Ordonnancement de graphes de tâches
(D'après XENS 2015)

1 Introduction

1.1 Présentation du problème et notations

Il est fréquent, en informatique comme dans la vie de tous les jours, que l'on ait un ensemble de tâches à réaliser, et que l'on se pose la question de la meilleure façon de réaliser l'ensemble de ces tâches, en particulier l'ordre dans lequel il convient de les traiter. Dans le domaine de l'informatique, on qualifie ce problème de *problème d'ordonnancement*.

On considère un ensemble T de n tâches à effectuer (chaque tâche devant être effectuée une seule fois). On supposera ces tâches numérotées de 0 à $n-1$, de sorte que T correspond à l'ensemble des n entiers de $[\![0\,..\,n-1]\!]$.

Pour effectuer ces tâches, on dispose d'un ensemble de p « processeurs ». À chaque instant $t \in \mathbb{N}$, chacun des processeurs peut effectuer dans son intégralité une (et une seule) tache $u \in \mathrm{T}$ quelconque.

Un *ordonnancement* est la donnée d'une fonction σ de $\mathrm{T} = [\![0\,..\,n-1]\!]$ dans $\mathbb{N}$ qui, à chaque tâche $u \in \mathrm{T}$ associe l'instant $t \in \mathbb{N}$ auquel cette tâche est effectuée par l'un des processeurs. L'ensemble S_t des tâches effectuées à un instant $t \in \mathbb{N}$ est défini par

$$\mathrm{S}_t = \left\{ u \in \mathrm{T} \;\middle|\; \sigma(u) = t \right\}$$

Puisque chaque processeur ne peut effectuer qu'une seule tâche à la fois, pour tout $t \in \mathbb{N}$, le cardinal $|\mathrm{S}_t|$ de chacun de ces ensembles S_t doit vérifier $|\mathrm{S}_t| \leqslant p$.

La *durée d'exécution totale* d'un ordonnancement σ, correspondant au premier instant $t \in \mathbb{N}$ pour lequel toutes les tâches ont été effectuées, est définie par

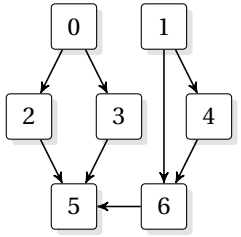
$$\max_{u \in \mathrm{T}} (\sigma(u) + 1).$$

Les tâches ne peuvent toutefois pas être exécutées dans n'importe quel ordre, il existe des *dépendances*. Une dépendance est la donnée d'un couple $(u, v) \in \mathrm{T}^2$ avec $u \neq v$. Elle signifie que la tâche v ne peut être entreprise que si la tâche u a déjà été effectuée. On dit alors que la tâche u *précède* la tâche v ou que la tâche v *succède* à la tâche u. L'ordonnancement σ doit satisfaire $\sigma(u) < \sigma(v)$.

On qualifie de *prédécesseurs* de u l'ensemble (potentiellement vide) des tâches qui précèdent la tâche u. De même, on qualifie de *successeurs* de u l'ensemble (potentiellement vide) des tâches qui succèdent à la tâche u. Une tâche qui n'a aucun prédécesseur s'appelle une *racine*. Une tâche qui n'a aucun successeur s'appelle une *feuille*.

On notera $\mathrm{D} \subseteq \mathrm{T} \times \mathrm{T}$ l'ensemble des dépendances à satisfaire. L'ensemble des tâches et leurs dépendances peut être vu comme un graphe orienté $\mathrm{G} = (\mathrm{T}, \mathrm{D})$. Les tâches à effectuer correspondent aux sommets du graphe, les dépendances à des arcs liant ces sommets.

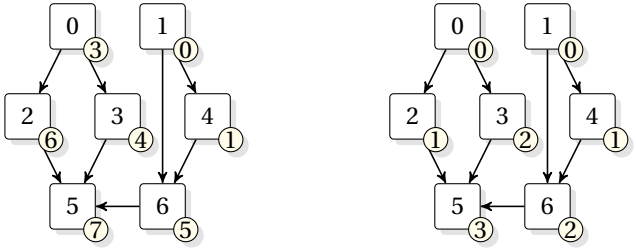
Voici un exemple de graphe décrivant un ensemble de sept tâches avec huit dépendances :



$\mathrm{T} = [\![0\,..\,6]\!]$

$\mathrm{D} = \left\{ (0,2), (0,3), (1,4), (2,5), (3,5), (1,6), (4,6), (6,5) \right\}$

Deux ordonnancements possibles pour le graphe précédent, respectivement pour $p = 1$ et $p = 2$ processeurs, sont illustrés ci-dessous ($\sigma(u)$ étant indiqué pour chaque tâche u) :



Les durées d'exécution totale de ces deux ordonnancements sont respectivement 8 et 4.

