Projet : Un problème de tomographie discrète

MOGPL : Modélisation et Optimisation par les Graphes et la Programmation Linéaire

Realisé par
BECIRSPAHIC Lucas
et
ADOUM Robert

Contents

I. Raisonnement par programmation dynamique

1 - Première étape

Question 1:

Si l'on a calculé tous les $T(j,l)$, pour savoir si il est possible de colorier la ligne l_i entière avec la séquence entière il suffit de de regarder $T(m-1,k)$, si ce dernier vaut vrai alors il est possible de colorier la ligne entière avec la séquence entière. Si il vaut faux alors ce n'est pas possible.

Question 2:

- Cas $l = 0$, $j \in \{0, ..., m-1\}$: Vrai
  justification : Si il n'y a pas de bloc à poser, alors un coloriage est toujours possible.
- Cas $l \geqslant 1$, $j < s_l - 1$: Faux
  justification : Si le nombre de cases dont on dispose est inférieur à la taille du bloc, on ne peut pas le poser donc faux.
- Cas $l \geqslant 1$, $j = s_l - 1$:
  - Si $l = 1$ alors Vrai
  - Si $l \neq 1$ alors Faux

  justification : Si le bloc fait exactement la taille de nos cases, on regarde si il y a un unique bloc à poser. Si ce n'est pas le cas, on renvoi faux.

Question 3:

La relation de récurrence permettant de calculer $T(j,l)$ est la suivante:

$T(j,l) = T(j-(s_l+1), l-1) \vee T(j-1,l)$

En effet si l'on se trouve à la case j qui est noire, et que l'on veut savoir si il est possible de colorier la sous séquence $(s_1, ..., s_l)$ il faut pouvoir colorier s_l case(s) et laisser une case de séparation entre les colorations de s_{l-1} et s_l, il faut donc regarder si l'on peut colorier la ligne de la case 0 à $j - s_l - 1$ avec la sous séquence $(s_1, ..., s_{l-1})$.
En revanche si la case j est blanche, il n'est pas possible de placer le bloc par consequent on regarde si il est possible de placer la sequence sur les bloc précédents, ce qui s'exprimer par la formule : $T(j-1,l)$

Question 5:

1) Dans le cas, ou l'on a pas de bloc, il faut vérifier qu'aucune case n'est coloriée.
2.a) Si il n'y a pas la place pour mettre un bloc, c'est toujours faux peu importe la ligne
2.b) Si $j = sl - 1$ alors il faut vérifier que l'on peut poser le bloc, c'est à dire il n'y a pas de cases noires sur les cases considérées.
2.c)

- Si la première case est blanche, on ne peut pas poser le bloc par conséquent on regarde $T(j-1,l)$
- Si la case j est noire, on regarde si on peut placer le bloc de manière correcte, c'est à dire pas de blanc sur l'emplacement et un blanc apres et avant pour s'assurer que les blocs son bien séparés.
- Si la case n'est pas encore coloriée, on traite les deux cas de figures précédent et il suffit qu'un seul soit vrai pour que l'on considère $T(j,l)$ vrai.

Question 8:

instances	nbCases	time
0	20	0.00042200088501
1	25	0.000617027282715
2	400	0.117752075195
3	481	0.0961720943451
4	625	0.182909011841
5	675	0.199213027954
6	900	0.51091504097
7	1054	0.300116062164
8	1400	0.43498301506
9	2500	5.42304491997
10	9801	8.71296691895

