POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Devoir 1 - Optimisation d'une galerie d'art

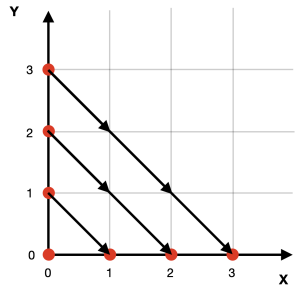
Djian, Benjamin 2226903 - Saikali, David 2015144
Département de Génie Informatique Génie Logiciel

Trimestre Hiver 2024

NB : Ce document reprend les notations définit dans l'énoncé du devoir.

1 Construction de la solution initiale

Pour construire notre solution initiale, on essaye de placer toutes les pièces d'art sur le premier mur. S'il nous reste des pièces non placées, on créé un nouveau mur et on essaye de placer ce qui reste. Ce processus est répété jusqu'à ce qu'il n'y ait plus de pièces non placées.



Le parcours d'un mur se fait en suivant la droite $x + y = k$ où $k \in \mathbf{N}$ en augmentant progressivement la valeur de k tout en gardant la somme de la hauteur et de la largeur du mur comme borne supérieure à la valeur k. L'idée est de toujours placer les pièces dans l'espace le plus en bas à gauche possible. On essaie ainsi de réduire au maximum les espaces vides entre les pièces. On n'a pas de preuve que cet ordre de placement est optimal, c'est un raisonnement intuitif. Par contre, pour une solution quelconque, déplacer une pièce vers la droite ou vers le haut devrait mener vers une solution équivalent ou pire. Le seul cas où il pourrait y avoir une amélioration est si ce déplacement permet de placer une pièce dans le trou créé et, pour atteindre cette solution, il ne faut que changer l'ordre de placement des pièces. La qualité de notre solution dépend donc de l'ordre dans lequel on place les pièces d'art. Les résultats détaillés dans la section 3 semblent confirmer cette intuition. Nous supposons donc qu'une solution optimale peut être construite ainsi et le reste du travail continuera avec cette supposition.

Lors de la construction de solution, on doit vérifier si une pièce d'art peut rentrer sur un mur, pour le faire, il faut itérer sur toutes les tuiles possibles du mur. En choisissant la valeur minimale et maximale de x intelligemment, il est possible de faire ce passage en:

$$\mathcal{O}\left((W - ArtWidth) \cdot (H - ArtHeight)\right)$$

Par contre, pour une pièce d'art quelconque, en pire cas, on peut simplifier pour ne garder que $\mathcal{O}(W \cdot H)$. On aurait cette complexité en plaçant une pièce de 1x1 en haut à droite.

Il faut ensuite vérifier pour chaque tuile dans notre mur si la pièce rentre. Grâce à une structure de données qui garde en mémoire la taille maximale que cette tuile peut héberger si on y mettait le coin inférieur gauche d'une pièce d'art, cette vérification se fait en temps constant. Par contre, à chaque fois que l'on place une pièce, il faut ajuster cette matrice ce qui prend, en pire cas, $\mathcal{O}(W \cdot H)$. Par contre, puisqu'on place les pièces le plus en bas à gauche possible et que la forme des pièces varie, ce coût sera plus bas en moyenne.

Finalement, il faut répéter ce processus pour les pièces non placées tant qu'on n'a pas réussi à placer toutes les pièces d'art. La complexité de placer une pièce dépend donc de l'index du mur sur lequel elle est placée. La complexité de placer une pièce d'art dépend donc de la surface du mur puisque la recherche et la mise à jour de la matrice sont tous deux bornées par cette taille. Pour placer les k pièces on obtient la somme suivante:

$$\mathcal{O}\left(W \cdot H \cdot \sum_{i=1}^{k} WallIndex_i\right)$$

Puisque l'index des murs va contenir tous les chiffres de 1 au maximum au moins une fois, on sait que la complexité de la somme est au moins dans l'ordre de $nWalls^2$, mais ceci ne nous permet pas de poser une borne supérieure. On peut poser que $\mathcal{O}(nWalls^k)$ est une borne supérieure à la somme (en posant que chaque terme de la somme est égale à la valeur maximale).

Ainsi, avec cette implémentation, une solution pour une instance donnée correspond à un ordre de placement des pièces. Cette équivalence solution/ordre nous est très utile pour effectuer une recherche locale efficace.

