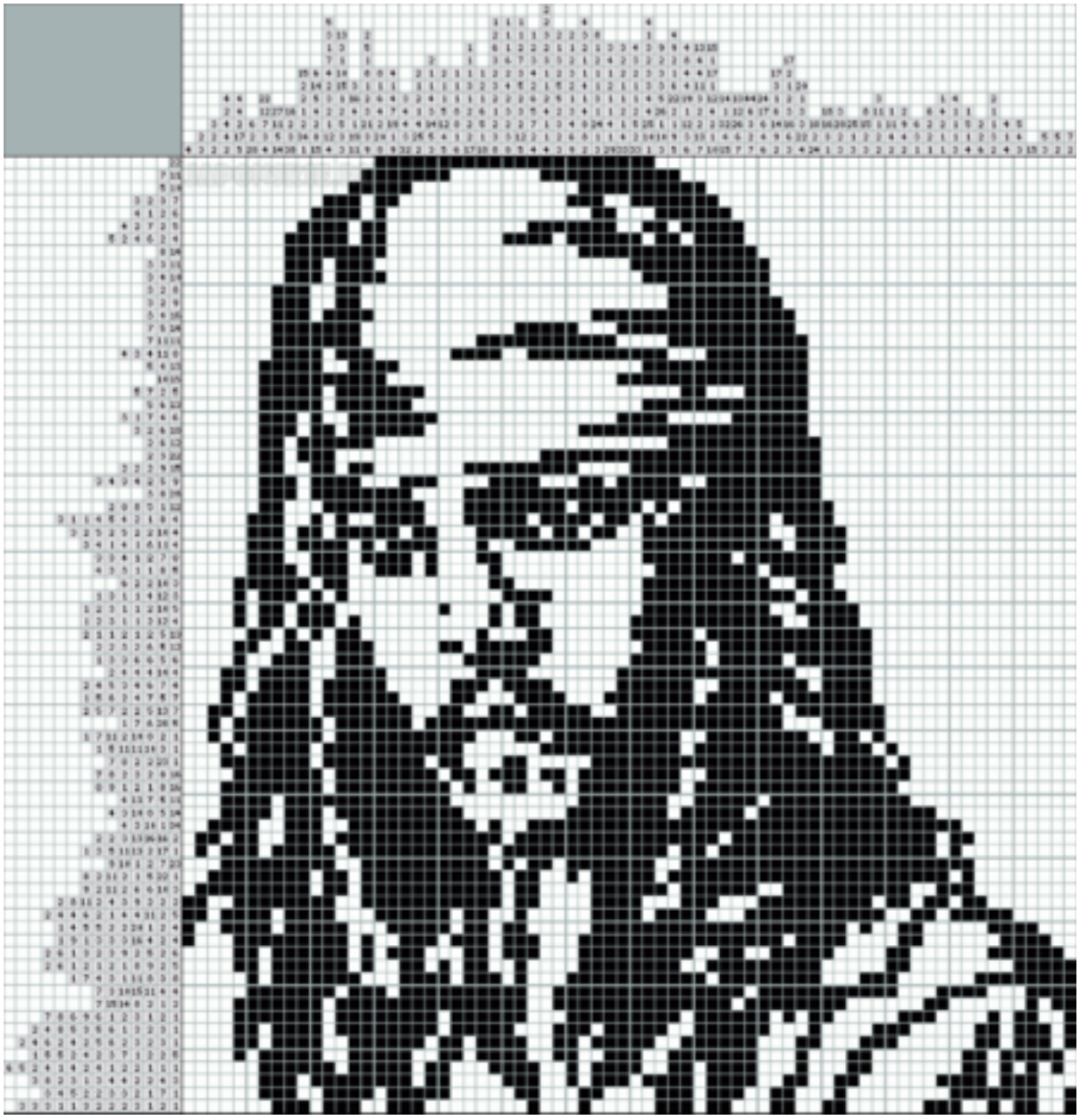
Projet MOGPL

Un problème de tomographie discrète



Lachiheb Sarah
Leroy Cassandre

Introduction :

Le but de ce projet est de construire, s'il en existe, une grille coloriée en noir ou en banc respectant des contraintes sur les lignes et colonnes de la grille. Ces contraintes correspondent à des séquences (s_1,…, s_k) de nombre entiers strictement positifs représentant un nombre de case devant être colorié en noir. Aussi, ces blocs doivent être tous séparés par, au moins, une case blanche.
Pour résoudre ce type de problème nous allons passer par la récursivité, pour ensuite introduire la programmation dynamique et finir par la programmation linéaire entière.

