Complexité & Récursivité

# Compte rendu

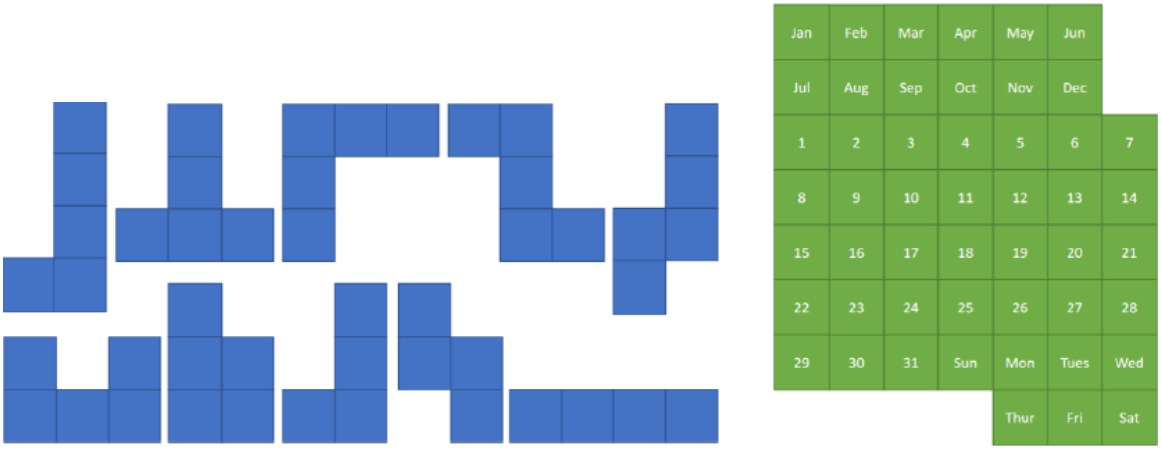
TP Calendrier de Gérard Dray - 2023

FORAY Leo Paul, MU Maxime (IMT Mines Alès)
26/03/2023

# Table des matières

# I. Introduction

L'objectif de ce travail est d'examiner les différentes solutions du problème suivant :




On cherche les différentes possibilités de remplissage du plateau à droite avec les pièces de gauche.

On sait que lorsque toutes les pièces sont posées, seules trois cases vertes restent non couvertes, et l'on souhaite que ces trois cases forment une date (Mois – Jour – Jour de la semaine).

Dès-lors, nous aimerions trouver toutes les solutions possibles à ce problème, autrement dis toutes les solutions possibles pour chaque triplet/date, aussi appelé instance par la suite.

## 1 – Les variables

Pour une instance donnée, les variables qui régissent le problème sont les pièces bleues qui possèdent des propriétés de translation et de rotation. En effet chacune peut être positionné à divers endroit, et peut subir des rotations de 90°, 180° et 270° (exception sera faite pour les pièces symétriques).

Pour la modélisation du problème, on considère le placement du point haut gauche de la pièce.

Ainsi, dans certains cas, comme celui-ci, il n'y a pas de carré bleu dans ce coin spécifique. Dans cette modélisation, chaque pièce **a** appartenant à l'ensemble [1,10] est représentée par le triplet ($x_a$, $y_a$, $r_a$) qui traduit sa position sur le plateau selon x et y ainsi que sa rotation.

## 2 – Domaine de valeurs

Le domaine de valeurs pour les variables est défini par les coordonnées ($x_a$, $y_a$) représentant la position de chaque pièce dans un plateau de dimension 8 x 7, ainsi que la variable $r_a$, représentant la rotation de chaque pièce avec 4 valeurs possibles. Par conséquent, le domaine de valeurs pour chaque variable est le suivant :

- $x_a \in \{1, 2, 3, 4, 5, 6, 7, 8\}$
- $y_a \in \{1, 2, 3, 4, 5, 6, 7\}$
- $r_a \in \{1, 2, 3, 4\}$

## 3 – Les contraintes

Le plateau de taille 8x7 comporte de base 6 cases bloquées auxquelles, pour chaque instance, viendront s'ajouter trois cases définissant la date auquel on s'intéresse, soit 9 cases inaccessibles en tout.

