TP 15 : Algorithmes gloutons

On s'intéresse dans ce TP à un dernier problème d'ordonnancement, qui ressemble au problème "je suis en retard" du TD 5. Je vous propose aussi deux autres problèmes gloutons à la fin.

I. Ordonnancement avec échéance (SchedulingDeadline)

On considère un entier $n \geq 2$ et un ensemble de n tâches $T = \{t_0, \dots, t_{n-1}\}$. Dans la suite, on indexera toujours les tâches par $[\![0, n-1]\!]$. Chaque tâche t_i prend une unité de temps (par exemple exactement une seconde) pour être traitée sur une unité de calcul.
Chaque tâche t dispose d'une *échéance* $f(t) \in [\![1, n]\!]$ (appelée *deadline* en anglais), à laquelle la tâche t doit avoir été traitée, sans quoi on doit payer une certaine *pénalité* $p(t) \in \mathbb{N}$.
On appelle *stratégie d'ordonnancement* une fonction $d : T \to [\![0, n-1]\!]$ qui associe à chaque tâche $t \in T$ un unique temps de début $d(t)$. Évidemment, deux tâches différentes t_i et t_j doivent avoir un temps de début différent $d(t_i) \neq d(t_j)$.
Selon cette stratégie d, on en déduit une séparation de l'ensemble des tâches T en deux ensembles disjoints $T = T^+(d) \sqcup T^-(d)$:

— $T^+(d)$ est l'ensemble des tâches t traitées dans les délais : $t \in T^+(d)$ ssi $d(t) < f(t)$ (t est commencée au temps $d(t)$ donc finit en $d(t)+1$ qui doit être inférieure à sa *deadline* $f(t)$),

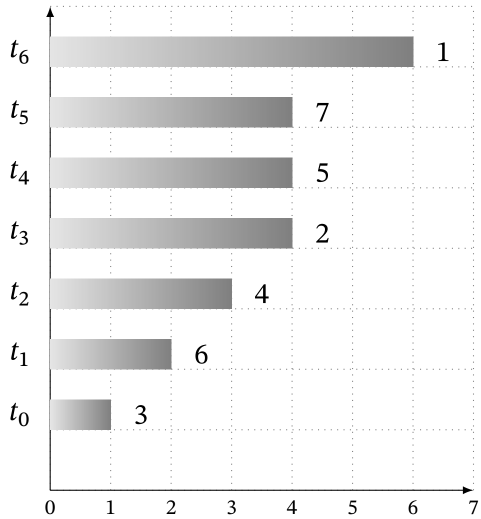
— $T^-(d)$ est l'ensemble complémentaire, autrement dit l'ensemble des tâches t traitées en retard : $t \in T^-(d)$ ssi $d(t) \geq f(t)$ (finie en retard, après sa *deadline*).

On note alors $P(d) = \sum_{t \in T^-(d)} p(t) \in \mathbb{N}$ la somme des pénalités des tâches en retard.

Exercice 1 – Exemple

On donne cet ensemble de tâches t_i, avec leur *deadline* $f(t_i)$ et leur pénalité $p(t_i)$.

t_i	0	1	2	3	4	5	6
$f(t_i)$	1	2	3	4	4	4	6
$p(t_i)$	3	6	4	2	5	7	1



Une stratégie d'ordonnancement est donnée dans le tableau suivant :

t_i	0	1	2	3	4	5	6
$d(t_i)$	**6**	0	1	**4**	3	2	5

▶ **Question 1** À quoi correspondent les tâches pour lesquelles $d(t_i)$ est noté en gras, dans ce tableau ?

▶ **Question 2** Calculer la pénalité totale $P(d)$ pour cette stratégie d'ordonnancement d.

Exercice 2 – Problème d'optimisation

Étant donné un ensemble de tâches T, et la donnée des *deadlines* $f(t_i)$ et des pénalités $p(t_i)$, on cherche à trouver une stratégie d'ordonnancement d, de valeur $P(d)$ *minimale*.

▶ **Question 1** Donner un exemple d'instance du problème telle que l'on puisse trouver une solution d qui ait une pénalité nulle.

▶ **Question 2** A contrario, donner un autre exemple d'instance telle que l'on ne puisse pas trouver de solution d avec une pénalité nulle. Quelle sera sa pénalité minimale ?

Ces deux autres exemples pourront être utiles pour tester l'implémentation, par la suite.

Exercice 3 – Résolution par force brute

Dans cette section on étudie la faisabilité d'une approche naïve d'exploration exhaustive.