1.1 - Raisonnement par la récursivité :

Dans cette première partie, nous allons définir un algorithme permettant de déterminer s'il existe un coloriage possible d'une ligne, ou colonne, avec sa séquence associée. *On considère dans cette partie que les lignes et colonnes de la grille ne sont pas encore coloriées.*

- *Notation :*
  - **l(i)** : une sous-ligne dont la séquence associée est (s_1, s_2, …, s_k)
  - **T(i, j)** où j = (0,…, m-1) et l = (1, …, k) : fonction qui retourne vrai s'il est possible de colorier les j + 1 premiers cases de la ligne **l(i)** avec la sous-séquence (s_1, …, s_l).

1 - Questionnement : Si l'on a calculé tous les T(j,l), comment savoir s'il est possible de colorier la ligne l(i), entière avec la séquence entière ?

1 - Réponse : 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 donc possible de savoir si une ligne l(i) peut être coloriée avec sa séquence entière. Pour cela, il faut récupérer la valeur du 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 = s_k où s_k 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.

2 - Questionnement : Pour tous les cas de base, montrer si T(j, l), prend la valeur vrai ou faux ?

- *Rappel sur les cas de base :*
  - *1) Cas l = 0 (pas de bloc)*
  - *2) Cas l >= 1 (au moins un bloc)*
    - *(a) j < s(l) - 1*
    - *(b) j = s(l) - 1*
    - *(c) j > s(l) - 1*

2 - Réponse :

- Pour le **cas 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, nous constatons qu'il y aucune case coloriée, ainsi l = 0 est vérifié.

- Pour le **cas 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 le cas **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. (Je rappel qu'on considère que la grille est vide, c'est pourquoi on ne regarde que l'une des cases allant de 0 à j soit blanche, si c'était le cas, T(j, l) rendrait False)

- 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 = s_l-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).

3 - Questionnement : Exprimez une relation de récurrence permettant de calculer T(j, l) dans le cas 2c.

- *Remarque : On considère la case (i,j) de la sous-ligne l(i) coloriée en blanche ou en noir, mais les cases (i, j=(0,…, j-1) sont considérés vide.*

3 - Réponse : Pour le **cas l ≥ 1 et j > s(l) -1**, nous pouvons rencontrer 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).

4 - Questionnement : Codez l'algorithme.

4 - Réponse :

Cette fonction T(j, l) considère que la grille est non coloriée donc vide :

```
def T_first_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 retourn 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_first_without_color(j-(sequence[l-1])-1, l-1, ligne, sequence)
```

Cette fonctionT(j, l) considère que chaque ligne de la grille à sa case (i,j) coloriée et les cases (i, j=(0,.., j-1) sont non coloriées donc vide.

```python
def T_first_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 coloriée avec une sequence
                 donnée sinon retourn 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 considere que seul la case (i,j) est coloriée en blanc ou en noir :
        if j > sequence[l-1]- 1 :
            # Cas 1 : la case (i,j) est blanche :
            if ligne[j] == CASE_WHITE:
                return T_first_color(j-1, l, ligne, sequence)
            # Cas 2 : la case (i, j) est noir :
            if ligne[j] == CASE_BLACK :
                return T_first_color(j-sequence[l-1]-1, l-1, ligne, sequence)
```

1.2 - Généralisation (Programmation dynamique) :

5 - Questionnement : Modifiez chacun des cas de l'algorithme précédent afin qu'il prenne en compte les cases déjà coloriés ?

5 - Réponse :

Pour le cas **l = 0** : (les modifications)

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

Pour le cas **l ≥ 1 et j < s(l) - 1** : (les modifications)

- Il n'y aucune modification pour ce cas là. C'est à dire, si les deux conditions sont vérifiés alors le T(j, l) retourne False.

