TP 12 : Approximation

D'après un sujet d'Anne le Gluher.

Les objectifs de ce TP sont d'étudier le problème du *bin packing*. D'abord avec des algorithmes gloutons d'approximation puis avec un algorithme par séparation et évaluation.

Ce TP est accompagné d'un TD dans lequel on montre que les algorithmes gloutons sont bien des α-approximations du problème et que le problème est NP-Complet.

Ce TP se décline en deux niveaux de difficulté :

- Dans la partie 1, le TP est plus guidé et les structures de données sont fournies. Ce TP utilise le fichier `bin_packing_guidé.ml`.
- Dans la partie 2, le TP est beaucoup moins guidé, les algorithmes à implémenter sont décrits mais les choix de structures et d'implémentation sont libres. Ce TP utilise le fichier `bin_packing_non_guidé.ml`.

On s'intéresse au problème d'optimisation BIN PACKING défini comme suit :

BIN PACKING

Entrée : n objets de volume $v_1, ..., v_n$ dans $\mathbb{N}$ et un volume maximal $V \in \mathbb{N}^*$
Sortie : Un entier m et une fonction $f : [\![1, n]\!] \to [\![1, m]\!]$ tels que $\forall i \in [\![1, m]\!], \sum_{f(v_j)=i} v_j \leq V$ qui minimise m.

Autrement dit, on cherche à ranger les n objets dans m boîtes de volume V de telle sorte à ce que la somme des volumes des objets dans une boîte n'excède jamais V et en utilisant un minimum de boîtes. Un exemple concret dans lequel ce problème intervient est le suivant. On dispose de trains de marchandises pouvant chacun contenir un volume fixé et de marchandises de volumes divers à acheminer : quel est le nombre minimal de trains à utiliser pour pouvoir déplacer toute la marchandise ?

1) Résoudre à la main BIN PACKING de manière exacte lorsque $V = 10$ et les volumes des objets sont $2, 5, 4, 7, 1, 3, 8$.

On suppose à partir de maintenant que le volume de chacun des objets est inférieur au volume commun des boîtes. Car sinon, aucun rangement n'est possible.

Partie 1 : TP guidé

Exercice 1.1 : Algorithmes d'approximation gloutons

Dans la suite, une instance de `BIN PACKING` est représentée par la donnée d'un entier correspondant au volume V des boîtes et d'un tableau contenant en case i le volume du i-ème objet. Une des boîtes de volume V du problème `BIN PACKING` sera implémentée via le type suivant :

```
type box = {remaining_volume : int ; elements : int list}
```

Le premier champ contient le volume libre dans la boîte (si la boîte est vide, c'est V). Le second champ contient le numéro de tous les objets qui sont dans la boîte dans une liste non ordonnée.

> **2)** Écrire une fonction `empty_box : int -> box` qui crée une boîte vide de volume donné.

> **3)** Écrire une fonction `add : int -> int -> box -> box` prenant en entrée un objet, son volume et une boîte et qui renvoie la boîte résultant de l'ajout de cet objet dans la boîte. On suppose, sans le vérifier, que l'objet rentre dans la boîte.

On propose d'étudier trois algorithmes gloutons d'approximation pour `BIN PACKING`. Le premier algorithme s'appelle *next-fit*. Son principe est le suivant : On maintient à jour une boîte courante. Pour chaque objet, *next-fit* détermine s'il peut rentrer dans la boîte courante : si oui, il l'y place, si non, il ferme **définitivement** la boîte courante, ouvre une nouvelle boîte qui devient la nouvelle boîte courante et y place l'objet.

> **4)** Écrire une fonction `add_next : int-> int -> box list -> int -> box list`. Ses paramètres sont : le numéro i d'un objet, son volume, une liste de boîtes L (on suppose sans le vérifier que la capacité maximale de chaque boîte est respectée) et le volume maximal de chacune de ces boîtes. Elle renvoie une liste correspondant à la nouvelle répartition des objets dans les boîtes après ajout de l'objet i selon la stratégie *next-fit*.
>
> *Indication : quel élément de L est-il judicieux de considérer comme étant la boîte courante ?*

> **5)** En déduire une fonction `next_fit : int -> int arry -> int` prenant en entrée le volume maximal des boîtes, un tableau de taille n contenant en case i le volume du i-ème objet et renvoyant le nombre de boîtes nécessaires pour stocker les n objets selon l'algorithme d'approximation *next-fit*.

Le deuxième algorithme utilise l'heuristique *first-fit*. Il consiste à maintenir une liste (initialement vide) de boîtes $B_1, ..., B_k$. Il considère ensuite chaque objet et le place dans la première boîte qui peut le contenir en commençant par B_1. S'il ne rentre dans aucune boîte, il crée la boîte B_{k+1} et y place l'objet.

> **6)** Écrire une fonction récursive `add_first : int -> int -> box list -> int` ayant la même spécification que `add_next` mais utilisant la stratégie *first-fit*.

7) En déduire une fonction `first_fit : int -> int arry -> int` fonctionnant comme `next_fit` mais utilisant la stratégie *first-fit*.

Le dernier algorithme étudié est FFD (*first-fit decreasing*). Le principe est le même que pour l'algorithme *first-fit* à cette différence près que les objets sont considérés par ordre de volume décroissant.