▶ **Question 1** Calculer le nombre de solutions $d : T \to [\![0, n-1]\!]$ possibles, sachant que cette application doit ordonnancer chaque tâche sur un temps de début unique (et que $|T| = n$).

▶ **Question 2** En déduire une borne inférieure sur la complexité temporelle dans le pire cas de l'algorithme de recherche exhaustive suivant :

— On examine chaque ordonnancement possible d, pour lequel on calcule sa valeur $P(d)$, et on garde celui qui a la pénalité totale minimale.

— On renvoie ce dernier à la fin.

Exercice 4 – Un algorithme glouton

On peut commencer par remarquer un point intéressant : l'ordonnancement des tâches en retard (celles de $T^-(d)$) n'a aucune importance, et on peut donc se contenter de déterminer une stratégie d'ordonnancement d pour les tâches traitées dans les délais (celles de $T^+(d)$) et la compléter par n'importe quel ordonnancement des autres tâches. Autrement dit : quitte à être en retard, on s'en fiche d'à quel point.

On peut ainsi reformuler le problème : il faut déterminer un sous-ensemble de tâches $T^+ \subseteq T$ *pouvant être traitées dans les délais*, tel que $\sum_{t \in T^+} p(t)$ soit *maximale*. Alors, on aura bien $P(d)$ minimale.

On va résoudre maintenant ce problème de maximisation des pénalités de T^+ par l'algorithme glouton que voici :

— On commence en posant $T^+ = \varnothing$ et tous les temps de l'ensemble $[\![0, n-1]\!]$ sont marqués comme étant disponibles,

— On parcourt ensuite les n tâches, dans un certain ordre (à préciser par la suite) :

  ◦ Quand on considère la tâche t, s'il existe un temps i disponible, tel que $i < f(t)$, alors on marque comme indisponible le plus grand de ces temps possibles : $i_0 = \max\{i \in [\![0, n-1]\!], i < f(t) \text{ et } i \text{ disponible}\}$, et on rajoute alors la tâche t à l'ensemble T^+, en la commençant au temps i_0 (i.e., $d(t) := i_0$),

— À la fin, on place les tâches restantes aux temps disponibles (elles sont dans $T^- = T \setminus T^+$ et donc leur ordonnancement relatif n'a pas d'importance).

On peut en envisager plusieurs manières de trier ler tâches. Celui que nous choisirons ici sera par ordre *décroissant* des pénalités $p(t)$. En effet, il semble logique d'essayer de placer en premier dans T^+ les tâches ayant les plus fortes pénalités, car le problème a été réécrit comme cherchant à maximiser la somme des pénalités $\sum_{t \in T^+} p(t)$.

▶ **Question 1** Au brouillon, exécuter l'algorithme glouton précédent, avec cet ordre initial des tâches, pour l'exemple de la figure de la page précédente.

Exercice 5 – Implémentation (en C)

Pour représenter une instance du problème, on se propose de définir une structure, appelée `tache`, qui contient les champs suivants :

— `unsigned int id` : un identifiant de la tâche t, qui pourra par exemple être son numéro $1, \dots, n$ comme dans l'exemple étudié plus haut,

— `unsigned int date_limite` : la échéance (*deadline*) $f(t) \in \mathbb{N}$,

— `unsigned int penalite` : la pénalité $p(t) \in \mathbb{N}$,

— `int debut` qui sera son temps de début $d(t)$, initialement placé à `-1` tant que la tâche n'est pas ordonnancée, puis modifié (une seule fois) quand on a trouvé le temps i_0 à laquelle l'ordonnancer (ou un autre temps dans le deuxième cas de l'algorithme glouton, pour les tâches qui ne seront pas dans T^+).

▶ **Question 1** Définir en cette structure en C. On pourra ensuite écrire `typedef struct tache tache;` pour utiliser le type `tache` et plutôt que `struct tache` dans le code.

▶ **Question 2** Écrire une fonction de prototype `tache creer_tache(unsigned int id, unsigned int date_limite, unsigned int penalite)` qui crée un objet local (sur la pile et pas sur le tas, donc pas besoin de `malloc`) avec ces champs, et le renvoie. [a]

▶ **Question 3** Dans la fonction `main`, créer les tâches de l'exemple ci-dessus, c'est-à-dire les `tache0` à `tache6`.

