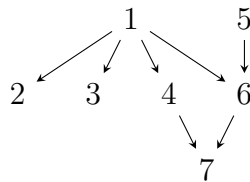

Devoir maison n°3 - À rendre le 28 février 2024

Algorithme de Branch-and-bound pour le sac-à-dos

Notions abordées

- problème de maximisation KNAPSACK et son relâché fractionnaire
- résolution exacte par énumération de tous les sous-ensembles
- résolution exacte par programmation dynamique en temps pseudo-polynomial
- obtention d'un minorant grâce à un algorithme glouton
- obtention d'un majorant en résolvant le relâché fractionnaire
- résolution exacte par Branch-and-bound

Organisation du devoir. Dans ce devoir maison, on s'intéresse au problème d'optimisation du sac-à-dos (*knapsack* en anglais), rappelé ci-dessous. On étudiera (et implémentera en OCAML) plusieurs algorithmes pour résoudre ce problème : un algorithme d'énumération exhaustive (section 2), un algorithme de programmation dynamique (section 3), un algorithme de Branch-and-Bound (section 7). Les dépendances entre les sections de ce devoir sont résumées par le diagramme de Hasse ci-dessous.



$$\text{KNAPSACK} : \begin{cases} \text{Entrée} : n \in \mathbb{N}, (v_i)_{i \in \llbracket 1, n \rrbracket} \in \mathbb{N}_*^n, (w_i)_{i \in \llbracket 1, n \rrbracket} \in \mathbb{N}_*^n, P \in \mathbb{N} \\ \text{Sortie} : \arg \max \{ \sum_{i=1}^n v_i x_i \mid \sum_{i=1}^n w_i x_i \leq P \} \end{cases}$$

Pour une instance (n, v, w, P) , on appelle **objets** les éléments de $\llbracket 0, n - 1 \rrbracket$, et pour $i \in \llbracket 0, n - 1 \rrbracket$, v_i est appelée la **valeur** de l'objet i et w_i son **poids**. Enfin P est appelé la **capacité** du sac.

Afin de représenter les instances du problème, on se munit du type `sad♣` défini comme suit.

```
1 type sad =
2   {
3     n : int;           (* le nombre d'objets *)
4     wi : int array;   (* le tableau des poids, de taille n *)
5     vi : int array;   (* le tableau des valeurs, de taille n *)
6     p : int           (* le poids maximal du sac à dos, >=0 *)
7   }
```

♣. `sac-à-dos`

Dans toute la suite du sujet, on supposera que les objets d'une instance sont toujours triés par rapports valeur/poids décroissants. Ainsi, si e est un objet de type `sad`, la suite $\left(\frac{e.vi.(i)}{e.wi.(i)}\right)_{i \in \llbracket 0, e.n-1 \rrbracket}$ est décroissante. Ci-dessous un exemple de tel objet.

```
1 | let (ex: sad) =
2 |   {
3 |     n = 6 ;
4 |     wi = [|6 ; 8 ; 10; 14; 2; 5|] ;
5 |     vi = [|13 ; 16; 19; 24; 3; 5|] ;
6 |     p = 20 ;
7 |   }
```

Sous-instances et masques. En prévision de la dernière partie de ce DM où l'on implémentera un algorithme de Branch-and-Bound pour le problème `KNAPSACK`, on se munit d'une notion de **sous-instance** correspondant aux restrictions imposées par le fait de choisir de prendre ou non certains objets. En effet, dans le schéma de Branch-and-Bound considéré, on est amené à résoudre des instances qui se définissent à partir de l'instance initiale par les décisions qui ont déjà été prises, par exemple la décision de sélectionner l'objet 1, ou bien la décision de ne pas sélectionner l'objet 4. Ainsi, pour une instance initiale fixée, une sous-instance est définie par un ensemble de décisions portant sur les objets. Pour chaque objet, il y a alors trois cas possibles : ou bien on l'a déjà sélectionné, ou bien on l'a déjà exclu, ou bien on n'a encore rien décidé le concernant. Pour une instance initiale de taille n , une sous-instance est donc définie par un **masque**, c'est-à-dire la donnée de n valeurs dans "oui, non, peut-être". On représente en machine les masques grâce au type suivant.

```
1 | type masque = bool option array
```

Pour m un objet de type `masque` de taille n et pour $i \in \llbracket 0, n-1 \rrbracket$:

- si $m.(i) = \text{None}$, alors il n'y a aucune contrainte sur l'objet i , il peut être choisi dans la solution comme il peut ne pas l'être ;
- si $m.(i) = \text{Some}(\text{true})$, alors l'objet i doit nécessairement être dans la solution ;
- si $m.(i) = \text{Some}(\text{false})$, alors l'objet i doit ne peut pas être dans la solution.