Pour le cas **l == 1 et j = s(l) - 1** : (les modifications)

- On vérifie si tout les cases allant (i, j=(0,…, j)) sont, soit noir, 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 le cas **l > 1 et j = s(l) - 1** : (les modifications)

- Il n'y a aucun changement pour ce cas. 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 le cas **l ≥ 1 et j > s(l) - 1** avec une case blanche : (les modifications)

- 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 le cas **l ≥ 1 et j > s(l) - 1** avec une case noir : (les modifications)

- 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 se doit d'observer s'il n'y 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(=Top-down) ou suivant(=Bottom up) 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.

6 - Questionnement : Codez l'algorithme.

6 - Réponse : Nous avons programmés deux type de programmation dynamique :

- une **forme récursive « Top down »** dite de mémorisation, dans ce cas, on utilise directement la formule de récurrence et lors d'un appel récursif, avant d'effectuer un calcul on regarde dans le tableau de mémoire cache (dictionnaire) si ce travail n'a pas déjà été effectué.

- une **forme itérative « Bottom up »** où on résout d'abord les sous problèmes de plus "petite taille", puis ceux de la taille "d'au dessus", etc… Au fur et à mesure on stocke les résultats obtenus dans un tableau de mémoire cache et on continue jusqu'à la taille voulue.

Programmation dynamique sous forme Bottom up :

```
def Bottom_Up_generalization(ligne, sequence):
    """ Entree          :ligne(correspond à une ligne de la matrice du probleme)
                        :sequence(correspond à la sequence de bloc associe à la ligne)

        Retourne        :Retourne vrai si on peut colorier la ligne avec la sequence de bloc associe sinon False.

        Fonctionnement :Resolution de tout les sous problèmes(=sous-sequences de la sequence) pour resourdre le probleme
                         principal(=la sequence entiere).
    """
    # Nombre de case dans la ligne:
    nb_case = len(ligne)
    # Nombre de bloc dans la sequence (+1 car on part de la sous-sequence s0):
    nb_sous_sequence = len(sequence) + 1
    # Tableaux a deux dimensions de la memoire cache :
    memoire_cache = np.zeros((nb_case,nb_sous_sequence),dtype=bool)
    # Algorithme :
    for j in range(nb_case):
        for l in range (nb_sous_sequence) :
            # Si entre [0,j] il y a aucun bloc Noir:
            if l == 0:
                memoire_cache[j][l] = np.all(ligne[:j+1]<=0)
            elif j < sequence[l-1]-1 :
                memoire_cache[j][l] = False
            # Si un seul bloc et une sequence egale au nombre de case entre [0,j] avec aucun bloc Blanc:
            elif l == 1 and j == sequence[l-1]-1:
                memoire_cache[j][l] = np.all(ligne[:j+1]>=0)
            elif l > 1 and j == sequence[l-1]-1:
                memoire_cache[j][l] = False
            elif j > sequence[l-1]-1 :
                # Case White :
                if ligne[j] == CASE_WHITE:
                    memoire_cache[j][l] = memoire_cache[j-1][l]
                else : # Ici la case est soit Black ou White :
                    # 1 er partie : Si la case est Empty
                    if ligne[j] == CASE_EMPTY and memoire_cache[j-1][l] :##
                        memoire_cache[j][l] = memoire_cache[j-1][l]
                    else :
                        # Decomposition de la condition en plusieur variable :
                        insert_bloc = np.all(ligne[j-sequence[l-1]+1:j+1] >= 0) ##
                        one_bloc = l==1 and np.all(ligne[:j-sequence[l-1]+1] <=0)
                        several_bloc = (ligne[j-sequence[l-1]] <= 0) and memoire_cache[j-sequence[l-1]-1,l-1]
                        # Condition final pour validation :
                        condition = insert_bloc and (one_bloc or several_bloc)
                        # Case Black :
                        if ligne[j] == CASE_BLACK or ligne[j] == CASE_EMPTY:
                            memoire_cache[j][l] = condition
    if memoire_cache[j][-1] and np.all(ligne[j+1:] <= 0 ):
        return True
    return memoire_cache[-1,-1]
```

Programmation dynamique sous forme Top down :

```python
def Top_Down_generalization(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_generalization(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_generalization(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_generalization(ligne[:j - 1],sequence,memoire_cache)
    return memoire_cache[(j,l)]
```

```python
def Top_Down_generalizationInit(ligne, sequence):
    return Top_Down_generalization(ligne, sequence,{})
```

1.3 - Propagation :

7 - Questionnement: Codez l'algorithme de propagation.

