Rapport MOGPL

Yue WANG et Loic URIEN

December 2017

Contents

1 Raisonnement par programmation dynamique

1.1 Première étape

(**Q1**) Si tous les $T(j, l)$ ont été calculés, alors, étant donné que $j \in \{0, ..., m-1\}$ et $l \in \{1, ..., k\}$, il suffit de vérifier que $T(m-1, k)$ est vrai. En effet, si $T(m-1, k)$ est vrai, alors on peut colorier la ligne avec la séquence entière par définition des $T(j, l)$.

(**Q2**) Voici les valeurs de vérités pour les cas présentés :

- 1 : Toujours vrai, on n'a aucun bloc à placer, donc tout nombre de cases sera valide, donc $T(j, 0)$ vrai $\forall j \in \{0, ..., m-1\}$.
- 2 :

  (a) Il n'y a pas assez de place pour placer le bloc l, donc $T(j, l)$ nécessairement faux dans ce cas.

  (b) Il y a tout juste assez de place pour placer le bloc l, donc $T(j, l)$ vrai si et seulement si $l = 1$ et faux sinon.

(**Q3**) On peut calculer la valeur de vérité de $T(j, l)$ dans ce cas en utilisant la relation de récurrence $T(j, l) = T(j - s_l - 1, l - 1)$. On vérifie en effet si en plaçant le bloc l et en laissant une case blanche on peut toujours placer les autres blocs.

(**Q4**)

Algorithm 1: T(j, l) basique

Data: $j \in \{0, ..., m-1\}$, $l \in \{1, ..., k\}$
Result: une valeur booléenne

```
1  if l = 0 then
2  |  retourner Vrai
3  else
4  |  if j < s_l − 1 then
5  |  |  retourner Faux
6  |  else if j = s_l − 1 then
7  |  |  if l = 1 then
8  |  |  |  retourner Vrai
9  |  |  else
10 |  |  |  retourner Faux
11 |  |  end
12 |  else
13 |  |  retourner T(j − s_l − 1, l − 1)
14 |  end
15 end
```

1.2 Généralisation

(**Q5**) La généralisation modifie tous les cas présentés dans le cas simple. Ainsi, nous avons maintenant :

- 1 : Dans ce cas, $T(j, 0)$ vrai si et seulement si la case (i, j) est blanche ou indeterminée et $T(j-1, 0)$ vrai (dans le cas où $j > 0$).
- 2 :

  (a) De même que dans le cas précédent, $T(j, l)$ toujours faux dans ce cas.

  (b) et (c) Plus complexes

Les deux derniers cas sont plus complexes que précédemment. Voici comment nous les traiterons pour toute case (i, j) et bloc l :

- Si la case (i, j) considérée est blanche, alors forcément le bloc l doit être placé avant. Ainsi $T(j, l) = T(j-1, l)$.
- Si la case est indeterminée, alors le bloc l peut éventuellement commencer avant, nous allons donc commencer par tester si $T(j-1, l)$ vrai.

  - Si oui, alors nous pouvons placer le bloc l même sans utiliser la case (i, j), son ajout ne change rien et donc $T(j, l)$ vrai.

  - Si non, alors on est obligé d'utiliser la case (i, j) pour placer notre bloc l, cette case sera donc forcément noire et on traitera ce cas comme si elle l'était (voir prochaine condition)
- Si la case est noire, alors le bloc l doit forcément finir dessus. On vérifie donc que pour $j \in \{j - s_l + 1, ..., j\}$, aucune n'est blanche, sinon $T(j, l)$ faux. Dans le cas contraire, alors :

  - Si la case juste avant notre séquence est noire, donc si $(i, j - s_l)$ noire, on ne peut pas placer notre séquence ici, donc on retourne faux.

  - Si $j - s_l - 1 >= 0$, donc si en laissant au moins un bloc d'écart entre le bloc actuel et le précédent on ne sort pas de la grille, on a que $T(j, l) = T(j - s_l - 1, j - 1)$ comme précédemment.

  - Si $j - s_l - 1 < 0$, mais que $l - 1 = 0$, alors on n'a plus aucun bloc à placer quoiqu'il en soit, donc $T(j, l)$ vrai.

  - Si $j - s_l - 1 < 0$, et que $l - 1 \neq 0$, alors on ne peut en aucun cas placer les blocs restant, donc $T(j, l)$ faux.