Ainsi le masque `[|None; Some(true); Some(true); None; Some(false); None|]` représente une sous-instance où les objets 1 et 2 ont déjà été sélectionnés tandis que l'objet 4 a déjà été exclu.

Représentation des solutions. Une solution d'une instance de `KNAPSACK` est un sous-ensemble des objets de cette instance. Une telle solution peut être représentée par l'indicatrice de cet ensemble, elle même encodée par un tableau de booléens. On représente donc les solutions en machine grâce au type suivant.

```
1 | type solution = bool array
```

Par exemple, pour la solution $\{1, 2, 4\}$: `[|false; true; true; false; true; false|]`.

compagnon.ml. On trouvera dans `compagnon.ml` : les définitions des types mentionnés dans l'énoncé, ainsi que des fonctions d'affichage pour les éléments de ces types.

1. Prise en main

- Q. 1 Définir une fonction `est_sad_valide : sad -> bool` prenant en argument une instance et testant si elle est valide, en particulier si les objets sont bien classés par rapports décroissants. On ne passera pas dans le monde des flottants pour comparer deux rapports $\frac{a}{b}$ et $\frac{c}{d}$.
- Q. 2 Définir une fonction `valeur_sol : sad -> solution -> int` et une fonction `poids_sol : sad -> solution -> int` prenant en arguments une instance et une solution et retournant respectivement la valeur et le poids de cette solution.
- Q. 3 Définir une fonction `est_masque_valide : sad -> masque -> bool` prenant en arguments une instance et un masque et testant si le masque est valide, au sens où il représente un choix d'objets compatible avec la capacité du sac. Autrement dit, il s'agit de tester si la sous-instance correspondante admet des solutions.

2. Par force brute

Dans cet exercice on se propose de résoudre le problème du sac-à-dos de manière exacte par énumération exhaustive de l'espace des solutions. En vue de résoudre aussi des sous-instances avec cet algorithme, on énumère en fait, pour une instance de taille n et un masque donnés, les n -uplets de booléens qui sont compatibles avec le masque, et on cherche parmi eux celui représentant la meilleure solution.

- Q. 4 Définir une fonction `next : masque -> solution -> unit` prenant en arguments un masque, un tableau de booléens et modifiant le tableau de booléens pour qu'il contienne le prochain tableau de booléens pour un ordre d'énumération bien choisi. *On pourra par exemple choisir l'ordre induit par l'énumération en binaire.*

Exemples :

Les appels successifs à la fonction `next` avec le masque `[|Some(true); None; None; Some(false); None; None|]` peuvent conduire à la séquence de tableaux suivante.

- d'abord `[|true; false; false; false; false; false|]`
- puis `[|true; true; false; false; false; false|]`,
- puis `[|true; false; true; false; false; false|]`,
- puis `[|true; true; true; false; false; false|]`,
- puis `[|true; false; false; false; true; false|]`,
- puis `[|true; true; false; false; true; false|]`,
- puis `[|true; false; true; false; true; false|]` ...

- Q. 5 Définir une fonction `brute_force : sad -> masque -> (solution * int)` option calculant une solution optimale au problème KNAPSACK, par énumération de l'espace des solutions, pour la sous-instance représentée par les données du problème et le masque.

3. Par programmation dynamique

Dans cet exercice on se propose de calculer une solution optimale au problème du sac-à-dos par programmation dynamique. Dans tout cet exercice, on fixe une instance (n, v, w, P) . On note alors, pour $i \in \llbracket 0, n \rrbracket$ et $j \in \llbracket 0, P \rrbracket$, $s_{i,j}$ la valeur optimale pour l'instance de KNAPSACK constituée des i premiers objets de l'instance initiale (n, v, w, P) et d'un sac de capacité j . Cette famille de valeurs vérifie la définition récursive suivante ♣.

♣. Il convient de ne pas admettre un tel résultat mais de s'assurer qu'on saurait le remonter.