7- Réponse :

```python
def Test_dynamic(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_generalizationInit(grille[i],lig) and Top_Down_generalizationInit(np.transpose(grille)[j], col)
            test_black = Bottom_Up_generalization(grille[i],lig) and Bottom_Up_generalization(np.transpose(grille)[j], col)
            # Test avec une case (i,j) White :
            grille[i][j] = CASE_WHITE
            #test_white =  Top_Down_generalizationInit(grille[i],lig) and Top_Down_generalizationInit(np.transpose(grille)[j], col)
            test_white =  Bottom_Up_generalization(grille[i],lig) and Bottom_Up_generalization(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) # Permet au chien de s'afficher en entier
            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
```

```python
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 = Test_dynamic(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 = Test_dynamic(np.transpose(grille), sequences_colonnes, sequences_lignes, colonnesAVoir)
        colonnesAVoir = set()
    return grille
```

1.4 - Test : 8

7 - Questionnement:

Le tableau des temps d'exécution des instances de 1.txt à 10.txt en mode Top Down et Bottom up :

instances	Temps (seconde) Top Down	Temps (seconde) Bottom up
1	0.00875401496887207	0.0340728759765625
2	1.303440809249878	5.879356861114502
3	0.686438798904419	2.7938480377197266
4	1.5500149726867676	11.467868089675903
5	1.7756149768829346	5.297926902770996
6	6.207081079483032	15.842377424240112
7	1.9807569980621338	7.9064881801605225
8	4.035547256469727	16.09430193901062
9	51.04985189437866	227.22625494003296
10	82.30248689651489	247.10261917114258

Nous constatons, sur le tableau ci-dessus, que les temps d'exécution pour l'algorithme Bottom up sont plus long que ceux de Top Down. Cependant, cela ne devrait pas être le cas, les deux algorithmes on en effet une complexité linéaire. On pense que cela peut être dû au fait que nous n'avons pas utilisé la même structure de stockage. Pour Top Down un dictionnaire et pour Bottom up un tableau.

La grille obtenue pour l'intenses 9.txt est

9 - Questionnement : Appliquer votre programme sur l'instance 11.txt. Que remarquez-vous ?

9 - Réponse : On remarque, pour la grille de l'instance 11.txt, que chaque case (i,j) de la grille est indécidable. On peut se demander pourquoi, car lorsqu'on regarde le temps d'exécution pour cette instance dans le mode Top Down, on observe un temps de 0.0027 secondes, ce qui est très rapide. Ce temps s'explique par le fait que lorsqu'on applique la fonction Test_dynamic 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 elle n'a rien ajouter aux colonnes a voir. De même pour l'appel de fonction Test_dynamic sur les colonnes, elle retourne un ensemble de ligne à voir vide. Donc le programme se termine sans avoir résolu une seule case (i,j) de la grille.

2 - La PLNE à la rescousse :

2.1 - Modélisation :

- *Notation :*
  - Xi,j (i=0,…N-1, j=0,…,M-1) : vaut 1 si la case (i,j) est noir sinon 0 si elle est coloriée en blanc.
  - Yi,j,t vaut 1 si la t ième bloc de la ligne l(i) commence à la case (i,j) et 0 sinon.
  - Zj,i,t vaut 1 si la t ème bloc de la colonne c(i) commence à la case (j,i) et 0 sinon.

10 - Questionnement : Formulez une contrainte qui exprime le fait que si le $t^{ième}$ bloc de la ligne l(i), commence à la case (i,j), alors les cases (i, j) à (i, j+s_t-1) sont noires. De même pour les colonnes.