Figure 0.1: Tableau représentant les résultats de la programmation dynamique

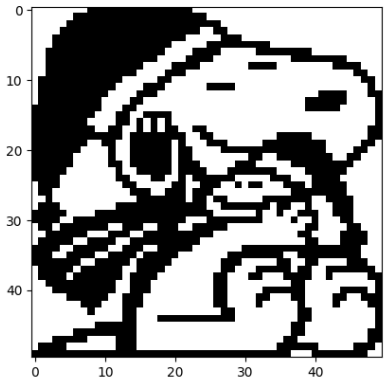

Figure 0.2: Grille de l'instance numéro 9

Question 9 En appliquant notre programme sur l'instance 11, on observe qu'en dépit de la petite taille de l'instance notre algorithme ne colorie rien. En effet quand une case peut être coloriée à la fois en blanc et en noir notre algorithme ne fais rien. Si la couleur d'une case ne peut être déterminée de manière exacte grace aux contraintes elle ne sera pas coloriée. Ce qui explique pourquoi notre algorithme ne colorie pas correctement l'instance 11.
Une solution à ce problème est d'implémenter un algorithme de backtracking qui une fois la coloration effectuée, observe toutes les cases non coloriées et leur affecte 0 et 1 arbitrairement puis on relance coloration avec la nouvelle grille. On réitère jusqu'à obtenir une grille complète (dans ce cas fin de l'algorithme) ou une grille insolvable. Si le grille ne peut pas être résolue , on retourne jusqu'à l'affectation la plus récente et on prend l'autre couleur. On notera que cette algorithme prend beaucoup plus de temps pour résoudre les grilles, une autre approche est d'utiliser la PLNE.

1. Initialisation : A <- coloration(M)
2. Tant que la grille n'est pas complète et que le nœud parent n'a pas testé toutes les couleurs:
   a) Si coloration(A) != False:
      Alors choisir arbitrairement une case incomplète c et lui attribuer une couleur
   b) Si coloration(A) == False:
      Alors en retourne à la grille précédente et on attribue l'autre couleur.
   c) A <- coloration(A)

remarque : Cette algorithme n'a pas été implémenté néenmoins nous allons procéder à une analyse de complexité pire cas. Dans le pire des cas, chaque case restante est examinée 2 fois. Soit n le nombre de cases restantes. On a donc 2^n itérations, qui appellent toutes la fonction coloration. Cette algorithme est donc de complexité exponentielle.

II. La PLNE à la rescousse

1 - Modélisation

Question 10:

On veut avoir une contrainte qui force nos cases à êtres noires si un bloc est posé. De plus si un bloc n'est pas posé sur la case (i,j), il ne faut rien imposer aux x_{ij} .

Par conséquent la condition est: $\sum_{k=j}^{j+s_t-1} x_{ik} \geq y_{ij}^t \times s_t$ c'est à dire :

$\sum_{k=j}^{j+s_t-1} x_{ik} - y_{ij}^t \times s_t \geq 0$

Cette contrainte est bien valide car si y_{ij} vaut 1 alors toutes les cases associées aux bloc doivent être noires. Et si y_{ij} vaut 0, on n'a pas de contrainte.

Avec le même raisonnement on a pour les colonnes: $\sum_{k=i}^{i+s_t-1} x_{kj} \geq z_{ij}^t \times s_t$

Question 11:

On cherche à exprimer une contrainte qui empêche de poser un bloc t+1 avant que le bloc t soit posé.

La condition est: $y_{ij}^t + \sum_{k=0}^{j+s_t} y_{ik}^{t+1} \leq 1$

Si y_{ij} vaut 0, la contrainte est toujours vraie . Et si y_{ij} vaut 1, aucun bloc k avec $k > t$ ne peut être posé avant $j + s_t$.

Avec le même raisonnement on a pour les colonnes: $z_{ij}^t + \sum_{k=0}^{i+s_t} z_{kj}^{t+1} \leq 1$

Question 12:

$$\text{Min z} = \sum_{i=0,j=0}^{N,M} x_{i,j}$$

$$s.c \begin{cases} \sum_{k=j}^{j+s_t-1} x_{ik} - y_{ij}^t \times s_t \geq 0 \mid \forall i \in \{0,1,2,...,N-1\}, \forall t \in \{1,2,...,k_i\} \\ \sum_{k=i}^{i+s_t-1} x_{kj} - z_{ij}^t \times s_t \geq 0 \mid \forall j \in \{0,1,2,...,M-1\}, \forall t \in \{1,2,...,k_j\} \\ y_{ij}^t + \sum_{k=0}^{j+s_t} y_{ik}^{t+1} \leq 1 \mid \forall i \in \{0,1,2,...,N-1\}, \forall t \in \{1,2,...,k_i-1\} \\ z_{ij}^t + \sum_{k=0}^{i+s_t} z_{kj}^{t+1} \leq 1 \mid \forall j \in \{0,1,2,...,M-1\}, \forall t \in \{1,2,...,k_j-1\} \\ \sum_{j=0}^{M-1} y_{ij}^t = 1 \mid \forall i \in \{0,1,2,...,N-1\}, \forall t \in \{1,2,...,k_i\} \\ \sum_{i=0}^{N-1} z_{ij}^t = 1 \mid \forall j \in \{0,1,2,...,M-1\}, \forall t \in \{1,2,...,k_j\} \\ x_{ij} \in \{0,1\} \mid \forall i \in \{0,1,2,...,N-1\}, \forall j \in \{0,1,2,...,M-1\} \\ y_{ij}^t \in \{0,1\} \mid \forall i \in \{0,1,2,...,N-1\}, \forall j \in \{0,1,2,...,M-1\}, \forall t \in \{1,2,...,k_i\} \\ z_{ij}^t \in \{0,1\} \mid \forall i \in \{0,1,2,...,N-1\}, \forall j \in \{0,1,2,...,M-1\}, \forall t \in \{1,2,...,k_j\} \end{cases}$$

