TP 17

BIN PACKING

À partir d'un sujet dû à Aude Le Gluher.

Le problème dit BinPacking est défini ainsi :

Instance : un entier naturel C et une famille $X = x_0, \dots, x_{n-1}$ d'entiers naturels

Solution admissible : une partition de X en $B_0 \sqcup \dots \sqcup B_{k-1}$ telle que $\sum_{x \in B_i} x \leqslant C$ pour tout i

Optimisation : minimiser k

Autrement dit, on nous donne n *objets* de *volumes* $x_0, \dots, x_{n-1}$, et l'on dispose de *boîtes* de *capacité* C. Il faut répartir les objets dans k boîtes de manière à ce que la somme des volumes reste toujours inférieure à la capacité, en minimisant le nombre k de boîtes utilisées.

Remarque

Des problèmes de ce type se posent en particulier quand on s'intéresse à l'allocation mémoire (quand on veut implémenter `malloc`, essentiellement) : les boîtes représentent les zones mémoire (contiguës) disponibles, les objets les demandes d'allocation. Une difficulté supplémentaire dans ce cas est qu'on doit décider comment traiter chaque objet dès qu'il arrive, sans savoir quels autres objets arriveront plus tard (on parle d'algorithme *online*).

▶ **Question 1** Donner une solution optimale pour BinPacking sur l'instance suivante : $C = 10$ et $X = 2, 5, 4, 7, 1, 3, 8$.

1 Caractère NP-complet

On admet (*cf* exercice **??**) que le problème SubsetSum, défini comme suit, est NP-complet :

Instance : un multi-ensemble A d'entiers naturels et un entier $s > 0$

Question : existe-t-il $B \subseteq A$ tel que $\sum_{x \in B} x = s$.

▶ **Question 2** Définir le problème de décision BPD associé au problème d'optimisation BinPacking.

▶ **Question 3** On considère le problème Partition :

Instance : n entiers naturels $x_0, \dots, x_{n-1}$

Question : Existe-t-il $I \subseteq [0 \dots n-1]$ tel que $\sum_{i \in I} x_i = \sum_{i \notin I} x_i$?

Montrer que Partition est NP-complet. *On pourra partir d'une instance de SubsetSum et lui ajouter deux éléments bien choisis. Trouver ces deux éléments n'est pas facile !*

▶ **Question 4** En déduire que BPD est NP-complet.

2 Stratégies gloutonnes

On va considérer dans ce sujet trois stratégies gloutonnes pour le problème BinPacking. On suppose à partir de maintenant que les poids des différents objets sont majorés par C, puisque sinon il n'y a clairement aucune solution.

- La stratégie **next-fit** considère les objets dans l'ordre d'arrivée, et ajoute ces objets dans la boîte courante tant que c'est possible. Quand ce n'est plus le cas, on ferme (définitivement) cette boîte et l'on en crée une nouvelle, qui devient la boîte courante.

- La stratégie **first-fit** considère aussi les objets dans l'ordre d'arrivée, mais maintient une liste (initialement vide) de boîtes $B_0, \dots, B_{k-1}$. À chaque fois que l'on considère un objet, on cherche le premier i tel que l'objet rentre dans la boîte B_i :
  — s'il en existe un, on ajoute l'objet dans cette boîte ;
  — sinon, on crée une nouvelle boîte B_k dans laquelle on place l'objet.
- La stratégie **first-fit-decreasing** procède comme **first-fit** mais commence par trier les objets par ordre décroissant de volume.

On représente une instance de BINPACKING par un entier `capacity` (la capacité C des boîtes) et une liste d'entiers correspondant aux poids des objets ; on supposera que tous les poids sont inférieurs ou égaux à la capacité.

```
type instance = int * int list
```

► **Question 5** Proposer un type `box` pour représenter une boîte (et son contenu). On souhaite pouvoir déterminer le volume disponible dans la boîte, et y ajouter un objet, en temps constant.

► **Question 6** Écrire une fonction `next_fit` qui résout le problème en utilisant la stratégie *next-fit*.

```
val next_fit : instance -> box list
```

► **Question 7** Écrire de même une fonction `first_fit`.

► **Question 8** Écrire une fonction `first_fit_decreasing`.

► **Question 9** Quelle solution obtient-on pour l'instance de la question 1 avec les différentes stratégies ?