- $\forall j \in \llbracket 0, P \rrbracket, s_{0,j} = 0$
- $\forall (i, j) \in \llbracket 1, n \rrbracket \times \llbracket 0, P \rrbracket, w_i \leq j \Rightarrow s_{i,j} = \max(s_{i-1,j}, v_i + s_{i-1,j-w_i})$
- $\forall (i, j) \in \llbracket 1, n \rrbracket \times \llbracket 0, P \rrbracket, w_i > j \Rightarrow s_{i,j} = s_{i-1,j}$

La valeur optimale de la solution au problème du sac-à-dos est alors $s_{n,P}$.

- Q. 6** Définir une fonction `prog_dyn_tab` : `sad` \rightarrow `int array array` prenant en argument une instance et retournant le tableau des valeurs $(s_{i,j})_{i \in \llbracket 0, n \rrbracket, j \in \llbracket 0, P \rrbracket}$ pour cette instance. On assurera une complexité pire cas en $\mathcal{O}(nP)$.
- Q. 7** En déduire une fonction `prog_dyn` : `sad` \rightarrow `solution * int` prenant en argument une instance et retournant une solution optimale ainsi que sa valeur. On fabriquera cette solution à partir du tableau obtenu grâce à la fonction `prog_dyn_tab`. On assurera une complexité pire cas en $\mathcal{O}(n)$ (hors appel à `prog_dyn_tab`).

Organisation de la suite du sujet. En vue d'implémenter un algorithme de Branch-and-Bound pour résoudre le problème KNAPSACK, on souhaite calculer facilement des bornes sur la valeur optimale d'une instance.

- On obtient un minorant de la valeur optimale grâce à un algorithme glouton, dit **algorithme glouton entier**, qui fournit une solution valide mais pas nécessairement optimale. Cet algorithme fait l'objet de la section suivante.
- On obtient un majorant de la valeur optimale en résolvant le problème relâché $\text{KNAPSACK}_{\mathbb{Q}}$ défini ci-après.

$$\text{KNAPSACK}_{\mathbb{Q}} : \begin{cases} \text{Entrée} : n \in \mathbb{N}, \text{ deux suites } (v_i)_{i \in \llbracket 0, n-1 \rrbracket} \in (\mathbb{N}^*)^n \text{ et } (w_i)_{i \in \llbracket 0, n-1 \rrbracket} \in (\mathbb{N}^*)^n, P \in \mathbb{N} \\ \text{Sortie} : x \in ([0, 1] \cap \mathbb{Q})^n \text{ tel que } \sum_{i=1}^n x_i p_i \leq P \text{ et maximisant } \sum_{i=1}^n x_i v_i \end{cases}$$

Les instances de ce problème sont semblables aux instances de KNAPSACK, mais l'ensemble des solutions pour une instance donnée est élargi : au lieu de devoir trancher entre prendre un objet ou ne pas le prendre, il est ici possible de prendre une fraction de chaque objet. Le poids et la valeur d'une fraction $\rho \in [0, 1] \cap \mathbb{Q}$ d'un objet i sont respectivement ρw_i et ρv_i . Ce problème peut être résolu à l'optimum par un algorithme glouton dit **algorithme glouton fractionnaire**, qui fait l'objet de la section 6.

4. Algorithme glouton entier

Dans cette section, on se propose de résoudre le problème KNAPSACK de manière heuristique par une stratégie gloutonne : on prend en priorité les objets ayant le meilleur rapport valeur/poids. Cet algorithme n'est pas optimal, *i.e.* il ne fournit pas toujours une solution optimale, mais il fournit toujours une solution, et avec la valeur de cette solution un minorant de la valeur optimale.

- Q. 8** Définir une fonction `glouton_n` : `sad` \rightarrow `masque` \rightarrow (`solution * int`) option calculant une solution (non nécessairement optimale) au problème KNAPSACK, pour la sous-instance représentée par l'instance et le masque passés en arguments. Si cette instance n'admet pas de solution le fonction devra retourner None. Dans le cas contraire, elle devra calculer non seulement une solution (en suivant l'heuristique gloutonne), mais aussi sa valeur. On assurera une complexité linéaire.