2 - Implantation et tests

Question 13:

(N'oublions pas que j commence à 0 et termine à M-1)

- Pour une ligne l_i le l^{ieme} bloc ne peut commencer avant la case $(i, \sum_{n=1}^{l-1}(s_n+1))$, ni commencer après la case $(i, M - s_l - \sum_{n=l+1}^{k_i}(s_n+1))$.

- Pour une colonne l_j le l^{jeme} bloc ne peut commencer avant la case $(i, \sum_{n=1}^{l-1}(s_n+1))$, ni commencer après la case $(i, N - s_l - \sum_{n=l+1}^{k_i}(s_n+1))$.

On ajoute bien évidemment ces contraintes dans notre programme linéaire et mettant les y_{ij}^t et z_{ij}^t à 0, pour les i et j ne faisant pas parties de l'intervalle des blocs possibles.

Question 14:

L'implantation du plne résout parfaitement l'instance 11:

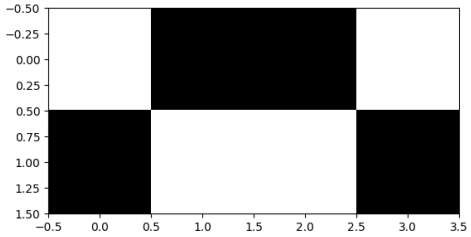

Figure 0.3: Grille de l'instance numero 11

Question 15:

instances	dynamique-time	plne-time	mix-time	nombre de cases
0	0.0004119873046875	0.0006170272827148438	0.0008900165557861328	20
1	0.0004858970642089844	0.0013060569763183594	0.0009407997131347656	25
2	0.11206889152526855	2.535233974456787	0.12527084350585938	400
3	0.08992695808410645	0.126816987991333	0.10281085968017578	481
4	0.1614398956298828	9.870957851409912	0.19457793235778809	625
5	0.19578003883361816	2.034217119216919	0.20783019065856934	675
6	0.5122568607330322	351.21364879608154	0.5453379154205322	900
7	0.3045821189880371	0.4993908405303955	0.32914304733276367	1054
8	0.44316697120666504	0.8764519691467285	0.498075008392334	1400
9	5.620426177978516	timeout	5.995676279067993	2500
10	9.328194856643677	timeout	10.79225778579712	9801
11	0.0003771781921386719	0.0005419254302978516	0.0009791851043701172	8
12	0.8796999454498291	162.9039990901947	1.081050157546997	924
13	1.0365591049194336	2.343104124069214	1.3090941905975342	2025
14	0.772252082824707	0.7354490756988525	0.824674129486084	1140
15	0.28334784507751465	18.160336017608643	7.860313892364502	900
16	0.7040119171142578	timeout	1650.8602929115295	1750

Figure 0.4: Comparaison des temps pour les 2 méthodes

On remarque que la programmation dynamique est la methode la plus rapide mais elle n'est pas capable de résoudre completement les instances a partir de 11. De plus le temps de calcul pour la PLNE semble plus dépendre des contraintes sur les lignes et les colonnes que du nombre de cases. Par exemple elle prend 162 secondes pour résoudre l'instance 12 de 924 cases alors qu'elle résoud l'instance 13 de 2025 cases en 2 secondes !
Finalement la methode mix qui consiste à initialiser la grille avec la programmation dynamique et à appliquer la PLNE est nettement plus rapide que la PLNE classique.

