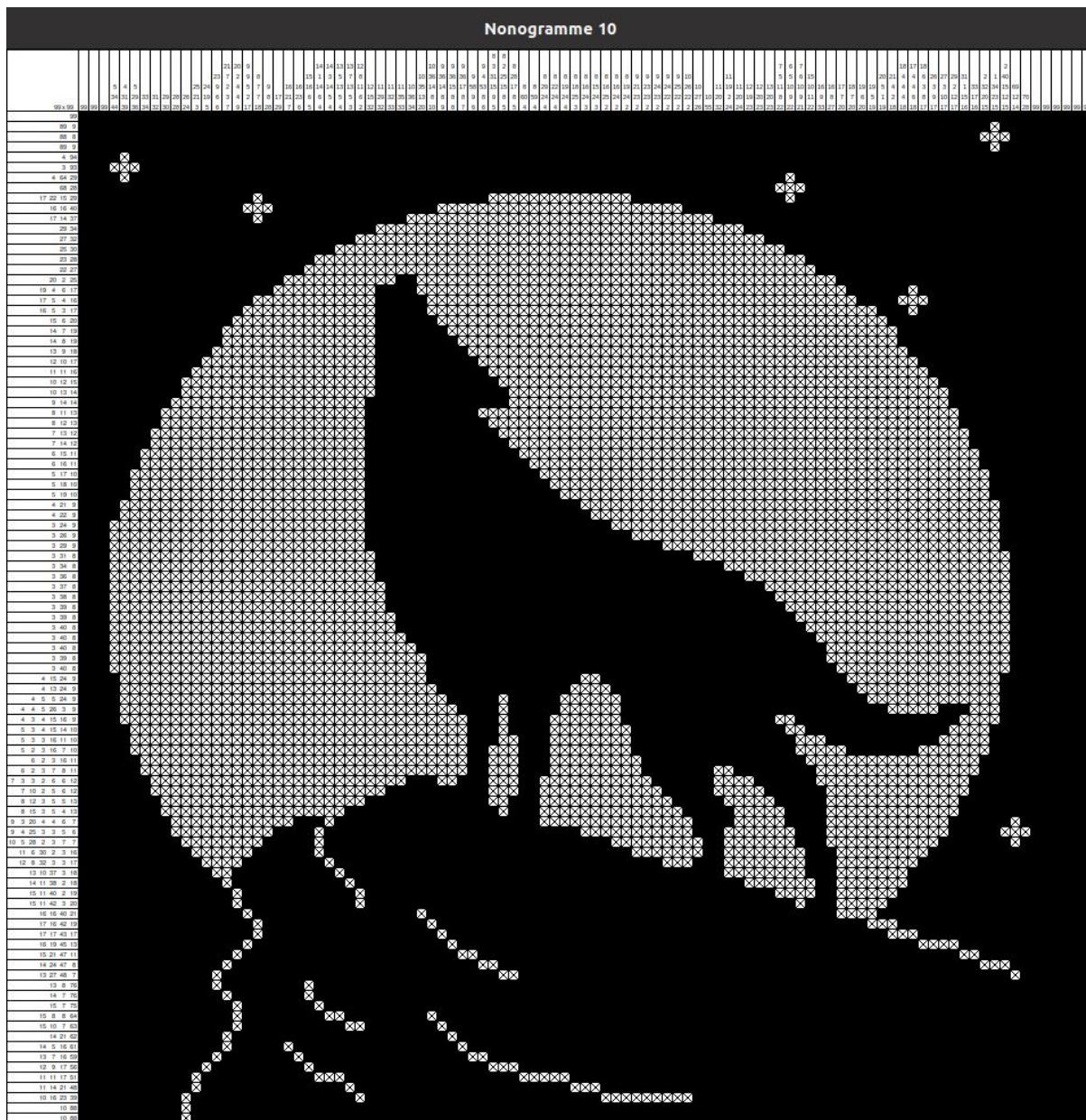
D'une part, nous retrouvons la grille après avoir appliqué la méthode COLORATION (image gauche). D'autres part, nous retrouvons la grille après avoir appliqué la méthode ENUMERATION (image droite). Nous pouvons admirer la beauté du panda.

Conclusion :

Dans ce projet de tomographie discrète, nous avons pu voir une stratégie permettant de résoudre le jeu du *Nonogramme* en appliquant deux méthodes complémentaires. D'une part, nous avons implémenté une méthode incomplète assez efficace en termes de complexité temporelle mais qui n'est pas capable de résoudre entièrement le jeu. D'autres part, nous avons écrit une fonction qui permet de résoudre complètement la grille en énumérant tous les coloriages possibles. Certes, cette fonction résoudra toujours la grille (si une solution existe) mais en raison de sa complexité exponentielle, il est préférable de l'associer à d'autres méthodes moins coûteuses en termes de temps.