5. Interlude : implémentation d'un module pour les rationnels

Afin d'éviter la représentation des nombres rationnels au moyen de nombres flottants, ce qui conduirait à des erreurs d'arrondis, on se propose dans cet exercice de se munir d'un module permettant la représentation et la manipulation des nombres rationnels sans passer par les flottants. Dans le

fichier `compagnon.ml` on trouvera le code suivant, squelette de la définition d'un module pour les nombres rationnels.

```
1 module Q = struct
2   type t = int * int
3   (* (a, b) encode a/b*)
4   (* invariant : a ∧ b = 1 et b > 0 et si a = 0 alors b = 1 *)
5   (* NB: vu l'invariant, l'égalité syntaxique de OCaml (a, b)=(c, d)*)
6   (* encode en fait l'égalité sémantique de Q : a/b = c/d *)
7   ...
8 end
```

Comme l'indique la définition de type ci-dessus, on veillera à maintenir la propriété invariante suivante : un élément (a, b) de type `Q.t` vérifie $a \wedge b = 1$ [♣], $b > 0$ et si $a = 0$ alors $b = 1$. En contrepartie, les fonctions manipulant des objets de type `Q.t` peuvent supposer opérer sur des objets vérifiant cet invariant.

Q. 9 Compléter ce module en y ajoutant des fonctions permettant de calculer :

- l'addition de deux rationnels;
- la différence de deux rationnels;
- l'opposé d'un rationnel;
- le produit de deux rationnels;
- la division de deux rationnels;
- le rationnel correspondant à un entier;
- le signe (dans $-1, 0, 1$) d'un rationnel;

et des fonctions permettant de tester :

- si un rationnel est strictement plus petit qu'un autre;
- si un rationnel est plus petit au sens large qu'un autre;
- si un rationnel est un entier.

Pour plus de lisibilité, on pourra définir les fonctions implémentant les opérations courantes sous forme d'opérateur infix. Une note en fin de sujet explique comment.

6. Algorithme glouton fractionnaire

Représentation des solutions fractionnaire. Pour une instance de KNAPSACK_Q de taille n , une solution est définie par la fraction sélectionnée de chaque objet, soit par la donnée de n rationnels. On représente en machine de telles solutions grâce au type suivant.

```
1 | type qsolution = Q.t array
```

La stratégie gloutonne consistant à sélectionner en priorité les objets ayant le meilleur rapport valeur/poids fournit une solution optimale dans le cas du problème fractionnaire[♡].

Q. 10 Définir une fonction `glouton_r : sad -> masque -> (qsolution * Q.t) option` calculant, pour une sous-instance représentée par une instance et un masque, une solution optimale au problème KNAPSACK_Q . Si cette sous-instance n'admet pas de solution la fonction retournera `None`. Dans le cas contraire, la fonction calculera non seulement une solution optimale, mais aussi sa valeur. On assurera une complexité linéaire.

♣. où \wedge désigne le PGCD

♡. Il convient de ne pas admettre un tel résultat mais de s'assurer qu'on saurait le redémontrer, par argument d'échange par exemple

7. Algorithme de Branch-and-Bound

On se propose finalement d'implémenter un algorithme de type Branch-and-Bound résolvant le problème KNAPSACK. De la même manière que dans le cours, cet algorithme est caractérisé par les choix suivants.

- Pour chaque sous-instance, une solution réalisable (non nécessairement optimale) sera fournie par l'heuristique gloutonne, définie en **Q. 8**.
- Pour chaque sous-instance, un majorant des valeurs des solutions sera fourni par l'heuristique gloutonne du problème fractionnaire, définie en **Q. 10**.
- La stratégie de branchement sera la suivante : si une instance doit être découpée, c'est que l'optimum rationnel et l'optimum entier ne coïncident pas, ainsi l'un des objets n'est choisi que de manière fractionnaire dans l'optimum rationnel, on génère alors la sous-instance dans laquelle cet objet a été sélectionné et celle dans laquelle cet objet a été exclu. On notera branchement la procédure qui génère ces deux nouvelles sous-instances à partir d'une sous-instance et d'une solution fractionnaire.
- La stratégie d'exploration est de parcourir l'arbre en largeur. Ainsi la structure adaptée pour gérer les sous-instances à traiter, les nœuds à explorer, est une file. Elle pourra être implémentée grâce au module Queue fourni par OCAML.

L'algorithme est alors le suivant.

