Projet de Complexité

Maël Audren de Kerdrel - Garance Vallat

04.01.15

1 Étude du problème et de sa complexité algorithmique

Le problème de ce projet est appelé problème Bin Packing, avec la particularité d'être en deux dimensions.
Il s'agit d'un problème d'optimisation combinatoire, qui a de nombreuses applications pratiques, notamment pour le transport de matériau.

Complexité algorithmique

Ce problème est similaire au problème de Bin Packing, qui est NP-complet. Ce problème est un problème de Bin Packing à 2 dimensions, en fait une généralisation.
On peut commencer par réduire le problème Partition à Bin Packing, afin de montrer la NP-difficulté.

Voici une présentation du problème sous forme de décision :
Données :

> Ensemble fini d'éléments U ($taille \in N$), une taille $B \in N$ et un nombre $k \in N$.

Question :

> Peut-on partitionner U en $U_1, U_2, ..., U_k$ sans que la somme des tailles des éléments des U_i dépasse B ?

Rappel du problème Partition sous forme de décision
Données :

> Un ensemble fini d'entiers non négatifs A

Question :

> Est-ce qu'il existe une partition en A en deux ensembles A' et A' $'$ telle que la somme des éléments de A' soit égale à la somme des éléments de A' $'$?

La transformation à effectuer en temps polynomial pour réduire Partition à Bin Packing est la suivante :
Si B est la moitié de la somme des tailles des éléments et $k = 2$, alors un Bin Packing existe si et seulement si un Partition existe.

Problème de Bin Packing 2D sous forme de décision :
DONNÉES :

> Un ensemble fini d'élements U avec une largeur $L \in N$ et une hauteur $H \in N$. Une dimension de boite B avec une largeur $J \in N$ et une hauteur $K \in N$, un nombre n.

QUESTION :

> Peut-on partitionner U en $U_1, U_2, ..., U_k$ sans que la somme des longueurs et largeurs des éléments des U_i dépasse la longueur et largeur de B ?

La transformation en temps polynomial pour réduire Bin Packing à Bin Packing 2D est la suivante :
La taille des éléments U =la hauteur H de chaque U. La largeur de chaque $U = 1$. La taille B = la hauteur K de la boite, et sa largeur $J = 1$ et enfin $n = k$.
Dans ces conditions, on a un Bin Packing si et seulement si on a un Bin Packing 2D. En effet, un Bin packing est un Bin Packing 2D lorsque les tailles de chaque élément U correspondent à la longueur de chaque élément du Bin packing 2D, et la largeur des éléments du Bin Packing 2D est la largeur de la boite du 2D, soit 1. La longueur de la boite est la taille B du Bin Packing 1 Dimension.
Donc Bin Packing 2D est NP-Difficile.
Par conséquent, il n'existe pas d'algorithme exact pour résoudre en temps polynomial ce problème. Cependant, on peut trouver des algorithmes d'approximation.

2 Présentation de notre solution

Algorithme choisi

Nous avons choisi d'utiliser une variante de l'algorithme *First Fit Decreasing*, en se servant d'"étagère" pour remplir les boites au fur et à mesure, tout en pouvant y revenir.
Pour cet algorithme, on effectue d'abord une série d'initialisations : On enregistre les dimensions de la boite, afin de pouvoir librement y revenir, on établit le nombre de boites à 0 pour commencer, on crée une structure où on maintiendra une liste des étagères "ouvertes", et on en crée une première, dans la première boite.
Enfin, on trie de façon décroissante la liste des rectangles, en fonction de leur largeur uniquement.

L'algorithme en lui-même consiste à tester, pour chaque rectangle, s'il entre dans un étage actuellement ouvert. Un rectangle peut entrer dans un étage s'il reste en largeur un espace suffisant pour qu'il s'y insère, et que la hauteur maximum de l'étage le permet aussi.
La hauteur maximum de l'étage est calculée ainsi : si l'étage est le "dernier" au sens du plus haut, de la boite, sa hauteur actuelle peut être augmentée d'autant que de toute la hauteur restante de la boite si besoin est.
Après le test de tous les étages ouverts, si le rectangle n'est pas encore positionné, on peut créer une nouvel étage dans une boite existante, ou bien créer une nouvelle boite si cet espace ne peut pas suffire.
Un étage est "ouvert" tant qu'il ne remplit pas la largeur de la boite.