10 - Réponse :
- Phase de raisonnement : On sait que $X_{i,j}$ est égale à 1 si la case (i,j) est noire. Si on veut savoir si un bloc de valeur s(t) est présent dans une ligne, il faut que la somme des $X_{i,j}$ consécutif soit égal à s(t). Pour exprimer la contrainte de la question 10, il faut que si le bloc s($t^{ième}$) commencent à la case (i,j), alors le produit de $Y_{i,j,t}$ et de s(t) moins la somme des $X_{i,j}$,où j = (j,…,j +s_t-1) doit être inférieur où égale à zéro, alors la contrainte est vérifiée. C'est à dire qu'on a bien le $t^{ième}$ bloc de la ligne l(i) qui commence de la case (i,j) et termine à la case (i, j+st-1).

Par contre, elle n'est pas vérifiée si $Y_{i,j,t}$ est égale zéro, ou s(t) moins la somme des $X_{i,j}$ est différente de zéro.

On veut montrer que si **$Y_{i,j,t}$ = 1 alors $\sum X_{i,j}$ où j=(0,…, j+s(t)-1) = s(t)** (cela traduit le fait que si le $t^{ième}$ bloc de la ligne l(i) commence à la case (i,j), alors les cases (i, j) à (i, j+s_t-1) sont noires).

Contrainte pour les lignes :
- Contrainte linéaire : **Pour tout i,j ,t ($Y_{i,j,t}$ x s(t)) ≤ $\sum X_{i,j}$, où j=(0,…, j+s(t)-1)**

Contrainte pour les colonnes :
- Contrainte linéaire : **Pour tout i,j,t ($Z_{j,i,t}$ x s(t)) ≤ $\sum X_{j,i}$, où i=(0,…, j+s(t)-1)**

11 - Questionnement : Formulez une contrainte qui exprime le décalage nécessaire entre le début des blocs s(t) et s(t+1) de la ligne l(i). Si le $t^{ième}$ bloc commence à la case (i,j), alors le bloc (t +1)^{ième} ne peut pas commencer avant la case (i, j+s(t)+1).

11 - Réponse :
- Phase de raisonnement : La contrainte sur le décalage exprime le fait qu'entre le bloc s(t) et les blocs s(t+1), il doit y avoir au moins un blanc. Dans cette contrainte, il faut utiliser les variables $Y_{i,j,t}$ pour les lignes, ou $Z_{j,i,t}$ pour les colonnes. Ceci permet d'exprimer le fait qu'il n'y ait pas de bloc (t+1), qui commence dans l'intervalle des cases (i,j) à (i,j+st) pour $Y_{i,j,t}$ ou $Z_{j,i,t.}$

Contrainte pour les lignes : Il faut récupérer toutes les variables $Y_{i,j,t+1}$ qui sont dans l'intervalle des cases (i,j) allant à (i,j+st). Et la somme des $Y_{i,j,t+1}$ avec j=(j,..,j+s(t)) et $Y_{i,j,t}$ doit valoir 1, expriment le fait qu'il y ait qu'un bloc qui commence dans cette intervalle et que ce soit bien $Y_{i,j,t}$.

- Contrainte linéaire : **Pour i,j,t où $Y_{i,j,t} + \sum Y_{i,k,t+1}$ (où k allant de 0 à j+s(t)) ≤ 1**

Contraintes pour les colonnes :

- Contrainte linéaire : **Pour i,j,t où $Z_{j,i,t} + \sum Z_{j,k,t+1}$ (où k allant de 0 à i+s(t)) ≤ 1**

12- Questionnement : Formulez alors le problème de coloriage comme un PLNE.

12 - Réponse :

Les contraintes actuelles ne sont pas suffisantes il faut préciser, pour chaque ligne, qu'il y a qu'un seul $t^{ième}$ bloc de la ligne qui commence à une case (i,j).Cela se traduit par:

- Pour tout i,j,t $\sum_{i=(0,..,N-1)} \sum_{t=(0,..,nb\ de\ bloc\ sur\ l(i))} \sum_{j=(0,...,M-1)} Y_{i,j,t} = 1$

De même pour les colonnes, il y a qu'un seul $t^{ième}$ bloc de la colonne qui commence à une case (j,i), cela se traduit par:

- Pour tout i,j,t $\sum_{j=(0,..,M-1)} \sum_{t=(0,..,nb\ de\ bloc\ sur\ c(j))} \sum_{i=(0,...,N-1)} Z_{j,i,t} = 1$