Algorithme 1 : Algorithme Branch-and-Bound pour le problème KNAPSACK

Entrée : E une instance du problème KNAPSACK

Sortie : Une solution optimale pour E et sa valeur

```
1 meilleure_valeur ← 0;
2 meilleure_solution ← ∅;
3 todo ← {E};
4 tant que todo ≠ ∅ faire
5   | Soit  $I$  une instance que l'on ôte de todo;
6   | Calculer  $s_Q$  une solution du relâché fractionnaire de  $I$  et  $M$  sa valeur;
7   | si  $M > meilleure\_valeur$  alors
8   |   | Calculer  $s$  une solution de  $I$  et  $m$  sa valeur;
9   |   | si  $m > meilleure\_valeur$  alors
10  |   |   | meilleure_valeur ←  $m$ ;
11  |   |   | meilleure_solution ←  $s$ ;
12  |   | si  $M \geq m + 1$  alors
13  |   |   |  $I_1, I_2 \leftarrow \text{branchement}(I, s_Q)$ ;
14  |   |   | Ajouter  $I_1$  et  $I_2$  à todo
15 retourner (meilleure_solution, meilleure_valeur)
```

Q. 11 Définir une fonction `impose_i` : `masque -> int -> bool -> masque` prenant en arguments un masque, un objet et un booléen, et qui retourne un **nouveau** masque ♣ obtenu en ajoutant la contrainte que l'objet doit être présent si le booléen est `true`, et qu'il ne doit pas être présent sinon.

Q. 12 Définir une fonction `find_i_frac` : `qsolution -> int` prenant en argument une solution fractionnaire et retournant le premier objet pour laquelle la fraction indiquée n'est pas entière.

Q. 13 Définir une fonction `branch_and_bound` : `sad -> solution * int` implémentant l'algorithme précédent.

♣. il est donc nécessaire de copier le masque pris en argument

Opérateurs binaires à notation infixe en OCAML

En OCAML, les opérateurs arithmétiques usuels comme `+,*,/,mod` ...ou `+,*.,/. ...`, les opérateurs booléens `&&` et `||` ou encore les opérateurs de concaténation^a `@` et `^` sont en fait des fonctions de deux arguments qui ont la particularité de pouvoir être appelées selon la notation infixe, c'est-à-dire en étant placé entre leur deux arguments et non à gauche de ceux-ci. Afin de désigner de telles fonctions en OCAML, on encadre l'opérateur de parenthèses. Par exemple `(+)` est une expression de type fonctionnel `int -> int -> int`, alors que l'expression `+` n'est pas syntaxiquement correcte. De même :

- `(mod)` est de type `int -> int -> int`;
- `(+.)` est de type `float -> float -> float`;
- `(&&)` est de type `bool -> bool -> bool`;
- `(^)` est de type `string -> string -> string`;
- `(|>)` est de type `'a -> ('a -> 'b) -> 'b`;
- ...^b

De tels opérateurs ne sont pas réservés à la librairie standard, on peut définir de tels opérateurs à condition de choisir un nom de fonction valide. La description précise des noms valides est donnée dans la documentation : <https://v2.ocaml.org/manual/expr.html>, rubrique *Operators*. Sans être exhaustif, avec $\Sigma_1 = \{\$, \&, *, +, -, /, =, >, ^, |, \%, <\}$ et $\Sigma_2 = \Sigma_1 \cup \{\sim, !, ?, :, .\}$, les noms dans $\Sigma_1 \cdot \Sigma_2^*$ sont valides.

Exemple. On pourrait par exemple définir l'opérateur `%%` pour le produit scalaire sur les vecteurs d'entiers encodés par des tableaux comme suit.

```
1 exception ProduitScalaireImpossible
2 let (%%) (x: int array) (y: int array) : int =
3   if Array.length x = Array.length y
4   then
5     let res = ref 0 in
6     for i = 0 to Array.length x - 1 do
7       res := !res + x.(i) * y.(i)
8     done; !res
9   else raise ProduitScalaireImpossible
```

Après une telle définition le jeu de tests suivant est vérifié.

```
1 assert ([|0; 0; 0|] %% [|1; 2; 3|] = 0 );
2 assert ([|0; 1; 0|] %% [|1; 2; 3|] = 2 );
3 assert ([|1; 1; 1; 1|] %% [|1; 2; 3; 4|] = 10)
```

a. Attention le symbole `::` n'est pas un opérateur de concaténation, en effet c'est un constructeur du type `'a list` (même s'il a un statut particulier qui lui permet d'être placé entre ses paramètres).

b. Les seules exceptions sont `(*)` et `(*.)` qu'il faut écrire en laissant un espace entre la parenthèse et l'étoile car l'enchaînement `(*)` est réservé pour marquer un début de commentaire.