Ainsi, la somme des carrés unitaires de la surface verte doit être égale à celle des carrés qui constituent les pièces bleues plus 3 unités.

Ceci traduit aussi le fait que chaque pièce qui est placée sur le plateau limite les options de placement disponibles pour les pièces suivantes, car celles-ci ne peuvent pas se chevaucher.
De plus, il est important de préciser que pour certaines pièces symétriques, il n'y aura que deux rotations possibles au lieu de quatre.

# II. Algorithme

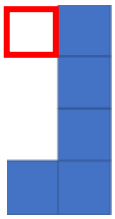
Pour résoudre ce problème, nous avons choisis d'utiliser le langage C qui permet une compilation et une exécution bien plus rapide que d'autres langages plus haut niveau, tels que Python.

Dans un premier temps, nous avons implémenté un programme proposant une approche récursive basée sur du backtracking pour résoudre le problème de Gérard Dray.

Il définit une grille de jeu de dimensions « **ROWS**x**COLUMNS** (8x7) » appelée « **board** » et représentée sous la forme d'un tableau à deux dimensions d'entiers, où les cases vides sont représentées par des 0, les obstacles sont représentés par les chiffres 11 et les cases occupées sont représentées par des entiers positifs correspondant au numéro de la pièce. Cette grille représente le plateau sur lequel les pièces doivent être placées.

Les constantes **ROWS**, **COLUMNS**, **PIECES** et **ROTATIONS** définissent respectivement le nombre de lignes et de colonnes du plateau de jeu, le nombre de pièces différentes disponibles pour le jeu, et le nombre maximum de rotations possible pour chaque pièce.

Comme spécifié plus haut, nos pièces sont référencées à partir de l'angle haut gauche, et sont composée des coordonnées de chacun des carrés composant la pièce par rapport à cet angle haut gauche.



Par exemple, cette pièce est représentée par {{0, 1}, {1, 1}, {2, 1}, {3, 0}, {3,1}}

Méthodes utilisées :

- « **placePiece** » est une méthode qui vérifie si la pièce peut être placée à partir des coordonnées (i,j) sur le plateau « **board** » en respectant les contraintes à savoir les bordures du plateau et la présence des pièces déjà placées. Si elle trouve qu'une pièce ne peut être placée, alors elle va renvoyer un booléen « **False** ». Sinon elle place la pièce et renvoie « **True** ».

- « **removePiece** » est une méthode qui consiste à retirer une pièce du puzzle sur le plateau « **board** ». Elle est utilisée pour corriger une erreur ou pour essayer différentes combinaisons de pièces pour trouver la solution optimale.

- « **isEncircled** » est une méthode booléenne qui consiste à vérifier si le plateau « **board** » contient un 0 isolé c'est-à-dire un espace libre qui n'a pas été occupé par une pièce. Cette méthode permet d'optimiser grandement le temps de calculs et d'exécution de notre programme.

- « **solvePuzzle** » est une méthode récursive qui résout automatiquement un puzzle sur le plateau « **board** ». Elle fait notamment appel aux méthodes « **placePiece** » pour placer les pièces, « **removePiece** » pour faire le backtracking, « **isEncircled** » pour vérifier si le plateau ne contient pas de 0 isolé.
  La fonction commence par déterminer la taille et le nombre de rotations possibles pour chaque pièce en fonction de son numéro. Ensuite, elle parcourt chaque rotation possible de chaque pièce, puis parcourt chaque case du plateau de jeu pour tenter de placer la pièce dans cette position. Si la pièce peut être placée sans chevaucher une autre pièce ou sortir du plateau, la fonction enregistre ce placement et passe à la pièce suivante. Si la pièce en cours de placement est la dernière pièce à placer, la fonction incrémente le nombre total de solutions trouvées. Si ce n'est pas la dernière pièce, la fonction appelle récursivement « **solvePuzzle** » avec la pièce suivante. Enfin, la fonction retire la pièce du plateau de jeu pour essayer une autre position (backtracking).