Nous pouvons aussi souligner l'utilité de la programmation dynamique qui dispense énormément certains appels de fonctions, quitte à perdre un peu de mémoire. De plus, le *Python* et son évaluation paresseuse semble être un langage informatique parfaitement adapté à notre projet pour éviter tout appel de fonctions inutile. On pourrait s'interroger sur la complexité temporelle de notre algorithme avec un langage concurrent.

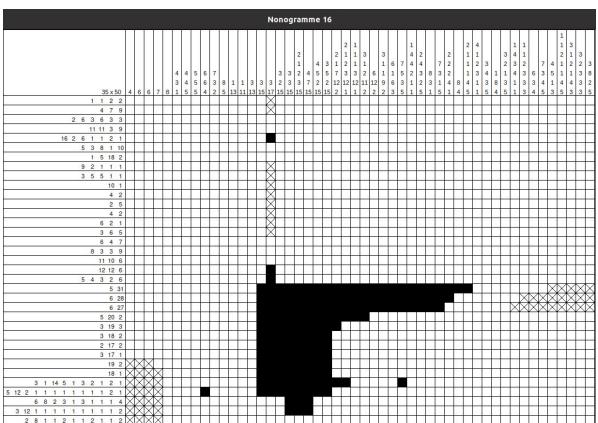
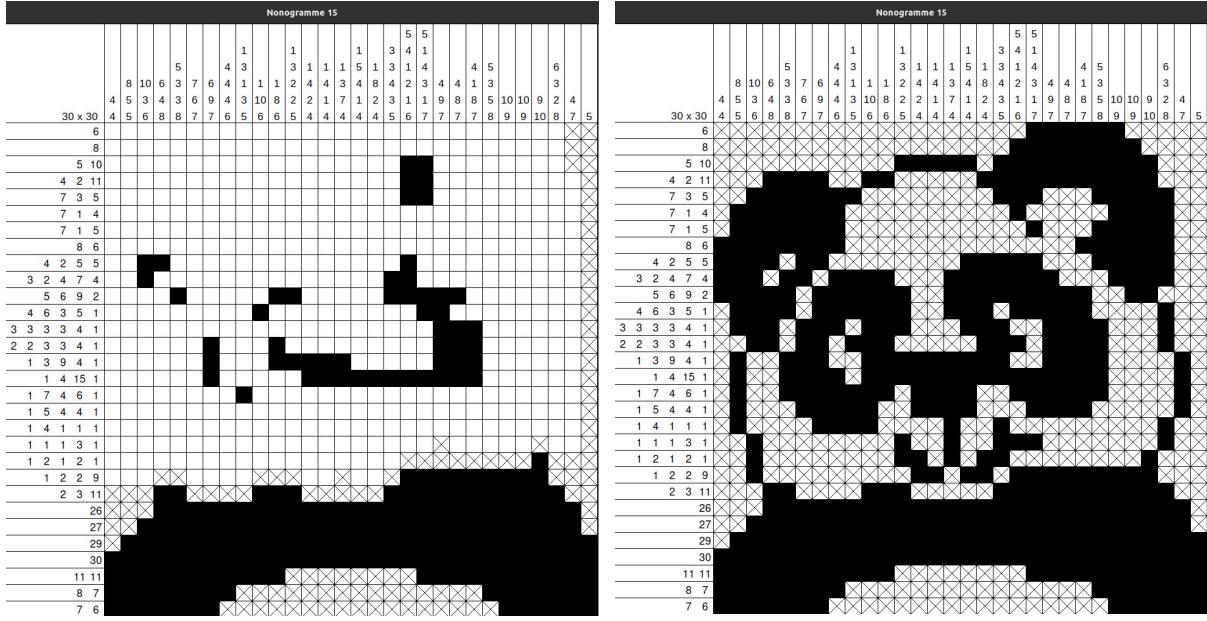
Il serait maintenant intéressant de pousser encore plus loin la stratégie vue d'en trouver de nouvelles à utiliser pour possiblement une optimisation de l'algorithme.



Pour la fonction COLORATION à gauche et pour la fonction ENUMERATION à droite :

Nonogramme 11				
? x 4	1	1	1	1
2				
1 1				

Nonogramme 11				
? x 4	1	1	1	1
2	X	X		X
1 1		X	X	



Bonus pour la victoire :