Minimiser $z = \sum_{i=(0,..,N-1)} \sum_{j=(0,..,M-1)} X_{i,j}$

Contraintes :

$Y_{i,j,t} + \sum Y_{i,k,t+1}$ (pour k allant de 0 à j+s(t)) ≤ 1 (Contrainte sur les décalages des lignes)

$Z_{j,i,t} + \sum Z_{j,k,t+1}$ (pour k allant de 0 à i+s(t)) ≤ 1 (Contrainte sur les décalages des colonnes)

$Y_{i,j,t} \times s(t) \leq \sum X_{i,j}$ (où j=(0,…, j+s(t)-1)) (Contrainte sur les blocs des lignes)

$Z_{j,i,t} \times s(t) \leq \sum X_{j,i}$ (où i=(0,…, j+s(t)-1) (Contrainte sur les blocs des colonnes)

$\sum_{i=(0,..,N-1)} \sum_{t=(0,..,nb\ de\ bloc\ sur\ l(i))} \sum_{j=(0,...,M-1)} Y_{i,j,t} = 1$

$\sum_{j=(0,..,M-1)} \sum_{t=(0,..,nb\ de\ bloc\ sur\ c(j))} \sum_{i=(0,...,N-1)} Z_{j,i,t} = 1$

$\sum\sum\sum Y_{i,j,t}$ (où i=(0,..,N-1) j=(0,..,M-1) t=(0,…,nombre de bloc dans séquence l(i)) ∈ {0,1}

$\sum\sum\sum Z_{j,i,t}$ (où j=(0,..,N-1) i=(0,..,M-1) t=(0,…,nombre de bloc dans séquence c(i)) ∈ {0,1}

$\sum\sum X_{i,j}$ (où i=(0,..,N-1) j=(0,..,M-1)) ∈ {0,1}

2.2 - Implantation et tests :

13 - Questionnement : Exprimez de manière générale, pour une ligne l(i), un intervalle hors duquel le $l^{ième}$ bloc l ∈ {1…k} ne peut commencer.

13 - Réponse : Afin de limiter le nombre de variable $Y_{i,j,t}$ et $Z_{j,i,t}$, on va définir un raisonnement permettant de le réduire en détectant des intervalles impossibles, pour une suite de bloc sur des lignes ou des colonnes données.

Prenons un exemple pour expliquer notre concept :

Si on prend une ligne de 4 cases avec une contrainte composée de deux blocs $s_1 = 1$ et $s_2 = 1$, en excluant pas les intervalles impossibles, nous aurions :

- $y_{0,0,0} + y_{0,1,0} + y_{0,2,0} + y_{0,3,0} = 1$
- $y_{0,0,1} + y_{0,1,1} + y_{0,2,1} + y_{0,3,1} = 1$

En excluant les intervalles impossible, nous nous retrouverons avec :

- $y_{0,0,0} + y_{0,1,0} = 1$
- $y_{0,1,1} + y_{0,2,1} = 1$

14 - Questionnement : Implanter la résolution du PLNE.

14 - Réponse : Dans le fichier nommé PLNE.py, le code de la programmation linéaire entière se trouve dans ce fichier.

15 - Questionnement : Résolvez les instances 1.txt à 16.txt avec un timeout de 2 minutes. Donnez les temps de calcul dans un tableau.

15 - Réponse :

Instances	Temps (secondes) PLNE	Temps (secondes) Top Down
1	0.03600788116455078	0.00875401496887207
2	4.398514032363892	1.303440809249878
3	1.1822431087493896	0.686438798904419
4	13.643924236297607	1.5500149726867676
5	18.109773874282837	1.7756149768829346
6	182.06448006629944	6.207081079483032
7	28.693341970443726	1.9807569980621338
8	53.35555291175842	4.035547256469727
9	TimeOut	51.04985189437866
10	TimeOut	82.30248689651489
11	0.03855299949645996	0.00875401496887207
12	231.39489603042603	4.342382192611694
13	23.348441123962402	6.14825701713562
14	272.4006540775299	3.092871904373169
15	38.874698877334595	2.1152939796447754
16	TimeOut	5.886656999588013

- *Notation : En rouge sont représentés les instances non complète. Et en orange les instances ayant dépassées le TimeOut de 2 minutes.*

On remarque, dans la version programmation linéaire, 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.
Cependant, on constate, sur les instances résolus par la programmation dynamique, qu'elle est en général plus rapide que la programmation linéaire.