► **Question 10** Déterminer la complexité dans le pire cas des fonctions qui correspondent aux différentes stratégies.

► **Question 11** **À ne traiter qu'à la fin.** On considère la variante suivante de *first-fit-decreasing*, appelée *best-fit-decreasing* : on commence par trier les objets par volume décroissante, puis à chaque étape on ajoute l'objet à la boîte *la plus remplie* dans laquelle il rentre (ou l'une quelconque des telles boîtes en cas d'égalité). On crée une nouvelle boîte si nécessaire.
Proposer un algorithme (qu'on ne demande pas d'implémenter) permettant d'appliquer cette stratégie avec une complexité $O(n \log n)$.

3 Analyse des approximations

3.1 Analyse de *next-fit*

On note m le nombre de boîtes utilisées par la stratégie *next-fit* sur une instance $C, X = (x_0, \dots, x_{n-1})$ et m^* le nombre optimal de boîtes sur cette même instance. On numérote B_0, B_{m-1} les boîtes utilisées par *next-fit*, dans l'ordre de leur utilisation et l'on note v_i le volume total des objets présents dans la boîte v_i et $V = \sum_{i=0}^{n-1} x_i$.

► **Question 12** Montrer que $v_i + v_{i+1} > C$ pour $0 \leqslant i < m - 1$.

► **Question 13** En déduire que *next-fit* fournit une 2-approximation pour BINPACKING.

3.2 Analyse de *first-fit-decreasing*

On reprend les notations de la partie précédente, avec m le nombre de boîtes utilisées par *first-fit-decreasing* et m^* le nombre optimal de boîtes. On note $x_0 \geqslant \dots \geqslant x_{n-1}$ les poids.

On considère la boîte B_j avec $j = \lfloor \frac{2m}{3} \rfloor$, et l'on note x le volume maximal d'un objet rangé dans la boîte B_j.

Cas $x > C/2$

▶ **Question 14** Montrer que $m \leqslant \frac{3}{2}m^*$.

Cas $x \leqslant C/2$ On note v le plus grand volume disponible (en fin d'exécution) dans les boîtes $B_0, \ldots, B_{j-1}$.

▶ **Question 15** Montrer que $\sum_{l=j}^{m-1} v_l > v + 2v(m-j-1)$.

▶ **Question 16** Conclure.

3.3 Difficulté de l'approximation

▶ **Question 17** Soit $\varepsilon > 0$. Par réduction de Partition, montrer que s'il existe un algorithme fournissant en temps polynomial une $(\frac{3}{2} - \varepsilon)$-approximation pour BinPacking, alors $\mathsf{P} = \mathsf{NP}$.

4 Résolution exacte

▶ **Question 18** Écrire une fonction `solve` calculant une solution optimale pour une instance de BinPacking. On utilisera (de manière assez basique) la technique dite *Branch and Bound* pour obtenir une fonction raisonnablement efficace. On doit par exemple trouver une solution utilisant 6 boîtes pour l'instance suivante, de manière presque instantanée.

```
let test_exact =
  (101,
   [27; 11; 41; 43; 42; 54; 34; 11; 2; 1; 17; 56;
    42; 24; 31; 17; 18; 19; 24; 35; 13; 17; 25])
```

```
val solve : instance -> box list
```

Solutions

▶ **Question 1** On peut répartir en $\{2,8\},\{5,4,1\},\{7,3\}$. Les trois boîtes étant pleines, cette solution est clairement optimale.

▶ **Question 2**

Instance : un entier naturel C, une famille $X = x_0, \dots, x_{n-1}$ d'entiers naturels et un entier k

Question : existe-t-il une partition $B_0, \dots, B_{l-1}$ de X telle que $l \leqslant k$ et, pour tout $j \in [0 \dots l-1]$, $\sum_{x \in B_j} x \leqslant C$?

▶ **Question 3** Partition est clairement dans NP : on peut prendre I comme certificat, et vérifier en temps polynomial que les deux sommes sont égales.
Considérons une instance A, s de SubsetSum, et soit $S = \sum_{x \in A} x$. On construit en temps polynomial l'instance $A' = A \cup \{2s, S\}$ de Partition. La somme des éléments de A' est $2S + 2s$, donc l'instance est positive si et seulement si on peut trouver $B' \subseteq A'$ tel que $\sum_{x \in B'} x = S + s$.

