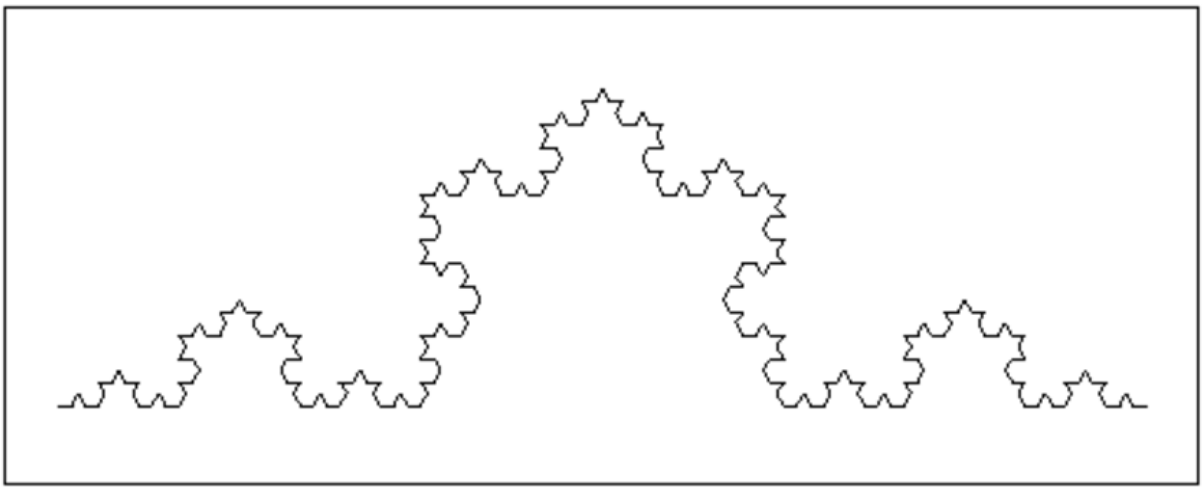
IMT Mines Alès
École Mines-Télécom

Rapport de projet
Complexité et Récursivité



« Puzzle-a-day »

Groupe :
BELKHITER Yannis
FAURE Benoît
STORAÏ Romain
TEXIER Lucas

Professeur :
M. Vasquez

Table des matières

I- Présentation du sujet

Le sujet proposé est un problème proposé par Gérard Dray (Professeur-chercheur à IMT Mines Alès). Il est inspiré d'un sujet célèbre : "Puzzle-a-day".
Le problème consiste à disposer 10 pièces différentes sur un plateau représentant un calendrier :