- « **chooseDate** » est une méthode qui permet de choisir les dates sous format (**MOIS, QUANTIEME, JOUR),** elle permet de placer les contraintes sur le plateau « **board** ».

Nous avons dans un second temps essayé de résoudre le problème par une autre approche que celle du backtracking. Cette nouvelle méthode consiste en ceci :

- On parcourt le tableau de ligne en ligne, avec toujours le coin haut gauche comme origine.
- Pour chaque case parcourue :
    - On essaie de placer toutes les pièces ainsi que leurs rotations.
        - Lorsqu'une pièce peut être placé sur cette case :
        - On place la pièce sur le plateau.
        - On appel par récursivité la méthode de résolution, avec en paramètre le nouveau plateau ainsi formé, la liste des pièces restantes à placer, et le nombre de pièces restantes à placer.
        - Puis l'on enlève cette pièce du plateau, pour que la boucle puisse tester d'autres possibilités.
        - Lorsque toutes les pièces ont été testées, si la case est vide (i.e. si elle n'a pas été initialement verrouillée) on renvoie le nombre de solutions trouvées.
        - Sinon la boucle continue son parcours.

Cependant, cette méthode qui ne devrait renvoyer que des résultats uniques, renvoie des résultats en double. Si l'on prend l'exemple du samedi 1 Janvier, pour lequel 54 solutions différentes existent, cette méthode en retourne 63, mais en seulement 1.23 secondes.

Nous avons évidemment des doublons parmi les solutions, et nous avons pu même en identifier 7 paires qui étaient trouvé à chaque fois l'une après l'autre. Dès lors, nous avons essayé de créer une sauvegarde des solutions trouvées pour ne les ajouter qu'une seule fois, cependant après de multiples tentatives cette méthode ne fonctionne pas, et nous ne parvenons donc pas à un résultat pertinent avec cet algorithme.

## III. Optimisations

Le premier programme que nous avons décrit ci-dessus utilisait l'approche de la force brute pour résoudre le problème de placement de pièces sur le plateau. En effet, cette méthode consiste à tester toutes les combinaisons possibles de rotations et de positions pour chaque pièce, puis vérifie si la combinaison est valide ou non. Ce processus est répété jusqu'à ce qu'une solution valide soit trouvée. Mais dans notre algorithme nous avons adaptée cette méthode de façon à cette approche nous donne toutes les solutions possibles.

Cependant, cette méthode peut être très chronophage voir peu efficace lorsque l'on a affaire à des problèmes de grandes dimensions.

Mais c'est aussi le cas pour notre problème, qui laisse place à nombre de simplifications. Par exemple, si une pièce a été mal placée au départ (par exemple en laissant un 0 isolé au départ, qui ne pourra jamais être comblé car les pièces sont trop grandes), il peut s'écouler beaucoup de temps avant que l'algorithme ne change sa position et qu'il comble le 0.

C'est pour cela que nous avons créé la méthode « **isEncircled** » qui permet de détecter la présence d'un 0 isolé sur le plateau « **board** ». Si l'on parvient à détecter plus tôt les emplacements invalides sur le plateau de jeu, il serait alors possible d'informer de manière directe l'algorithme pour qu'il change immédiatement la pièce en question. Cette méthode permet donc de gagner un nombre considérable de temps de calculs et d'exécution.

Par exemple :

| Mois | Quantième | Jour | #Solutions | Secondes CPU |
|---|---|---|---|---|
| 1 | 1 | 6 | 54 | **130.750** |

Sans « **isEncircled** ».

| Mois | Quantième | Jour | #Solutions | Secondes CPU |
|---|---|---|---|---|
| 1 | 1 | 6 | 54 | **7.078** |

Avec « **isEncircled** ».

Ainsi, on peut constater l'importance de cette méthode grâce à l'économie de calculs qu'elle fournit.