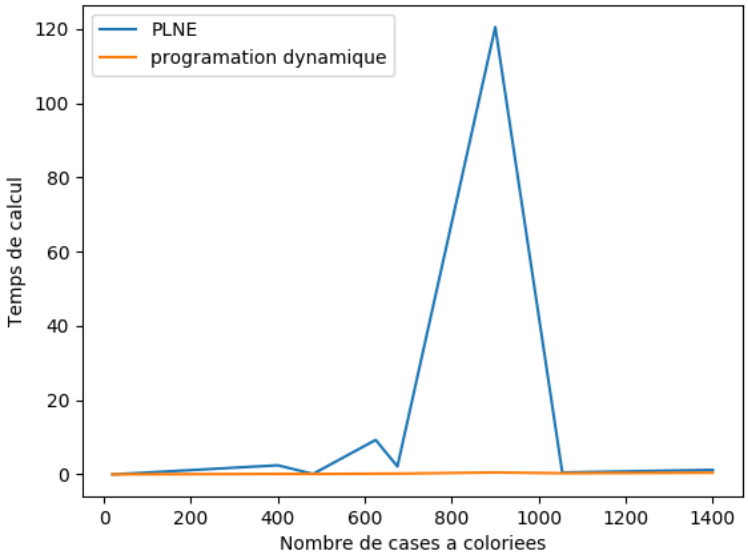

Figure 0.5: Comparaison des deux méthodes sur les 8 premières instances

Dans la figure 0.6, le gris correspond aux cases blanches, le noir aux cases noires et le blanc aux cases non déterminées

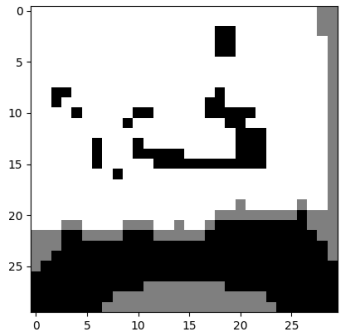

Figure 0.6: Images de l'instance 15 avec la programmation dynamique

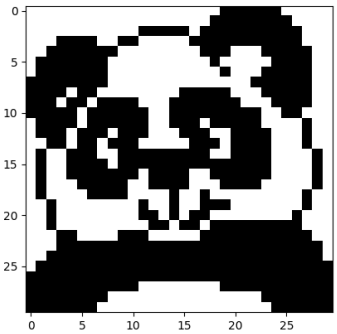

Figure 0.7: Image de l'instance 15 avec la PLNE

III. Pour aller plus loin

Analyse de complixté: On étudie la complexité pire cas de l'algorithme de programmation dynamique de la partie 1. Soient I et J le nombre de contraintes sur les lignes et colonnes. La fonction T(j,l) va calculer toutes les combinaisons de cases et séquences pour les lignes et les colonnes ce qui ce fait en $O(IN + JM)$ si l'on considère la version itérative.
L'algorithme de coloration est basé sur une boucle while. Dans le pire des cas on a une boucle pour chaque case de la grille. Donc la complexité de l'algorithme est $O(NM(IN + JM))$.
La complexité expérimentale semble bien plus faible.On fait l'hypothèse que nous avons une fonction polynomiale de la forme $f(x) = \lambda x^p$ avec λ et p appartenant à $\mathbb{R}$. De ce fait en passant au log on obtient : $log(f(x)) = log(\lambda) + p \times log(x)$ avec $log(\lambda)$ une constante. De ce fait, en tracant $log(f(x))$ en fonction de $log(x)$ et en calculant la pente on obtient p, c'est à dire la puissance de notre polynome. En procédant ainsi nous trouvons p=0.562258487574.

En implémentant la méthode qui consiste à commencer par la programmation dynamique puis la PLNE sur les cases non déterminées nous obtenons des meilleurs temps comme on peut le voir ci-dessous:

instances	plne-time	mix-time
11	0.000541925430298	0.00097918510437
12	162.90399909	1.08105015755
13	2.34310412407	1.3090941906
14	0.735449075699	0.824674129486
15	18.1603360176	7.86031389236
16	timeout	1650.86029291

Figure 0.8: Comparaison des temps entre la plne pure et avec une initialisation avec la programmation dynamique(mix)

Conclusion : La methode de résolution la plus rapide est la programmation dynamique. Mais elle n'est pas capable de résoudre certain types de grilles plus difficiles ou il y a un choix de couleur arbitraire à faire pour la résoudre. La PLNE permet de résoudre ces instances mais c'est une methode lente car nous utilisons des variables binaires donc gurobi fait un branch and boud pour trouver la solution optimale. Par conséquent c'est un problème np-complet. On peut néenmoins améliorer le temps de résolution en calculant une grille partielle à l'aide de la programation dynamique et en appliquant la PLNE sur cette grille.