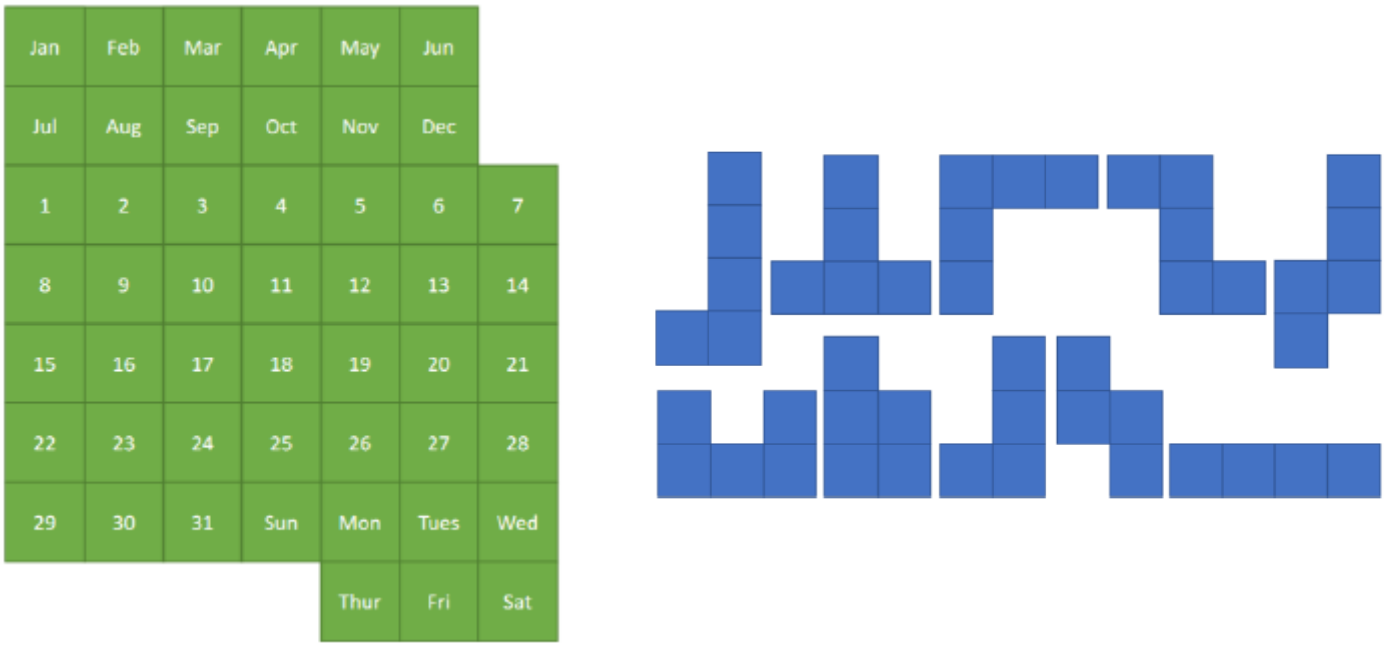


Figure 1 : Calendrier (contenant mois, quantième et jour) & les 10 pièces à placer

L'objectif du problème est de fixer 3 cases en sélectionnant un jour de la semaine, un jour du mois et un mois. Ensuite, il faut agencer les 10 pièces bleues sur le plateau de jeu (sans qu'elles se superposent), tout en laissant les trois cases sélectionnées non-recouvertes correspondant à la date du jour dans l'année. La somme des carrés unitaires de la surface verte est égale à celle des carrés qui constituent les pièces bleues, plus trois unités. Cette contrainte de non-recouvrement des pièces sur la surface verte est implicite dans le problème. Ce rapport aborde la résolution du problème proposé dans un cadre scolaire.

II- Problématique et contraintes du problème

La problématique du sujet proposé est :

"*Peut-on compléter le puzzle tous les jours de l'année* ?"

Pour répondre à la problématique dans le cadre du cours Complexité & Récursivité, la démarche adoptée est de réaliser un algorithme trouvant le nombre de solutions possibles par jour et d'en étudier sa complexité.
La réalisation de ce projet a été fait en plusieurs étapes :

1. Modélisation des paramètres du problème
2. Implémentation d'un premier algorithme fonctionnel
3. Recherche d'optimisation de la complexité par tous les moyens possibles (quitte à revenir sur notre modélisation initiale du problème)
4. Obtention des résultats
5. Analyse des résultats

Les contraintes du sujet sont les suivantes :
-Toutes les pièces doivent être placées.
-Les pièces ne se chevauchent pas et sont entièrement incluses sur la grille.
-Les pièces peuvent réaliser des rotations. (4 positions possibles par pièce)
De plus, nous sommes partis du principe que les pièces ne peuvent pas être tournées dans l'autre sens (sens miroir). Ce choix a été motivé par le fait de trouver avec cette hypothèse le même nombre de solutions que celles présentes dans le sujet pour la date du samedi 1er janvier.

mois	*quantième*	*jour*	*#solutions*	secondes CPU
1	1	7	54	17.5
...	...	...	...	...

Cet exemple correspond au samedi 1^{er} janvier

Figure 2 : Extrait du sujet du TP

III-Notre démarche

Le sujet nous invite à réaliser un algorithme récursif, nous l'avons implémenté en C.

La première modélisation

-Les jours, les quantièmes et les mois sont représentés dans cette première modélisation par des entiers. Respectivement de [0-6], de [0-30] et [0-11]. Par exemple, le lundi 9 septembre est représenté par (0, 8, 8).

- Les pièces du puzzle sont modélisées sous la forme d'une matrice de 0 et de « n » pour représenter la forme de la pièce « n ».

-C'est une liste chaînée qui stocke les pièces. Pour chaque pièce on représente toutes les rotations uniques possibles.

- La grille est modélisée sous la forme d'un tableau unidimensionnel d'entiers de taille 56 (8*7). Chaque entier représente une case de la grille et contient un nombre entier compris entre -1 et 10.

*le nombre -1 correspond à un endroit vide car la grille n'est pas carrée ou alors à la case du jour/quantième/mois.

*le nombre 0 représente une case vide.

*les nombres de 1 à 10 représentent que la case est remplie par la pièce associée au numéro.