Complexité de cet algorithme

Avec n le nombre de rectangles à placer.
Avant la recherche d'une position pour chaque rectangle, notre algorithme les trie. Cela provoque une complexité minimum de notre algorithme de $nlogn$.
Avec la recherche dans la liste des étagères laquelle est la première adaptée pour notre rectangle, on a dans le pire des cas une complexité de n^2. Dans le meilleur des cas, la complexité est de $nlogn$: si la boite est suffisamment grande pour contenir sur un seul étage tous les rectangles. En moyenne, le nombre de d'étages varie de 1 pour le premier rectangle, et augmente

ensuite au fur et à mesure, sans jamais dépasser le nombre de rectangles. La complexité est donc de $O(nlogn + nk)$, avec k le nombre d'étages, qui varie entre 1 et k, le nombre de boites.

Efficacité de cet algorithme

On sait déjà que pour tout jeu de données, $FF(D)$ le résultat de notre algorithme $< 2OPT(D)$, avec $OPT(D)$ la solution optimale. En effet, la somme des surfaces contenues dans la première et la deuxième boite est supérieure à 1. C'est également le cas pour la boite i et la boite $i + 1$. On a donc :

$$FFD(D) = k < 2\sum_{i=1}^{n} u_i$$

Enfin, comme $\sum_{i=1}^{n} u_i \leq OPT(D)$, on peut confirmer que

$$FFD(D) < 2OPT(D)$$

Au final, on trouve dans la littérature que cet algorithme a comme bornes :

— haute :

$$FFD(D) < \frac{11}{9}OPT(D) + 4$$

— basse :

$$FFD(D) < \frac{11}{9}OPT(D)$$

Autres possibilités

Plusieurs possibilités d'optimisation existent, tout en conservant les contraintes d'orientation des rectangles, et leur intégrité. Ces optimisations nécessitent également d'obtenir tous les rectangles d'un coup, et pas de manière inline, comme dans notre propre implémentation.
Cependant, même en étant loin d'être exponentiels, ces algorithmes demandent beaucoup plus de mémoire, et surtout de puissance pour obtenir une solution rapide. Ainsi, par exemple, on peut mélanger différents algorithmes "faciles" :

— On dispose les grands rectangles directement, à l'aide d'un algorithme de flots maximums. Le flot nous permettra d'avoir un nombre de boites minimum pour ces rectangles.

— On peut avoir recours à la force brute au contraire, pour les rectangles longs et plats en particulier : ils peuvent tapisser les boites, une fois que les plus gros sont positionnés.
— On peut enfin utiliser l'algorithme FFD pour les plus petits rectangles, dans les places restantes en priorité, avant d'ouvrir de nouvelles boites.

On peut également prendre le problème à l'envers, et après avoir défini un nombre de boites maximum, faire pour chaque boite jusqu'à son remplissage ou l'impossibilité d'ajouter quelque chose, une recherche dans la liste des rectangles pour utiliser celui qui est le plus approprié.

Cependant, ces solutions demandent non seulement plus de place en mémoire, mais aussi plus de puissance d'exécution. Pour de vrais problèmes, il est donc nécessaire d'avoir des machines conséquentes pour procéder à ces améliorations, qui peuvent pourtant apporter une vraie différence dans la qualité de la solution. Nous ne les avons pas choisies, car nous avons souhaité que notre solution reste la plus simple possible, dans les limites de la qualité. En premier lieu, par nécessité de rester performant, et en second lieu, pour éviter de tomber dans le piège d'un algorithme glouton en essayant d'optimiser notre solution.

Références

[1] Jukka Jylänki, *A Thousand Ways to Pack the Bin - A Practical Approach to Two-Dimensional Rectangle Bin Packing*. 2010.

[2] Johny Bond, *Cours de Complexité SI4*. 2014.

[3] ikhil Bansal and Arindam Khan, *Improved Approximation Algorithm for Two-Dimensional Bin Packing*. 2014