On dit qu'un ordonnancement est *optimal* pour un graphe de tâches G avec p processeurs si sa durée d'exécution est minimale parmi tous les ordonnancements possibles de G avec p processeurs. Dans les exemples ci-dessus, l'ordonnancement de droite est optimal, mais celui de gauche ne l'est pas (aucune tâche n'est effectuée pour $t = 2$, il est aisé de trouver un ordonnancement de durée d'exécution totale égal à 7).

L'objectif de ce problème va être de déterminer des ordonnancements optimaux pour certaines classes de graphes de tâches pour un nombre p donné de processeurs.

1.2 Implémentation

En OCaml, on représentera un graphe de n tâches $\mathrm{G} = (\mathrm{T}, \mathrm{D})$ par un tableau de longueur n contenant dans la case d'index `i` la liste des successeurs de la tâche `i` (l'ordre n'est pas

spécifié, mais les listes ne contiennent pas de doublon).

```
type graphe = int list array;;
```

Ainsi, le graphe précédent peut être décrit par :

```
let gr = [| [2;3]; [6;4]; [5]; [5]; [6]; []; [5] |];;
```

Un ordonnancement σ sera représenté par un tableau d'entiers positifs de longueur n, tel que dans la case d'index `i` on retrouve $\sigma(i)$.

```
type ordonnancement = int array;;
```

L'ordonnancement optimal proposé précédemment dans le cas $p = 2$ processeurs s'écrirait donc :

```
let ord = [| 0; 0; 1; 2; 1; 3; 2 |]
```

1.3 Rappels

On rappelle que l'on dispose en OCaml notamment des fonctions suivantes :

- `List.length` (`'a list -> int`) retourne la longueur de la liste passée en argument ;
- `Array.length` (`'a array -> int`) retourne la taille du tableau passé en argument ;
- `Array.make` (`int -> 'a -> 'a array`) prend en argument un entier `n` et un élément quelconque et retourne un tableau de `n` cases contenant toutes cet élément.

2 Opérations et résultats élémentaires

1. Proposer une fonction `transpose` de signature `graphe -> graphe` qui prend un graphe de tâche $G = (T, D)$ et renvoie un graphe de tâches $G' = (T, D')$ où les arcs ont été retournés, c'est-à-dire tel que pour toutes paire de tâches u et v de T vérifient l'équivalence $(u, v) \in D' \iff (v, u) \in D$.

2. Que représente, pour G, la liste dans la i^e case du tableau obtenu en appliquant la fonction fonction `transpose` à un tableau décrivant un graphe de tâches $G = (T, D)$?

3. Montrer que s'il existe un chemin $u \rightsquigarrow v$ dans $G = (T, D)$ menant de $u \in T$ à $v \in T$, alors nécessairement $\sigma(u) < \sigma(v)$.

4. En déduire que s'il existe un ordonnancement pour G, alors G est acyclique.

5. Montrer que si G est acyclique, alors G possède nécessairement au moins une racine et au moins une feuille.

Dans la suite du problème, on supposera les graphes de tâches G acycliques. Nous allons montrer, par construction, qu'il existe toujours un ordonnancement.

3 Ordonnancement par hauteur

3.1 Étiquetage par hauteur depuis les racines

On considère un graphe acyclique $G = (T, D)$ de $n \geqslant 1$ tâches, et on s'intéresse donc à la recherche d'un ordonnancement σ pour un ensemble de p processeurs.

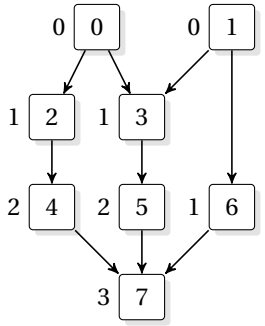
Une tâche peut être ordonnancée seulement si toutes les tâches dont elle dépend l'ont déjà été. Trouver un ordonnancement d'un graphe G acyclique avec un seul processeur revient donc à énumérer les tâches de ce graphe dans un ordre (total) qui respecte les contraintes de dépendance. Nous allons poser les bases d'un tel ordre.