Remarque : Nous avons stocké pour toutes les pièces leurs 4 variantes selon les rotations. Cependant pour améliorer le temps d'exécution et avoir le nombre exact de solutions, nous n'avons pas considéré les rotations des pièces symétriques qui ramènent à la forme initiale.

Fonctionnement de l'algorithme « old_solver »

L'algorithme suit une approche basée sur la recherche exhaustive. Il commence par charger toutes les formes possibles dans un ordre fixé et pour chaque forme il mémorise toutes les rotations uniques possibles.

Ensuite, l'algorithme essaie de placer les pièces sur le puzzle. Si une forme peut être placée, l'algorithme continue à placer la forme suivante.

Deux cas de figure :
-toutes les pièces ont pu être placées : la grille est donc correctement remplie, la solution est enregistrée et on reprend l'algorithme en partant d'une autre pièce.
-toutes les pièces n'ont pas pu être placées : la grille n'est alors pas correctement remplie. L'algorithme revient en arrière et essaie une autre combinaison de placements de formes.

A la fin, l'algorithme retourne les solutions trouvées.

Les optimisations de ce premier algorithme

Voici une liste non-exhaustive des optimisations présentent :

1. Utilisation de pointeurs : les structures de données sont définies sous forme de pointeurs. Cela permet de ne pas avoir à copier les données et donc de réduire les temps d'allocation et de libération de la mémoire.

2. Lecture des fichiers de formes une seule fois : lors du chargement des fichiers de formes, toutes les formes sont chargées une seule fois et stockées dans une liste de formes. Cela permet d'éviter de relire les fichiers de formes à chaque fois qu'une nouvelle forme est ajoutée.

3. Utilisation minimale de la fonction malloc : favorisation de l'allocation dynamique de mémoire.

La deuxième modélisation

Dans la deuxième version de notre algorithme, pour améliorer les performances, nous avons considéré la grille comme 8 lignes. Chaque ligne est représentée par un entier codé sur 8 bits. Nous passons donc d'un tableau de 56 entiers codés sur 8 bits (1ère modélisation) à 8 entiers codés sur 8 bits. En faisant cela on optimise encore la complexité de notre code.

Fonctionnement de l'algorithme « solver_solutions »

Dans le premier algorithme, on regardait la valeur de la case pour savoir si elle était libre (càd égale à 0) et on changeait la valeur de l'entier pour compléter la case. Ici, on utilise les opérations logiques "et" et "ou" pour changer l'état d'une case.

Par exemple :

1) Si l'on souhaite compléter la première case de la grille (en haut à gauche)

On réalise l'opération : (10000000 ou 00000000) sur la première ligne.

2) Si l'on souhaite regarder si on peut remplir une nouvelle fois la grille obtenue de la même manière

On réalise l'opération : (100000000 et 10000000) sur la première ligne. Comme la réponse contient un 1 : la case n'est pas libre.
On construit ainsi les pièces en "ajustant le nombre de zéro".

Les résultats

La réalisation d'un script a permis de collecter le nombre de solutions et le temps d'exécution pour l'ensemble des combinaisons (quantile, jour, mois) possibles (même le 31 février…).
À noter que les calculs ont pris 10h sur un CPU de performance moyenne. Tous les résultats figurent dans le tableur 'solutions_per_day'.

IV- Analyse des résultats

Une fois les données collectées, la première étude concerne la répartition du nombre de solutions pour l'ensemble des combinaisons (quantile, jours, mois).