- Si A, s est une instance positive, il existe $B \subseteq A$ tel que $\sum_{x \in B} x = s$. Dans ce cas, on pose $B' = B \cup \{S\}$ et l'on a bien $\sum_{x \in B'} x = S + s$: A' est une instance positive de Partition.
- Inversement, supposons que A' soit une instance positive, et soit alors $B' \sqcup B''$ une partition en deux ensembles de même somme. Supposons sans perte de généralité que $S \in B'$. On ne peut avoir $2s \in B'$, puisque la somme des éléments de B' vaut seulement $S + s$. En posant $B = B' \setminus \{S\}$, on a donc $B \subseteq A$ et $\sum_{x \in B} x = s$, donc A, s est une instance positive de SubsetSum.

On a montré que SubsetSum $\leqslant_p$ Partition, et l'on sait que SubsetSum est NP-complet : on en déduit que Partition est NP-complet (puisque l'on sait déjà qu'il est dans NP).

▶ **Question 4** BPD est dans NP : on peut utiliser comme certificat la liste $b_0, \dots, b_{n-1}$ des numéros des boîtes dans lesquelles les objets sont rangés, qui est bien de taille polynomiale. Il faut ensuite vérifier, ce qui se fait en temps polynomial, que ces numéros sont tous dans $[0 \dots k-1]$ et que la somme des éléments présents dans chaque boîte est inférieure ou égale à C.
Considérons maintenant une instance A de Partition, de somme totale S, et construisons l'instance $A, \lfloor S/2 \rfloor, 2$ de BPD, ce qui se fait en temps polynomial. Cette instance est positive si et seulement si il est possible de partager A en deux ensembles de somme inférieure ou égale à $\lfloor S/2 \rfloor$. On voit immédiatement que cela équivaut à ce que les deux sommes soient égales à $S/2$, et donc à ce que l'instance de Partition soit positive. Donc Partition $\leqslant_p$ BPD, ce qui permet de conclure.

▶ **Question 5** On prend un type enregistrement avec deux champs : le volume restant disponible et la liste des objets présents. On peut utiliser des champs mutables, mais ce n'est pas nécessaire.

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

▶ **Question 6** On définit quelques fonctions auxiliaires qui seront utiles par la suite :

```
let empty_box capacity =
  {available = capacity; elements = []}

let add_object x box =
  assert (box.available >= x);
  {available = box.available - x; elements = x :: box.elements}

let singleton capacity x =
  add_object x (empty_box capacity)
```

La fonction `next_fit` ne pose pas de difficulté : on utilise une fonction auxiliaire prenant en argument les objets qui restent à traiter et la boîte « active » :

```
let next_fit (capacity, objects) =
  let rec aux remaining_objects current_box =
    match remaining_objects with
    | [] -> [current_box]
    | x :: xs ->
      if x <= current_box.available then aux xs (add_object x current_box)
      else current_box :: aux remaining_objects (empty_box capacity) in
  aux objects (empty_box capacity)
```

▸ **Question 7** On commence par définir une fonction `add_first_fit` qui ajoute un objet à la première boîte d'une liste susceptible de l'accueillir, ou crée une nouvelle boîte si aucune ne convient. Pour ce faire, cette fonction prend aussi la capacité en argument :

```
(* int -> box list -> int -> box list *)
let rec add_first_fit capacity boxes x =
  match boxes with
  | [] -> [singleton capacity x]
  | b :: bs ->
    if x <= b.available then (add_object x b) :: bs
    else b :: add_first_fit capacity bs x
```

La fonction `first_fit` s'écrit ensuite sans difficulté :

```
let first_fit (capacity, objects) =
  let rec aux remaining_objects boxes =
    match remaining_objects with
    | [] -> boxes
    | x :: xs ->
      let boxes' = add_first_fit capacity boxes x in
      aux xs boxes' in
  aux objects []
```

▸ **Question 8** On trie puis on appelle `first_fit` :

```
let first_fit_decreasing (capacity, objects) =
  let sorted = List.sort (fun x y -> y - x) objects in
  first_fit (capacity, sorted)
```

▶ **Question 9** On obtient :

Next-fit : $(5,2),(4),(1,7),(3),(8)$

First-fit : $(1,5,2),(3,4),(7),(8)$

First-fit-decreasing : $(2,8),(3,7),(1,4,5)$

On peut observer que la solution renvoyée par *first-fit-decreasing* est optimale ici, même si cela n'a aucune raison d'être vrai en général.