16.1 - Questionnement : Implantation d'une méthode globale.

16.1 - Réponse :

Instance	Méthode globale	PLNE
1	0.02628779411315918	0.03600788116455078
2	0.4444580078125	4.398514032363892
3	0.23930907249450684	1.1822431087493896
4	0.4051990509033203	13.643924236297607
5	0.310715913772583	18.109773874282837
6	0.653588056564331	182.06448006629944
7	0.40755200386047363	28.693341970443726

Instance	Méthode globale	PLNE
8	0.5676510334014893	53.35555291175842
9	3.7191269397735596	TimeOut
10	7.508529901504517	TimeOut
11	0.035759925842285156	0.03855299949645996
12	0.783998966217041	231.39489603042603
13	1.1164989471435547	23.348441123962402
14	0.6564428806304932	272.4006540775299
15	9.798634052276611	38.874698877334595
16	1077.8669519424438	TimeOut

Remarque : Le code de la méthode globale se trouve dans le dossier code_python et se prénomme ***méthode_globale.py***.

Dans cette partie, on comptabilise le temps à partir de l'appel de la fonction PLNE qui se trouve après la propagation. Plus précisément, on mesure le temps de l'appel de fonction du PLNE et non de la propagation.

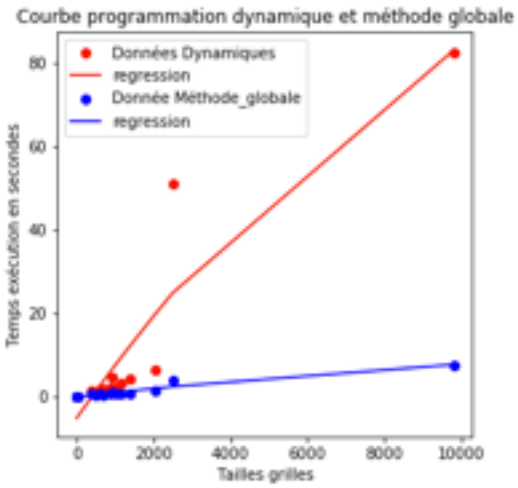
La méthode globale fonctionne de la manière suivante, elle appel la fonction de propagation (qui remplis totalement la grille des instances 0 à 10 et partiellement la grille des 11 à 16). Puis, on transmet cette grille à la fonction PLNE qui se charge de résoudre complètement celle-ci. Dans la fonction PLNE, nous avons rajouté plusieurs contraintes, qui prenne en compte les case(i,j) déjà coloriées :

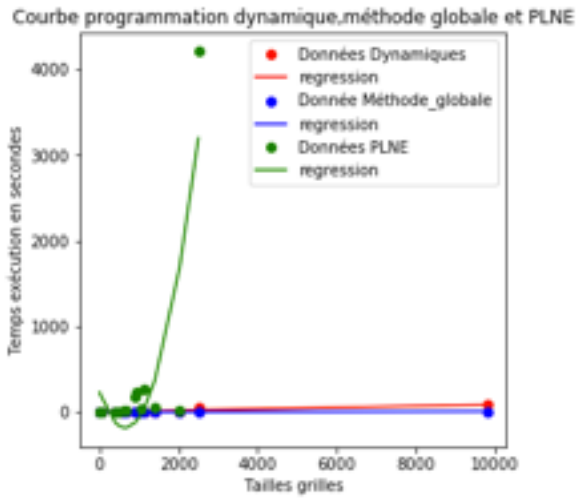
- Lorsque la case (i,j) de la grille est noire, on rajoute une contrainte de type $\mathbf{X_{i,j} = 1}$.
- Lorsque la case (i,j) de la grille est blanche, on rajoute une contrainte de type $\mathbf{X_{i,j} = 0}$.
- Lorsque la case (i,j) est blanche, on parcours les contraintes des lignes et on ajoute la contrainte de type $\mathbf{Y_{i,j,t} = 0}$, car on sait que cette case n'est pas le commencement d'un bloc t.
- De même, lorsque que la case (i,j) est blanche, on parcours les contraintes des colonnes et on ajoute la contrainte de type $\mathbf{Z_{j,i,t} = 0}$.