▶ **Question 4** Que fait la ligne suivante, à compléter et recopier dans votre programme ?

```c
tache taches[7] = {tache0, ... , tache6};
```

a. On rappelle qu'il est possible de faire un `return e;` avec e un élément d'un type défini par `struct`, sans avoir besoin de passer par des pointeurs vers des structures.

Exercice 6 – Représentation des temps disponibles et indisponibles

Pour représenter la disponibilité des temps $[\![0, n-1]\!]$, on propose utiliser un tableau de booléens `indisponible`. Les temps seront tous initialement marqués à `false` (c'est-à-dire qu'il ne sont *pas indisponible* et donc disponibles).

Tri des tâches

On dispose en C, dans la librairie standard, de la fonction `qsort` (qui contrairement à ce que son nom laisse penser, n'est pas obligatoirement implémentée avec un tri rapide — *quick sort* en anglais). Elle s'utilise de la manière suivante :

```c
// Tri des activités par ordre décroissant des pénalités
qsort(taches, nb_taches, sizeof(tache), compare_taches);
```

Cet appel trie par ordre croissant le tableau `taches`, qui contient `nb_taches` objets, tous de taille `sizeof(tache)` en mémoire, et qui sont comparables grâce à une fonction de comparaison `compare_tache` que l'on doit avoir implémenté au préalable.
La fonction `compare_tache` a le même rôle que la fonction de comparaison `compare : 'a -> 'a -> int` en OCaml. Elle renvoie 0 si `t1 == t2`, un nombre strictement négatif (par exemple -1) si `t1 < t2` (selon le critère choisi), et un nombre strictement positif (par exemple $+1$) sinon. Son prototype doit être :

```c
int compare_taches(const void* t1, const void* t2)
```

Pour pouvoir utiliser la fonction `qsort` sur tout type d'objets, le type des paramètres `t1` et `t2` est `void*` (vous pouvez ignorer le qualificatif `const` qui indique que l'on s'engage à ne pas modifier `t1` et `t2` pendant la procédure de comparaison). Il sera donc nécessaire, dans la fonction `compare_tache` de transtyper (*cast* en anglais) les arguments `t1` et `t2` :

```c
int compare_taches(const void* t1, const void* t2) {
  tache* tache1 = (tache*)t1;
  tache* tache2 = (tache*)t2;
  // On peut maintenant utiliser `tache1` et `tache2` qui sont de type `tache*`
  ...
}
```

▶ **Question 1** Implémenter cette fonction de comparaison pour pouvoir ensuite trier les tâches par ordre de pénalité *décroissante*.

Algorithme glouton

On cherche maintenant à implémenter l'algorithme glouton à l'aide d'une fonction de prototype :

```c
void ordonnancement(tache* tab_taches, int nb_taches)
```

— Cette fonction n'aura rien à renvoyer : quand elle décide d'ordonnancer la tâche t au temps $d(t)$, elle le fait en modifiant son champ `debut`.

— A la fin, toutes les tâches doivent avoir reçu une valeur unique et différente de $[\![0, n-1]\!]$ dans leur champ `debut`.

— Attention à bien libérer la mémoire allouée sur le tas pendant l'algorithme (par exemple le tableau `indisponible` de n booléens).

▶ **Question 2** Implémenter cette fonction.

▶ **Question 3** Afficher le résultat trouvé sous la forme suivante : `Tid (f:deadline, p:penalite) @ debut`. Pour l'exemple ci-dessus, on obtiendrait :

```
T6 (f:4, p:7) @ 3
T2 (f:2, p:6) @ 1
T5 (f:4, p:5) @ 2
T3 (f:3, p:4) @ 0
T1 (f:1, p:3) @ 4
T4 (f:4, p:2) @ 6
T7 (f:6, p:1) @ 5
```

▶ **Question 4** Écrire une fonction de prototype `unsigned int penalite_totale(tache* tab_taches, int nb_taches)` qui calcule la pénalité totale des tâches en retard. Afficher, dans la fonction `main` la pénalité totale trouvée par l'algorithme glouton.

▶ **Question 5** Tester cette fonction sur le tableau de tâches de l'exemple précédent, avant et après l'appel à `ordonnancement`. On devrait trouver le résultat suivant :

```
Avant résolution, pénalité totale = 28.
Après résolution, pénalité totale = 5.
```

▶ **Question 6** Afficher la pénalité totale, à chaque étape de la boucle principale de l'algorithme glouton, pour la voir diminuer au fur et à mesure. Par exemple, on pourra obtenir :

Shell
```
La tâche T0 est placée en d = 3, et la pénalité totale = 21.
La tâche T1 est placée en d = 1, et la pénalité totale = 15.
La tâche T2 est placée en d = 2, et la pénalité totale = 10.
La tâche T3 est placée en d = 0, et la pénalité totale = 6.
La tâche T6 est placée en d = 5, et la pénalité totale = 5.
```

▶ **Question 7** Tester l'algorithme avec d'autres instances du problème.