▶ **Question 10**

- Dans `next_fit`, on traite chaque objet en temps constant, et le coût total est donc en $O(n)$.
- Dans `ffirst_fit`, chaque objet est traité en temps $O(n)$ (un parcours partiel de la liste des boîtes, de longueur majorée par n), ce qui donne du $O(n^2)$ au total.
- Pour `first_fit_decreasing`, on commence par trier en $O(n \log n)$ puis l'on applique `first_fit` en $O(n^2)$. On a donc encore du $O(n^2)$ au total.

▶ **Question 11** On commence par trier en temps $O(n \log n)$. Il faut ensuite faire en sorte que la recherche de la boîte dans laquelle ajouter l'objet se fasse en temps $O(\log n)$. Pour ce faire, on range les boîtes dans un arbre rouge-noir en utilisant comme clé le volume disponible. La recherche dans un ABR du plus petit élément supérieur à une certaine valeur peut se faire en temps proportionnel à la hauteur, et donc ici en $O(\log n)$ puisque l'arbre est équilibré et qu'il y a au plus n boîtes. On supprime ensuite la boîte sélectionnée de l'arbre avant de la réinsérer avec son nouveau volume disponible.

▶ **Question 12** On n'a ouvert la boîte B_{i+1} que pour y ranger un objet x qui ne rentrait pas dans B_i. On a donc $v_{i+1} \geqslant x$ (on ne peut que rajouter des objets ensuite) et $v_i + x > C$ (sinon on aurait rangé x dans B_i). On en déduit $v_i + v_{i+1} > C$.

▶ **Question 13** Notons déjà qu'on a $\sum_{i=0}^{m-1} v_i = V$ (chaque objet est rangé dans une boîte) et $m^*C \geqslant V$ (la somme des volumes des boîtes de la solution optimale est minorée par le volume total des objets).
D'autre part, la question précédente donne $\sum_{i=0}^{m-2}(v_i + v_{i+1}) > (m-1)C$, c'est-à-dire $2\sum_{i=0}^{m-1} v_i > (m-1)C + v_0 + v_{m-1}$. On en déduit $2V > (m-1)C$ et donc $2m^*C > (m-1)C$. On a finalement $m < 2m^* + 1$, donc $m \leqslant 2m^*$ puisque m et m^* sont des entiers : on obtient bien une 2-approximation.

▶ **Question 14** Comme $x > C/2$ et comme on traite les objets par volume décroissant, on a $x_i > C/2$ pour $0 \leqslant i \leqslant j$. Il faut donc au moins $j+1$ boîtes pour ranger ces objets (on ne peut en mettre deux dans la même boîte), ce qui montre que $m^* \geqslant j + 1 > \frac{2m}{3}$. On en déduit $m \leqslant \frac{3}{2}m^*$.

▶ **Question 15** On ne range un objet y dans une boîte i avec $i \geqslant j$ que s'il ne rentre dans aucune boîte i' avec $0 \leqslant i' < j$. Il faut donc que le volume disponible à cet instant dans $B_{i'}$ soit strictement inférieur à y, et donc *a fortiori* que le volume disponible dans $B_{i'}$ en fin d'exécution soit strictement inférieur à y. Tous les objets des boîtes $B_j, \dots, B_{m-1}$ ont donc des volumes supérieurs à v. De plus, ils ont été traités après l'objet x, donc $y \leqslant x \leqslant C/2$: à part éventuellement la dernière boîte, il y aura au moins deux objets par boîte.

Ainsi, on a :

- $v_i > 2v$ pour $j \leqslant i < m-1$;
- $v_{m-1} > v$

On en déduit $\sum_{i=j}^{m-1} v_i > v + 2v(m-j-1)$.

▶ **Question 16** Pour chaque $i \in [0 \dots j-1]$, on a par définition $v_i \geqslant C - v$, donc $\sum_{i=0}^{j-1} v_i \geqslant jC - jv$. En combinant avec la question précédente, on obtient $\sum_{i=0}^{m-1} v_i > jC + v(2m - 3j - 1)$.