2 Recherche locale

La recherche locale commence par la construction de la solution initiale qui repose sur un ordre d'insertion des oeuvres d'art sur les murs. L'ordre initial privilégie les pièces dont au moins une des deux dimensions est proche de la dimension du mur. L'idée est qu'il est toujours bénéfique de placer en premier une pièce

dont la largeur (ou la hauteur) est celle du mur puisqu'on évite ainsi de séparer le mur en deux parties distinctes.

La fonction de voisinage propose donc des ordres "voisins" à l'ordre de la solution actuelle. Le premier voisinage que nous avons considéré est un 2-swap, comme vu en cours. Dans un second temps, après des tests de cette implémentation, on a jugé bon de réduire la taille du voisinage. On a eu donc recours à un "last-swap". Un "last-swap" échange le dernier élément de la séquence avec un autre élément de la séquence, distinct du premier. L'autre motivation pour ce changement est que les pièces d'art placées sur le dernier mur sont celles que l'on souhaite bouger sur un autre mur. Ce voisinage est de taille $k-1$ et est connecté comme le montre l'algorithme suivant :

```
// solution, an array containing our art_ids in a non_optimal order
// bestSolution, an array containing our art_ids in optimal order
// swapLast, a function which swaps the last element with the given index
// find, a function which returns the index of the given art_id in solution
for (int i = 0; i < n; i++) {
    // Already the right order
    if (solution[i] == bestSolution[i]) continue;

    // The last element is not the one we want to swap with
    if (solution[-1] != bestSolution[i]) {
        swapLast(find(bestSolution[i]));
    }

    // We swap the current element with the last one which is the optimal choice
    swapLast(i);
}
```

Puisque l'on construit les solutions à partir des ordres, il est clair que tous les voisins générés sont valides. On garde le premier voisin qui améliore notre solution en itérant de façon aléatoire sur les voisins.

Pour évaluer la qualité d'un voisin, on a recours à l'heuristique suivante: on fait la somme de toutes les tuiles vides sur les murs excluant le dernier. Ainsi, on cherche à diminuer l'espace non occupé par une oeuvre d'art sur tous les murs, sauf le dernier. Si on incluait le dernier mur, cette heuristique ne donnerait pas assez d'information. En effet, pour une instance donnée, la surface qu'occupe les oeuvres d'art est constante. À nombre de mur égal, l'heuristique ne serait pas assez précise, car on aurait toujours le même score. En excluant le dernier mur, on pousse la recherche vers des solutions où les premiers murs sont le plus remplis possible. Il y a trois façons de réduire le nombre de tuiles vides: en réduisant le nombre de murs, en enlevant une pièce du dernier mur (sans en ajouter une plus grande au dernier mur) ou en échangeant une grosse pièce du dernier mur avec une petite pièce des autres murs. Cette heuristique donnera presque toujours un meilleur score à une solution ayant moins de murs puisqu'on ignore le dernier mur. Une solution qui réussit à réduire le nombre de mur aura un score qui ne contient pas les tuiles vides de l'avant-dernier mur. Ceci n'est pas le cas quand l'avant-dernier mur est déjà plein, on a essayé deux autres fonctions d'évaluation ainsi qu'une autre fonction de sélection, mais toutes ces méthodes donnaient de pires résultats.

Pour éviter de tomber dans un minimum local, on s'assure de faire des relances aléatoires. Plus concrètement, si aucun des voisins immédiats n'améliorent la solution (on a atteint un minimum local), l'ordre de placement des pièces est mélangé aléatoirement à l'aide de la fonction `shuffle` du package `random` de Python. Ceci permet d'avoir les bénéfices de l'aléatoire lors des relances tout en gardant l'avantage d'avoir une solution initiale guidée par l'intuition humaine.

3 Résultats

Dans la fonction `solve` de `solver_advanced.py`, on modifie le paramètre `TIME_IN_SECONDS` pour adapter la durée de la recherche locale. Si le temps permis est dépassé, la solution qui est en cours de génération sera ignorée et l'algorithme retournera l'ancienne meilleure solution.

Instance (easy) (max 1 min)	in1	in2	in3	in4	in5
Nombre de murs trouvés	2	3	3	2	3

Instance (hard) (max 5 min)	in1	in2	in3	in4	in5	in6	in7	in8	in9	in10
Nombre de murs trouvés	5	8	14	3	12	14	3	9	17	2

Pour chacune des 15 instances révélées, Célestin devra s'attendre à dépenser une quantité acceptable de dollars. Pour l'instance 3 "hard", il aura même la surprise de dépenser un dollar de moins que prévu!