(**Q6**)

Algorithm 2: T(j, l) généralisé

Data: $j \in \{0, ..., m-1\}$, $l \in \{1, ..., k\}$
Result: une valeur booléenne

```
if l = 0 then
    if (i, j) case noire then
        retourner Faux
    else
        retourner T(j − 1, 0)
    end
else
    if j < s_l − 1 then
        retourner Faux
    end
    if j = s_l − 1 then
        retourner T(j − 1, l)
    else
        if (i, j) indéfinie et T(j − 1, l) Vrai then
            retourner Vrai
        else
            for k allant de j − s_l − 1 à j do
                if (i, k) blanche then
                    retourner Faux
                end
            end
        end
    end
    if j − s_l ≥ 0 then
        if i, j) noire then
            retourner Faux
        end
    end
    if j − s_l − 1 ≥ 0 then
        retourner T(j − s_l − 1, l − 1)
    else if l − 1 = 0 then
        retourner Vrai
    else
        retourner Faux
    end
end
```

(**Q8**) Ci-dessous le résultat obtenu sur l'instance 9. Pour une étude complète des performances, voir la partie 2.2, Q15 :

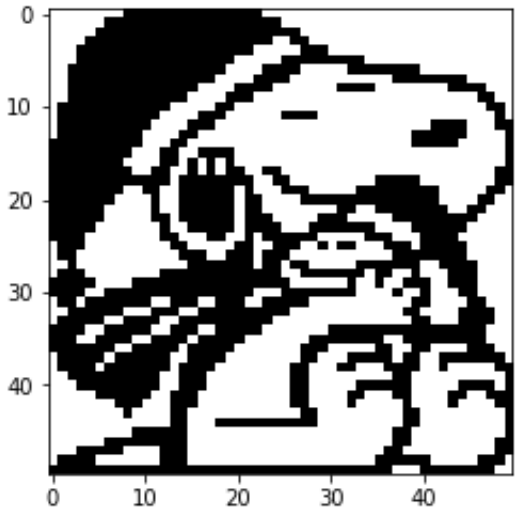

Figure 1: Instance 9 résolue par programmation dynamique

(**Q9**) Le programme n'arrive pas à résoudre l'instance 11. En effet, notre algorithme est basé sur le concept simple d'essayer de trouver dans une ligne ou une colonne, une case qui ne peut être coloriée que d'une seule et unique couleur, en considérant les colonnes indépendamment des lignes et les lignes indépendamment des colonnes. Dans le cas de l'instance 11, il est impossible d'inférer quoique ce soit en regardant les lignes et les colonnes séparément. Afin de palier à ce problème, nous pourrions éventuellement écrire un algorithme de backtracing qui, une fois qu'il n'y a plus aucune manière d'inférer simplement les couleurs comme on le faisait précédemment, testerait une à une les possibilités en faisant attention aux contraintes de validité et reviendrait en arrière lorsqu'aucune ne conviendrait.

2 Résolution par PLNE

2.1 Modélisation

(**Q10**) Pour toute colonne j et pour tout bloc t dans la ligne i, la contrainte s'exprime de la manière suivante :

$$\sum_{m=j}^{j+s_t+1} x_{im} \geq s_t y_{ij}^t$$

Nous voulons en effet que si t commence sur (i, j), alors forcément les cases

suivantes soient noires. Sinon, on les laisse libres.

(**Q11**) Nous voulons contraindre le bloc $t+1$ à commencer au minimum à la case $(i, j+s_t+1)$, nous avons donc la contrainte suivante :

$$\sum_{m=j+s_t+1}^{M-1} y_{im}^{(t+1)} \geq y_{ij}^{t}$$

(**Q12**) Afin que l'algorithme produise une solution valide, nous devons ajouter deux classes de conditions supplémentaires. Premièrement, un bloc ne peut pas commencer sur plusieurs cases simultanément, nous devons donc ajouter la condition suivante :

$$\sum_{m=0}^{M-1} y_{im}^{t} = 1 \ \forall t \in \{1, ..., k\}$$

De plus, pour le moment aucune condition n'oblige à placer tous les blocs sur une ligne. En effet, la deuxième classe de condition décrite dans la Q11 ne suffit pas. Par exemple, si on avait 3 blocs à placer, placer le bloc 2 oblige à placer le bloc 3, mais on peut omettre le bloc 1 car la condition est toujours valide dans ce cas. Ainsi, nous devons ajouter cette condition :

$$\sum_{m=0}^{M-1} x_{im} = \sum_{n=0}^{k} s_n$$

Toutes ces conditions se généralisent trivialement aux colonnes.