8) Écrire une fonction `ffd : int -> int array -> int` de même spécification que `first_fit` mais utilisant cette nouvelle stratégie gloutonne. On pourra utiliser la fonction `Array.sort`.

Exercice 1.2 : Algorithme par séparation et évaluation

Afin de pouvoir comparer les résultats de ces algorithmes d'approximation au résultat optimal pour les petites instances, on implémente enfin une fonction de résolution de BIN PACKING exacte (qui sera donc exponentielle au pire cas) en utilisant une stratégie par séparation et évaluation (ou *branch and bound*).

Un algorithme par séparation et évaluation utilise le même principe qu'un algorithme par force brute avec retour sur trace auquel on ajoute une optimisation. Dans le retour sur trace, on revient en arrière dès que la solution que l'on est en train de construire n'est plus valide. Avec un algorithme par séparation et évaluation on revient également en arrière si la solution qu'on est en train de construire sera forcément mois bonne que la meilleure solution trouvée jusqu'à maintenant.

Les algorithmes par séparation et évaluation sont en quelque sorte l'équivalent pour les algorithmes par force brute avec retour sur trace de l'élagage alpha-beta pour l'algorithme min-max.

Pour le problème BIN PACKING, le principe est le suivant :

- On note m le nombre minimal de boîtes pour stocker les éléments qu'on connaît pour le moment. On initialise m avec le nombre de boîtes trouvé par `ffd` de sorte à avoir déjà une bonne borne au départ.
- On parcourt l'arbre des rangements partiels dans m boîtes. Un nœud correspond à une façon de ranger les i premiers objets dans nos boîtes et ses fils correspondent aux rangements qu'on pourrait obtenir à partir du précédent en y ajoutant le $(i+1)$-ème objet. Dès qu'on obtient un rangement complet nécessitant moins de m boîtes, on met à jour m. Si un rangement partiel nécessiterait plus de m boîtes, on élague en arrêtant l'extension de ce rangement partiel.

9) Écrire une fonction `bb_bnb : int -> int array -> int` résolvant BIN PACKING de manière exacte par séparation et évaluation.

10) Vérifier que les fonctions `next_fit`, `first_fit` et `ffd` renvoient un résultat cohérent sur l'entrée décrite à la question 0. Ordonner empiriquement les facteurs d'approximation de `next_fit`, `first_fit` et `ffd`. On pourra consolider son intuition en appliquant ces fonctions à des instances générées aléatoirement via la fonction fournie `random_volumes`.

Partie 2 : TP non guidé

L'objectif de cette partie est d'implémenter trois algorithmes gloutons pour `BIN PACKING` dont on montrera que ce sont bien des algorithmes d'approximation à facteur constant pour ce problème.

1) Décrire un type `box` permettant de modéliser une boîte. Le calcul du volume libre dans une boîte doit se faire en temps constant de même que l'ajout d'un objet dans une boîte.

Suivent les descriptions des algorithmes à implémenter :

- Le premier algorithme à implémenter s'appelle *next-fit*. Son principe est le suivant : il maintient à jour une boîte courante. Pour chaque objet, *next-fit* détermine s'il peut rentrer dans la boîte courante : si oui, il l'y place, si non, il ferme **définitivement** la boîte courante, ouvre une nouvelle boîte qui devient la nouvelle boîte courante et y place l'objet.
- Le deuxième algorithme utilise l'heuristique *first-fit*. Il consiste à maintenir une liste, initialement vide, de boîtes $B_1, ..., B_k$. Il place chaque objet dans la première boîte qui peut le contenir en commençant par B_1. S'il ne rentre dans aucune boîte, il crée la boîte B_{k+1} et y met l'objet.
- Le dernier algorithme étudié est FFD (*first-fit decreasing*). Le principe est le même que pour l'algorithme *first-fit* à cette différence près que les objets sont considérés par ordre de volume décroissant.

Avant de progresser, il est conseillé de réfléchir à la façon dont représenter une instance de BIN PACKING.

2) Écrire une fonction `next_fit` renvoyant le nombre de boîtes nécessaires pour stocker les objets d'une instance de BIN PACKING selon l'algorithme d'approximation *next-fit*. Reprendre cette question pour une fonction `first_fit` et une fonction `ffd` (*first-fit decreasing*).

3) Implémenter par ailleurs une fonction `bb_bnb` résolvant de manière exacte le problème BIN PACKING en utilisant une stratégie par séparation et évaluation. Sur l'instance telle que $V = 101$ et les volumes des objets sont $27, 11, 41, 43, 42, 54, 34, 11, 2, 1, 17, 56, 42, 24, 31, 17, 18, 19, 24, 35, 13, 17, 25$, votre algorithme devrait répondre 6 de manière quasiment instantanée.

4) Vérifier que les fonctions `next_fit`, `first_fit` et `ffd` renvoient un résultat cohérent sur l'entrée décrite à la question 0. Ordonner empiriquement les facteurs d'approximation de `next_fit`, `first_fit` et `ffd`. On pourra consolider son intuition en appliquant ces fonctions à des instances générées aléatoirement via la fonction fournie `random_volumes`.