▶ **Question 8** Quelle est la complexité temporelle totale de l'algorithme glouton, en supposant que l'appel à `qsort` sur un tableau de taille `n` correspondant à `nb_taches` tâches se fait en temps $\Theta(n\log(n))$?

Exercice 7 – Preuve d'optimalité

Démontrons que cet algorithme glouton renvoie effectivement un ensemble T^+ optimal.

Propriété 1

Avec le critère de tri par pénalités décroissantes, l'algorithme glouton renvoie un ordonnancement optimal.

Première étape : Tout d'abord, on montre qu'il existe une solution optimale compatible avec le premier choix de l'algorithme glouton.

▶ **Question 1** Soit T un ensemble de tâches, et $t \in T$ une des tâches de pénalité maximale. Montrer qu'il existe un ensemble T^+ de tâches pouvant être traitées dans les délais, maximal pour la somme de ses pénalités et tel que $t \in T^+$.

Deuxième étape : On montre ensuite qu'en enlevant le choix glouton, on obtient une solution optimale du sous-problème. Soit $T^+ \subseteq T$ un ensemble de tâches pouvant être traitées, maximal pour la somme des pénalités, et contenant une tâche $t \in T^+$ de pénalité maximale. Soit i l'instant auquel commence cette tâche t dans un ordonnancement de T^+. On pose $T' = T \setminus \{t\}$, avec des dates limites modifiées : $\forall t' \in T', d_{T'}(t') = d_T(t')$ si $d_T(t') \leq i$, ou $d_{T'}(t') = d_T(t') - 1$ sinon.

▶ **Question 2** Montrer que $T^+ \setminus \{t\}$ est maximal pour T'.

▶ **Question 3** Conclure.

II. Deux autres problèmes

Exercice 8 – Le bibliothécaire optimisant

Un ou une bibliothécaire souhaite ranger des collections de livres classées par auteur sur une longue étagère. On considère ainsi une suite $(a_1, \dots, a_n)$ de collections, données par la taille a_i de la collection i sur l'étagère, par exemple, en nombre de pages ou en centimètres.
Un rangement des livres consiste à ordonner les collections, de la première à la dernière, sur l'étagère. Plus formellement, il s'agit d'une permutation $\sigma \in \mathfrak{S}_n$, où $\sigma(i)$ donne le numéro de la collection numéro i dans l'étagère.
Pour trouver l'auteur d'un livre, comme on ne les trie pas par ordre alphabétique, il est nécessaire de parcourir linéairement l'étagère en partant de la première collection. Le *coût d'accès* à la k-ième collection est donc $\mathrm{cout}(k) = \sum_{i=1}^{k} a_{\sigma(i)}$.

Le *coût moyen d'accès* aux n collections est alors donné par $\frac{1}{n}\sum_{k=1}^{n} \text{cout}(k)$.

▶ **Question 1** Déterminer un algorithme glouton, permettant d'obtenir un rangement de coût minimal et déterminer sa complexité.

▶ **Question 2** Démontrer la validité de votre approche, c'est-à-dire que votre algorithme (glouton) renvoie bien une solution optimale.

Exercice 9 – Un genre de « Le compte et bon » (simplifié)

On considère le processus suivant : on part de l'entier 1, et à chaque étape on peut soit doubler la valeur de l'entier courant, soit lui ajouter 1. L'objectif est d'atteindre un entier cible donné $n \in \mathbb{N}^*$.

▶ **Question 1** Montrer qu'il est toujours possible d'atteindre n'importe quel entier cible $n \in \mathbb{N}^*$.

Par exemple, on peut atteindre 10 en quatre étapes, ainsi :

$$1 \xrightarrow{+1} 2 \xrightarrow{\times 2} 4 \xrightarrow{+1} 5 \xrightarrow{\times 2} 10$$

▶ **Question 2** Mettre au point un algorithme glouton permettant d'obtenir le nombre minimal d'étapes nécessaires pour atteindre un entier n. Analyser sa complexité et surtout, démontrer la validité de votre approche, c'est-à-dire que cet algorithme renvoie bien le nombre minimal d'étapes nécessaires.