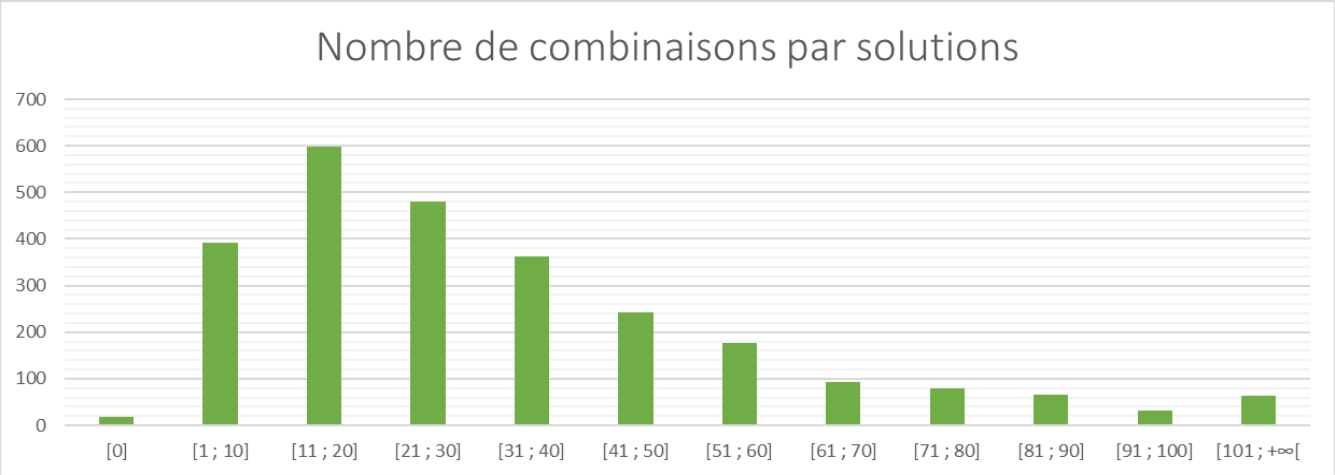


Figure 3 : Répartition du nombre de solutions pour l'ensemble des combinaisons

On constate que seulement 19 combinaisons sur les 2604 (31×12×7) n'admettent aucune solution. En moyenne, on trouve 32.9 solutions et la valeur maximale est atteinte le mardi 19 novembre avec 197 solutions.

Concernant le temps d'exécution, il varie de 1.6 à 22.698 secondes avec un temps moyen de 7.1 secondes.
On ne note pas de corrélation significative entre le nombre de solutions et le temps d'exécution. On expliquerait les temps les plus faibles par le fait que l'algorithme détermine très tôt qu'une certaine disposition de pièces n'est pas envisageable ce qui réduit le nombre d'itération.

Nombre total de solutions	85637
Nombre de combinaisons	2604
Nombre moyen de solutions	32.9
Nombre médian de solutions	26
Nombre minimal de solutions	0
Nombre maximal de solutions	197

Temps moyen d'execution	7.1 s
Temps médian d'execution	6.7 s
Temps minimum d'execution	1.7 s
Temps maximum d'execution	22.7 s

Figure 4 : Analyse statistique des résultats obtenus

Pour aller plus loin dans l'analyse compte tenu de la problématique, l'étude s'est ensuite portée sur les 19 combinaisons n'admettant pas de solutions.

month	day	week_day	solutions
0	26	3	0
1	26	3	0
2	0	1	0
2	5	1	0
2	26	3	0
3	5	0	0
3	5	3	0
3	26	3	0
4	26	3	0
5	26	3	0
6	26	3	0
7	26	3	0
8	26	3	0
9	4	1	0
9	4	6	0
9	5	4	0
9	26	3	0
10	26	3	0
11	26	3	0

Figure 5 : Liste des combinaisons comportant 0 solutions

On remarque que parmi ces 19 combinaisons, il figure 12 fois le mercredi 26.
En d'autres termes, avec ces pièces il est impossible de réaliser le puzzle les mercredis 26.

V- Conclusion et pistes d'améliorations

Réponse à la problématique

Pour rappel la problématique était :

"*Peut-on compléter le puzzle tous les jours de l'année ?*".

Pour construire cette réponse, l'objet principal du projet est l'optimisation presque maximal du code de manière à faire fonctionner le script dans un temps restreint avec nos moyens (CPU de qualité moyenne).

On peut, à l'aide de notre étude, réfuter la problématique : le puzzle ne peut être compléter pour exactement 19 combinaisons.

Cependant, nous avons dans l'esprit du TP interdit aux pièces de pivoter de sens.
En rajoutant dans l'implémentation de nouvelles possibilités de rotations aux pièces, et en reprenant le même algorithme, appliqué aux 19 combinaisons précédentes on parvient pour chacune a trouvé des solutions. En levant la contrainte concernant les rotations miroirs, la réponse à la problématique est oui, il y a au moins une solution pour les 2604 combinaisons !

Rappel des livrables et guide d'utilisation

À faire

Pistes d'améliorations

Pour aller encore plus loin et réduire encore le temps de calcul, on pourrait :

- S'intéresser aux solutions et par exemple se dire que certains agencements n'aboutissent jamais à aucune solution : par exemple, il est impossible de commencer avec certaines pièces dans certaines positions, ou d'enchaîner certaines pièces à la suite.
- Trouver l'ordre optimal pour le placement des pièces : le but serait d'arranger l'ordre pour qu'en moyenne, le parcours des branches du graphe s'arrête le plus tôt possible pour limiter les itérations.