L'avantage de la seconde solution que nous proposions est que la vérification des cases vides isolées est directement incluse dans la fonction récursive : nous y avons ajouté deux paramètre $i_{min}$ et $j_{min}$, qui sont les coordonnées de la case ayant réalisé l'appel.

Ainsi on peut vérifier dès le début si la case est vide, et donc renvoyer 0 dans ce cas-là. On se rapproche des 54 solutions réelles avec maintenant 57 solutions trouvées, et ce en 1.01 secondes. On parvient maintenant à trouver tous les doublons dans l'affichage, mais nous ne sommes toujours pas parvenus à résoudre ce problème.

Dans l'optique où nous trouvions quelle partie de la boucle créer ces solutions en double, nous pourrions y inclure un nouveau cas fermant les branches redondantes, ce qui accélèrerait encore notre programme.

## IV. Résultats et interprétations

On obtient ces résultats sur notre algorithme le plus optimisé :

| Mois | Quantième | Jour | Solutions #1 | Secondes CPU #1 |
|---|---|---|---|---|
| 1 | 1 | 6 | 54 | 7.078 |
| 3 | 30 | 4 | 36 | 5.592 |
| 10 | 15 | 6 | 21 | 3.995 |
| 5 | 5 | 1 | 13 | 1.863 |
| 10 | 20 | 5 | 21 | 1.562 |
| 1 | 14 | 5 | 45 | 2.812 |
| 1 | 14 | 6 | 60 | 8.240 |
| 7 | 31 | 7 | 47 | 6.741 |
| 9 | 2 | 4 | 15 | 2.433 |
| 11 | 14 | 5 | 47 | 1.915 |
| 12 | 24 | 4 | 23 | 4.211 |
| 8 | 29 | 1 | 47 | 3.435 |
| 2 | 30 | 2 | 58 | 4.241 |
| 1 | 7 | 2 | 108 | 9.736 |
| 1 | 13 | 6 | 49 | 8.382 |
| 1 | 15 | 2 | 102 | 7.678 |
| 1 | 14 | 4 | 54 | 8.405 |
| 1 | 29 | 4 | 104 | 12.852 |
| 1 | 29 | 6 | 97 | 12.595 |
| 7 | 12 | 1 | 13 | 1.452 |
| 8 | 6 | 1 | 5 | 1.054 |
| 4 | 29 | 2 | 142 | 8.282 |
| 4 | 27 | 1 | 5 | 1.231 |
| 3 | 27 | 1 | 1 | 0.971 |
| 3 | 27 | 3 | **0** | 1.604 |
| 3 | 1 | 1 | **0** | 1.811 |
| 2 | 27 | 3 | **0** | 1.375 |
| 1 | 27 | 3 | **0** | 2.526 |
| 2 | 27 | 3 | **0** | 1.375 |
| 3 | 6 | 1 | **0** | 1.004 |
| 3 | 27 | 3 | **0** | 1.540 |
| 12 | 27 | 3 | **0** | 1.463 |
| 11 | 29 | 2 | **197** | 8.381 |
| 6 | 7 | 2 | 176 | 10.618 |
| 1 | 6 | 6 | 12 | 4.831 |
| 1 | 28 | 3 | 95 | 10.913 |
| 8 | 27 | 1 | 9 | **0.936** |
| 9 | 11 | 1 | 12 | 1.444 |

Nous avons testé notre algorithme sur les 2,604 possibilités et 19 dates ne possèdent pas de solutions, le mardi 29 novembre est l'instance qui possède le plus de solutions, avec pas moins de 197 possibilités.

# V. Conclusion

De notre travail nous pouvons conclure que seules 19 instances ne possèdent pas de de solutions, ce qui répond à notre problématique de départ. Cependant, il est intéressant de noter que l'optimisation de nos programmes a été la partie la plus complexe du travail, car il était impensable de résoudre le problème si chacune des instances nécessitaient 130 secondes. Le travail d'optimisation a ici été très important, et bien que nous ne sommes pas parvenus à compléter cette deuxième méthode pour comparer les performances, le temps d'exécution pour chaque instance reste inférieur à 10 secondes.