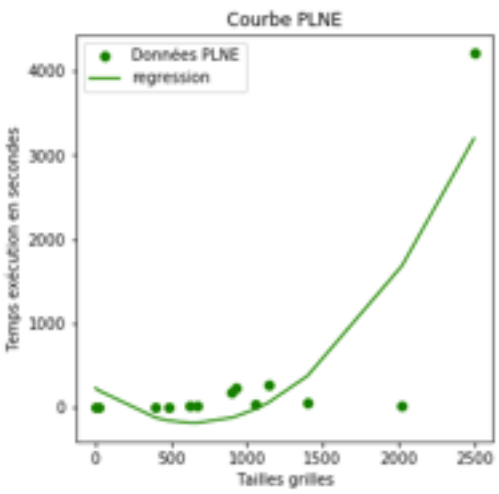
Avec les changements insérés dans le programme du PLNE, on constate, d'après le tableau ci-dessus, que la combinaison des deux fonctions qui créée la méthode globale rend le PLNE très performant. En effet, les 15 premières instances se finissent maintenant avant le TIMEOUT, cependant, la dernière instance reste compliquée à résoudre avec un TIMEOUT de 2 minutes.

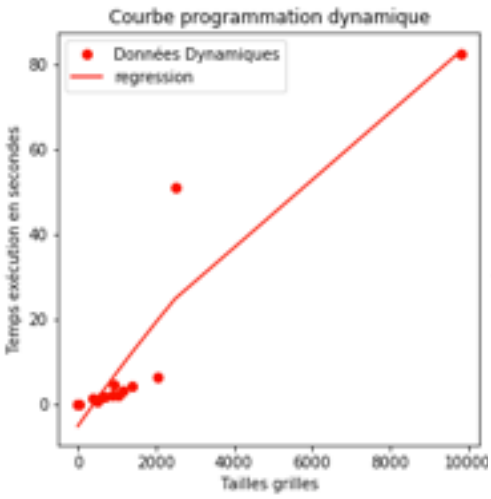
16.2 - Questionnement : Evaluer la complexité.

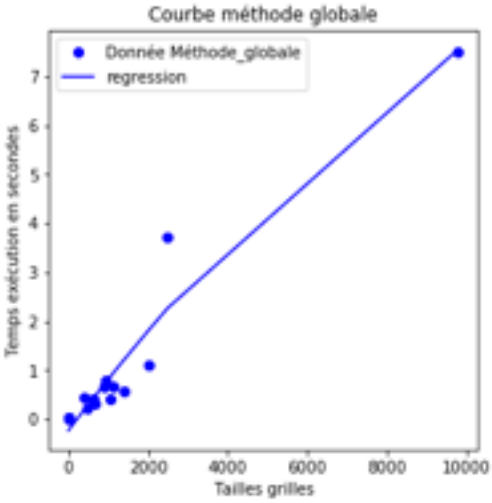
16.2 - Réponse : Afin d'évaluer la complexité de nos différents algorithme, nous avons tracés les courbes du temps d'exécution en fonction de la tailles des grilles des instances qui sont placés ci-dessous :











La complexité de la programmation dynamique et de la méthode globale nous semble linéaire, tandis que la complexité du programme PLNE semble essentiellement exponentielle en vue de la courbe et du temps d'exécution.

Conclusion :

Au cours de ce projet, nous avons pris conscience de l'importance de la programmation dynamique qui nous a permis d'évité 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.

Pour résoudre ce problème de case à choix multiple, nous avons implémentés sur Gurobi un programme linéaire résolvant toute les grilles des instances données par le projet, mais il y avait une contre partie qui est celle du temps d'exécution, elle était beaucoup plus importante que celle de la programmation dynamique.

Pour résoudre le problème de ce temps d'exécution, nous avons mis en place un programme nommé méthode globale, faisant l'alliance de la programmation dynamique et du PLNE. Cette alliance à permis de diminuer considérablement le temps d'exécution du programme tout en résolvant toutes les grilles.