2.2 Implantation des tests

(**Q13**) Afin de trouver une borne inférieure pour j avant laquelle le bloc t ne peut être placé, il suffit de considérer que l'on doit places les blocs précédents tous séparés d'une case blanche au minimum. On déduit donc que pour y_{ij}^{t}, $\forall t \in \{1, ..., k\}$ et i fixé, on a :

$$j \geq t + \sum_{l=1}^{t-1} s_l$$

Similairement, on trouve une borne supérieure pour j :

$$j \leq (k - t - 1) + \sum_{l=t+1}^{k} s_l$$

(**Q15**) Avant de présenter les résultats de cette section, il est important de signaler les problèmes rencontrés durant l'implémentation du solveur par PLNE. En effet, après avoir écrit les contraintes et supprimé les variables demandées, le programme était capable de résoudre toutes les instances de la 11 à la 15 (bien que la 15 soit résolue en plus de 5 minutes), mais était bien trop lent sur

l'instance 16. Nous avons donc implémenté l'optimisation suggérée en faisant tourner d'abord l'algorithme par programmation dynamique sur l'instance, puis en fixant les variables déterminées, mais le temps d'exécution était toujours bien trop lent. Dans nos recherches, nous nous sommes rendu compte que l'ordre des contraintes pouvaient influencer le temps de résolution de manière non negligeable [1] [2].

Afin de modifier l'ordre des contraintes sans changer le code, ce qui aurait ajouté une charge de travail non neligeable et nuit fortement à la lisibilité, nous avons choisi de modifier la seed du générateur aléatoire de gurobi. Pour ce faire, nous avons supposé que les autres instances étaient d'une forme similaire à l'instance 15 et nous avons choisi des seeds aléatoirement avec un timeout de 2 minutes afin de garder la meilleure. Tous les tests ont été réalisés sur un Intel(R) Core(TM) i3-5010U CPU @ 2.10GHz, tournant sous ArchLinux. Voici les résultats obtenus, en secondes, pour les 10 premières instances, avec le pré-traitement désactivé pour le PLNE :

Instances	0	1	2	3	4	5
Dynamique	0.001	0.006	0.46	0.44	0.93	0.64
PLNE	0.23	0.58	78.8	0.10	66.55	21.53

Instances	6	7	8	9	10
Dynamique	1.57	0.98	1.4	19.34	18.52
PLNE	115.82	74.61	12.0	trop long	28.44

Comme nous pouvons le voir, le temps de résolution pour l'algorithme par programmation dynamique croît en fonction de la taille de l'instance, ce qui n'est pas le cas pour le PLNE qui affiche des temps bien plus aléatoires. Voici les résultats pour les instances de 11 à 16 :

Instances	11	12	13	14	15	16
PLNE (sans pré-traitement)	0.12	117.67	0.58	0.44	72.68	trop long
PLNE (avec pré-traitement)	0.12	0.43	0.05	0.04	30.14	428.22

Comme nous pouvons le voir, le pré-traitement a un effet positif non négligeable sur les performances de la méthode dans tous les cas.
Les grilles obtenues pour les deux algorithmes pour l'instance 15 sont donnés ci dessous, les cases grises étant les cases laissées indeterminées à la fin de l'exécution :

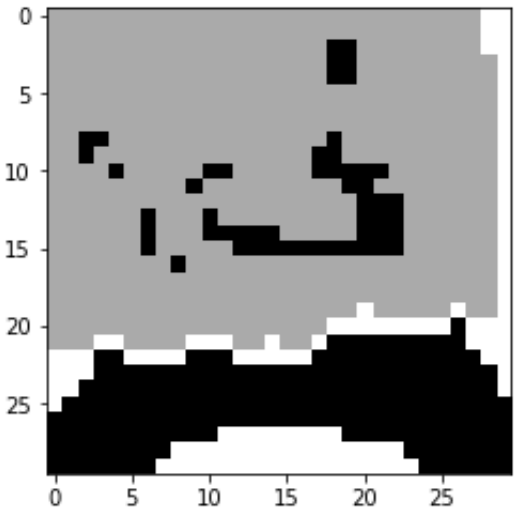

Figure 2: Instance 15 résolue partiellement par programmation dynamique

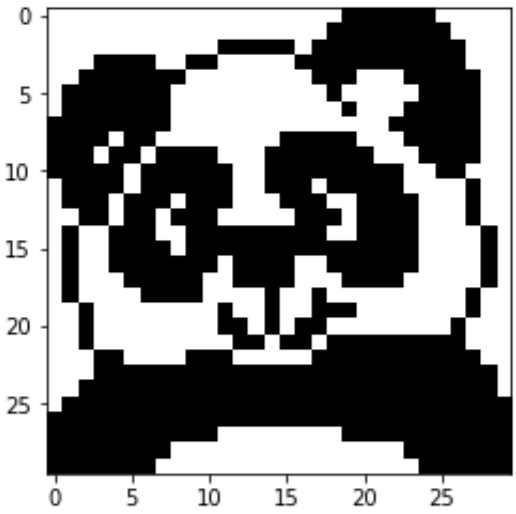

Figure 3: Instance 15 résolue par PLNE

References

[1] Emilie Danna. Performance variability in mixed integer programming.

[2] Juan Esteban. Changing the order of the constraints for improving mip solving time.