Pour y parvenir, on va associer à chaque tâche $u \in T$ une *étiquette* $\text{tag}(u) \in \mathbb{N}$. Cet étiquetage a la propriété suivante : une tâche v reçoit une étiquette $\text{tag}(v)$ portant un numéro strictement supérieur à celui de toutes les étiquettes $\text{tag}(u)$ des tâches u prédécesseurs de v dans le graphe de tâches G. Notez que plusieurs tâches peuvent recevoir la même étiquette. Pour ce faire, on applique l'algorithme suivant :

Algorithme 1 : (étiquetage par hauteur depuis les racines).

1. Initialement, aucune tâche n'est étiquetée.
2. À l'itération d'ordre $k = 0$, on parcourt l'ensemble des tâches et on affecte l'étiquette 0 aux tâches racines.
3. À l'itération $k > 0$, on parcourt l'ensemble des tâches en repérant toutes les tâches qui n'ont pas encore été étiquetées mais dont toutes les tâches prédécesseurs sont déjà étiquetées. On affecte ensuite à chacune de ces tâches l'étiquette k.
4. L'algorithme termine quand toutes les tâches sont étiquetées.

La figure ci-dessous présente un exemple d'étiquetage selon l'algorithme 1. Les étiquettes sont indiquées à gauche des tâches.



6. Proposer une fonction `nb_dependances` de signature `graphe -> int array` prenant

en argument un graphe de n tâches et retournant un tableau de longueur n indiquant, pour chaque tâche i, le nombre de ses prédecesseurs dans la i^e case.

7. Écrire une fonction `racines` de signature `graphe -> int list` prenant en argument un graphe de tâches et retournant la liste des tâches qui sont des racines du graphe.

8. En déduire une fonction `etiquete` de signature `graphe -> int array` prenant en argument un graphe de n tâches et retournant un tableau d'entiers de longueur n contenant les étiquettes de chaque tâche (la case d'index `i` contenant tag(i)).

9. Quelle est sa complexité en fonction de n et du nombre |D| de dépendances?

Soit $u \in T$ une tâche de G. On note P_u l'ensemble des chemins menant à u de la forme $u_0 \mapsto u_1 \mapsto u_2 \mapsto \ldots \mapsto u_{n-1} \mapsto u$ où u_0 est une racine de G. On dit que u admet un *chemin critique amont* si P_u est non-vide et si l'ensemble des longueurs des chemins de P_u est majoré. Les *chemins critiques amont* de u sont alors les chemins de P_u de plus grande longueur. Le chemin critique amont d'une racine est de longueur nulle et il est unique.

10. On suppose le graphe G = (T, D) acyclique. Montrer que toutes les tâches $u \in T$ de G admettent des chemins critiques amonts.

11. Démontrez que, pour un graphe de tâches G = (T, D) acyclique, une tâche u reçoit l'étiquette de valeur k dans l'algorithme 1 si et seulement si la longueur commune des chemins critiques amont de u est k.

12. En déduire que l'algorithme termine si G est acyclique. Que se passe-t-il si G n'est pas acyclique?

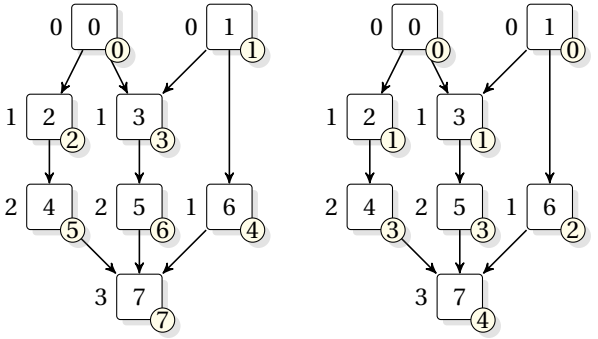
13. Démontrez que si une tâche u porte une étiquette k, alors $\sigma(u) \geqslant k$, quel que soit l'ordonnancement mais aussi quel que soit le nombre de processeurs utilisés.

3.2 Construction de l'ordonnancement

Une fois un graphe de tâches G = (T, D) étiqueté selon l'algorithme 1, il est possible de déterminer un ordonnancement avec p processeurs en exécutant les tâches par niveau selon la valeur de leurs étiquettes. Notons $k_{\max}$ la plus grande des étiquettes attribuées par l'algorithme 1.