- Si $m = 3m'$ avec $m' \in \mathbb{N}$, alors $V > 2m'C - v$, d'où $m^* > 2m' - \frac{v}{C}$. Comme $0 \leqslant \frac{v}{C} < 1$ et que m' et m^* sont des entiers, on en déduit $m^* \geqslant 2m'$ d'où $m \leqslant \frac{3}{2}m^*$.
- Sinon, $3j \leqslant 2m - 1$ donc $V > jC$ puis $m^* > j$ et donc $m^* \geqslant j + 1$. Donc $m^* > \frac{2}{3}m$

On obtient donc bien une 3/2-approximation.

▸ **Question 17** Considérons une instance $x_0, \dots, x_{n-1}$ de Partition, et posons $S = \sum_{i=0}^{n-1} x_i$.

En supposant que l'on dispose d'une $(3/2-\varepsilon)$-approximation pour BinPacking en temps polynomial, on construit en temps polynomial l'instance $(2x_0, \dots, 2x_{n-1}), S$ de BinPacking et on lui applique cet algorithme d'approximation :

- si l'on obtient une solution utilisant deux boîtes (une seule boîte n'est clairement pas possible), alors de manière immédiate l'instance de Partition était positive ;
- sinon, on obtient une solution utilisant $m \geqslant 3$ boîtes. Cela signifie que $(3/2-\varepsilon)m^* \geqslant 3$, et donc $m^* > 2$. Or si l'instance de Partition était positive, une solution avec deux boîtes serait possible.

Ainsi, une $(3/2-\varepsilon)$-approximation en temps polynomial pour BinPacking fournirait un algorithme en temps polynomial pour décider Partition, et impliquerait donc $\mathsf{P} = \mathsf{NP}$.

Remarque

On a en fait $m \leqslant \frac{11}{9}m^* + \frac{2}{3}$ si m est fourni par *first-fit-decreasing* : asymptotiquement, on fait donc nettement mieux qu'une 3/2-approximation.

▸ **Question 18** On utilise *first-fit-decreasing* pour obtenir une première valeur, et on arrête l'exploration d'une branche dès que le nombre de boîtes actuellement utilisé est supérieur ou égal au plus petit nombre de boîtes trouvé pour une solution complète. De nombreuses optimisations supplémentaires sont bien sûr possibles.

```
let solve (capacity, object_list) =
  let objects = Array.of_list object_list in
  Array.sort (fun x y -> y - x) objects;
  let n = Array.length objects in
  let first_guess = first_fit_decreasing (capacity, object_list) in
  let best_boxes = ref (Array.of_list first_guess) in
  let best_k = ref (Array.length !best_boxes) in
  let boxes = Array.make !best_k (empty_box capacity) in
  let rec explore k next_object_index =
    if next_object_index = n then (
      best_k := k;
      best_boxes := Array.copy boxes
    ) else if k < !best_k then (
      let x = objects.(next_object_index) in
      for i = 0 to k - 1 do
        if x <= boxes.(i).available then (
          let b = boxes.(i) in
          boxes.(i) <- add_object x b;
          explore k (next_object_index + 1);
          boxes.(i) <- b
        )
      done;
      if k < !best_k - 1 then (
        boxes.(k) <- singleton capacity x;
        explore (k + 1) (next_object_index + 1);
        boxes.(k) <- empty_box capacity
      )
    ) in
  explore 0 0;
  Array.to_list (Array.sub !best_boxes 0 !best_k)
```

Un exemple d'instance pour laquelle la résolution exacte donne une solution strictement meilleure que celle de *first-fit-decreasing* :

```
# let test_exact =
  (101,
   [27; 11; 41; 43; 42; 54; 34; 11; 2; 1; 17; 56;
    42; 24; 31; 17; 18; 19; 24; 35; 13; 17; 25]);;
val test_exact : int * int list = ...
# List.length (first_fit_decreasing test_exact);;
- : int = 7
# solve test_exact;;
- : int * box list =
 [{available = 0; elements = [2; 43; 56]};
  {available = 1; elements = [11; 35; 54]};
  {available = 0; elements = [17; 42; 42]};
  {available = 1; elements = [25; 34; 41]};
  {available = 0; elements = [19; 24; 27; 31]};
  {available = 0; elements = [1; 11; 13; 17; 17; 18; 24]}]
```