> **Algorithme 2 :** (algorithme d'ordonnancement par hauteur pour p processeurs).
>
> Pour chaque valeur de k entre 0 et $k_{\max}$, on exécute les tâches portant les étiquettes k par lot de p tâches. Pour chaque valeur k, le dernier lot pourra être incomplet. Les processeurs inutilisés sont alors inactifs.

La figure ci-après présente deux ordonnancements obtenus pour le graphe précédemment étiqueté, respectivement pour $p = 1$ et $p = 2$ processeurs. On notera en particulier que dans le second cas, $\sigma(4) = 3$ et non 2. Précisons également que ces ordonnancements ne sont pas uniques car ils dépendent de la constitution des lots pour chaque k.



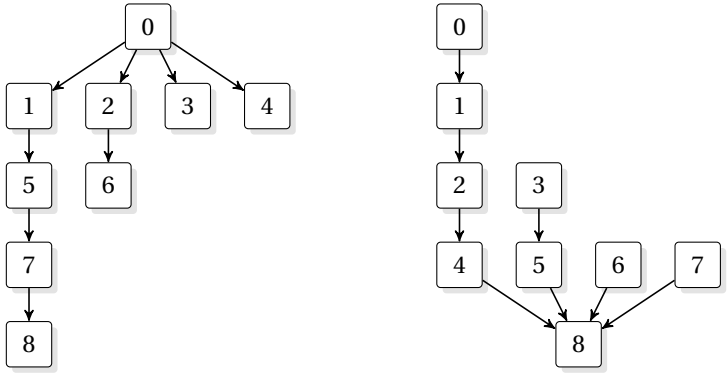
14. Proposer une fonction `ordonne` de signature `graphe -> int -> ordonnancement` prenant en argument un graphe de n tâches G et un nombre de processeurs p et retourne un ordonnancement σ (tableau de n entiers) en utilisant l'algorithme 2.

15. Estimer la complexité de la fonction précédente (en fonction par exemple du nombre de tâches n, du nombre de dépendances |D|, du nombre de processeurs p et de $k_{\max}$).

16. Montrer que l'ordonnancement obtenu est optimal dans le cas où $p = 1$.

3.3 Limitations

Un *graphe arborescent sortant* est un graphe de tâches avec une unique racine et dans lequel chaque tâche à l'exception de la racine a un unique prédécesseur. Un *graphe arborescent entrant* est un graphe de tâches avec une unique feuille et dans lequel chaque tâche à l'exception de la feuille a un unique successeur. Le graphe de gauche ci-dessous est un graphe arborescent sortant, celui de droite un graphe arborescent entrant.



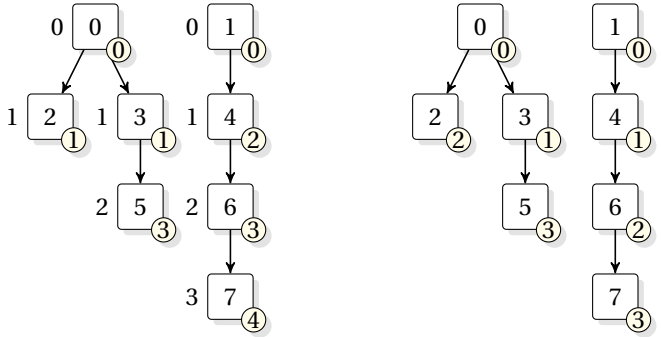
17. Déterminer un ordonnancement obtenu avec l'algorithme 2 pour les deux graphes ci-dessus dans le cas $p = 2$ processeurs.

18. Dans chacun des cas, préciser si ces ordonnancements sont optimaux.

4 Algorithme de Hu

L'étiquetage par hauteur ne fournit pas assez d'informations pour ordonnancer les tâches de manière optimale car il s'appuie uniquement sur la structure du graphe en amont des tâches étiquetées.

La figure ci-dessous décrit par exemple deux ordonnancements du même graphe pour $p = 2$ processeurs dans lequel les tâches exécutées sont exécutées dans l'ordre croissant des étiquettes de hauteur, celui de gauche ayant été obtenu au moyen de la méthode précédente. On remarque que l'ordonnancement de gauche conduit l'un des processeurs à rester inactif alors que l'ordonnancement de droite permet l'utilisation constante des deux processeurs, et conduit à une durée d'exécution totale inférieure.

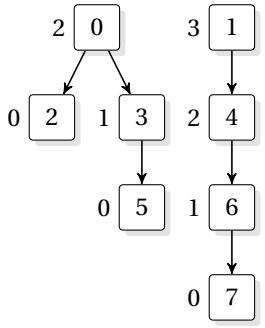


L'idée de cette partie est de mettre en place un autre étiquetage, cette fois-ci fondé sur les plus longs chemins de tâches en aval. En effet, nous montrerons que la longueur des plus longs chemins de tâches en aval d'une tâche limite inférieurement la durée d'exécution au-delà de cette tâche. Il sera donc intéressant d'exécuter les tâches avec les plus longs chemins de tâches en aval le plus tôt possible. C'est ce que nous ferons dans l'algorithme 4 ci-dessous dû à Hu. Il suffit donc d'adapter l'algorithme 1 pour étiqueter les tâches à partir des feuilles au lieu des racines.

Algorithme 3 : (étiquetage par profondeur depuis les feuilles)

1. Initialement, aucune tâche n'est étiquetée.
2. À l'itération d'ordre $k = 0$, on parcourt l'ensemble des tâches et on affecte l'étiquette 0 aux tâches feuilles.
3. À l'itération $k > 0$, on parcourt l'ensemble des tâches en repérant toutes les tâches qui n'ont pas encore été étiquetées mais dont toutes les tâches successeurs sont déjà étiquetées. On affecte ensuite à chacune de ces tâches l'étiquette k.
4. L'algorithme termine quand toutes les tâches sont étiquetées.

Ainsi, l'étiquetage obtenu avec l'algorithme 3 pour le graphe précédent sera :



19. En utilisant intelligemment les fonctions déjà écrites, proposer une fonction `etiquete_bis` simple de signature `graphe -> int array` prenant en argument un graphe de n tâches et retournant un tableau d'entiers de longueur n contenant les étiquettes de chaque tâche selon la méthode de l'algorithme 3.

Soit $u \in T$ une tâche de G. On note P'_u l'ensemble des chemins partant de u de la forme $u \mapsto u_1 \mapsto u_2 \mapsto \ldots \mapsto u_n$ où u_n est une feuille de G. On dit que u admet un *chemin critique aval* si P'_u est non-vide et si l'ensemble des longueurs des chemins de P'_u est majoré. Les *chemins critiques aval* de u sont alors les chemins de P'_u de plus grande longueur.

On rappelle que G est acyclique. Comme pour les chemins critiques amont, toutes les tâches u de G admettent des chemins critiques aval. On appelle *profondeur* $\text{depth}(u)$ d'une tâche u la longueur commune des chemins critiques aval de u.

20. Soit u une tâche exécutée à l'instant t. Montrer que, quel que soit le nombre de processeurs, utilisés, la durée d'exécution sera strictement supérieure à $t + \text{depth}(u)$.

Une tâche est dite *prête* à être exécutée à un instant t si toutes les tâches dont elle dépend ont été déjà exécutées.

Algorithme 4 : (algorithme de Hu pour p processeurs).

Soit G un graphe de tâches acyclique. On construit un ordonnancement de G pour p processeurs de manière suivante.

1. À chaque instant $t \geqslant 0$, on considère l'ensemble R des tâches de G prêtes à être exécutées. Soit r le cardinal de cet ensemble.
2. Si $r \leqslant p$, on choisit pour être exécutées à l'instant t les r tâches de R et $p - r$ processeurs restent inactifs. Sinon, on choisit pour être exécutées à l'instant t parmi les tâches de R les p tâches ayant la plus grande profondeur.
3. L'ordonnancement se termine quand toutes les tâches de G ont été exécutées.

21. Appliquer l'algorithme de Hu aux deux graphes arborescents de la question 16 pour

$p = 2$ processeurs, et préciser les durées d'exécution totales correspondantes.

On peut montrer que l'algorithme de Hu est optimal pour les graphes de tâches arborescents entrants avec un nombre de processeurs p arbitraire. La preuve de ce résultat est délicate, mais l'une des clés de la preuve est la propriété suivante.

22. Soit G un graphe de tâches arborescent entrant avec p processeurs. Montrez que dans l'algorithme de Hu le cardinal de l'ensemble R de tâches prêtes dans G ne peut pas croître au cours de l'algorithme.

Cette garantie d'optimalité est spécifique du caractère arborescent entrant, et l'algorithme de Hu ne donne pas un ordonnancement optimal pour tous les graphes.

23. Montrer que l'algorithme de Hu ne conduit pas nécessairement à un ordonnancement optimal en prenant pour exemple le graphe de tâches ci-dessous avec $p = 2$ processeurs.