**ÉCOLE POLYTECHNIQUE – ÉCOLES NORMALES SUPÉRIEURES**

**CONCOURS D'ADMISSION 2015** **FILIÈRE MP — SPÉCIALITÉ INFO**

**COMPOSITION D'INFORMATIQUE – A – (XULCR)**

(Durée : 4 heures)

L'utilisation des calculatrices **n'est pas autorisée** pour cette épreuve.
Le langage de programmation pour cette épreuve est Caml Light.

⋆ ⋆ ⋆

# Ordonnancement de graphes de tâches

## Introduction

Voici un problème important qui se pose à chacun et chacune de nous tous les jours ! J'ai un certain nombre de tâches à *exécuter* aujourd'hui ; comment planifier à quelle heure chacune va être exécutée ? Par exemple, je peux avoir aujourd'hui à terminer un devoir en mathématiques, faire ma lessive à la laverie, avancer un projet en informatique, aller faire quelques courses et repasser mon linge. Chacune de ces tâches va me prendre une certaine durée, que je peux estimer. Même si ce n'est pas tout à fait le cas dans la vie courante, on peut ici supposer que les tâches ne sont pas interrompues ; je ne commence une nouvelle tâche que lorsque la tâche en cours est terminée.

**Dépendances.** Le plus souvent, il existe des *dépendances* entre les tâches, en ce sens qu'il est nécessaire d'exécuter une tâche $A$ avant d'exécuter une tâche $B$. Par exemple, il est nécessaire de laver le linge avant de le repasser. Mais, si je n'ai plus de lessive, il est nécessaire de passer faire les courses avant d'aller à la laverie. Ces dépendances peuvent être modélisées par un graphe orienté. Les nœuds sont les tâches, les arcs orientés sont les dépendances. Il y a un arc $A \to B$ si la tâche $A$ doit être terminée avant que la tâche $B$ puisse commencer. Notez qu'il n'est pas du tout nécessaire de passer reprendre son linge à la laverie dès que la machine s'arrête. La tâche $B$ peut être ordonnancée longtemps après que la tâche $A$ sera terminée.

**Ordonnancement.** Un *ordonnancement* est la donnée d'une heure d'exécution pour chacune des tâches qui respecte cette contrainte qu'une tâche commence seulement lorsque toutes celles dont elle dépend sont terminées. Notez qu'il peut y avoir des moments de repos où aucune tâche n'est en exécution. La mesure intéressante est alors la *durée d'exécution totale*, c'est-à-dire la durée écoulée entre l'heure à laquelle l'exécution de la première tâche commence et l'heure à laquelle celle de la dernière tâche se termine.

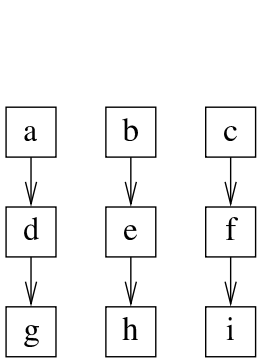
**Parallélisme.** Si je suis seul, je ne peux exécuter qu'une tâche à la fois. Mais je peux aussi me faire aider ! Il est alors possible d'exécuter plusieurs tâches en *parallèle*. Par exemple, je peux demander à quelqu'un de faire ma lessive à la laverie, pour aller faire mes courses pendant ce temps et ensuite revenir la prendre pour la repasser. En termes informatiques, on parle de multiples *processeurs* qui collaborent pour le traitement des tâches. Dans notre exemple, deux processeurs travaillent en parallèle : l'un pour faire la lessive, l'autre pour faire les courses. À chaque instant, plusieurs tâches peuvent être exécutées, au plus autant que de processeurs. Notez qu'il est cependant possible qu'un processeur soit forcé de rester inactif un certain temps, par exemple parce que la tâche qu'il doit exécuter dépend d'une autre tâche qui est en cours d'exécution sur un autre processeur.

**Défi algorithmique.** Les exemples ci-dessus sont bien sûr des illustrations simplifiées. En pratique, on va considérer des graphes de tâches de taille gigantesque, par exemple l'ensemble des actions nécessaires pour assembler un avion. Ces graphes comptent des millions de tâches avec des dépendances complexes. Les tâches seront allouées à des ouvriers. Ceux-ci travaillent à l'intérieur d'horaires fixés, avec des périodes de repos, des vacances, des arrêts imprévus pour cause de maladie ou de panne de machine. L'objectif sera de trouver le meilleur ordonnancement possible pour l'assemblage selon une mesure donnée. Notez que dans ce cas, le graphe est fixe, mais le nombre d'ouvriers peut varier : on peut par exemple embaucher plus d'ouvriers pour réduire la durée d'exécution totale. Cela augmente le coût de production mais réduit les délais de livraison. Trouver un ordonnancement optimal est alors un défi algorithmique majeur.

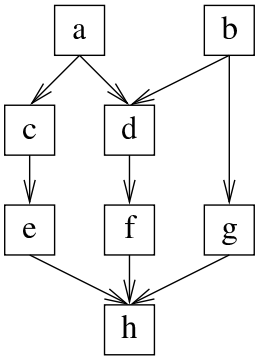
**Plan du sujet proposé.** La partie I introduit la notion d'ordonnancement d'un graphe de tâches. La partie II s'intéresse à quelques propriétés des graphes de tâches acycliques. La partie III étudie une première approche pour la recherche d'un ordonnancement d'un graphe de tâches. L'ordonnancement produit est optimal avec $p = 1$ processeur, mais pas avec $p > 1$. La partie IV étudie comment modifier cette approche pour produire un ordonnancement optimal avec $p = 2$ processeurs dans le cas particulier des arbres arborescents entrants. La partie V décrit comment compléter cette approche et obtenir un ordonnancement optimal avec $p = 2$ processeurs dans le cas général.

Les parties peuvent être traitées indépendamment. Néanmoins, chaque partie utilise des notations et des fonctions introduites dans les parties précédentes.

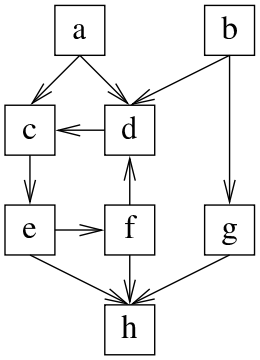
La complexité, ou le coût, d'un algorithme ou d'une fonction Caml est le nombre d'opérations élémentaires nécessaires à son exécution dans le pire cas. La notion d'opération élémentaire sera précisée dans chaque cas par le sujet. Lorsque cette complexité dépend d'un ensemble de




(a) Un graphe de tâches tout simple.




(b) Un graphe de tâches plus complexe sans cycle.




(c) Un graphe de tâches avec un cycle : c, e, f, d.

FIGURE 1 – Quelques exemples de graphes de tâches.

paramètres $(n, p, \dots)$, on pourra donner cette estimation sous forme asymptotique. On rappelle qu'une application $c(n, p, \dots)$ est dans la classe $\mathcal{O}(f)$ s'il existe une constante $\alpha > 0$ telle que $|c(n, p, \dots)| < \alpha \times f(n, p, \dots)$, pour toutes les valeurs de $n, p, \dots$ assez grandes.

# I Définitions de base

**Graphe de tâches.** Un *graphe de tâches* $G = (T, D, \delta)$ est constitué d'un ensemble *fini* et *non vide* $T$ de tâches notées $u$, $v$, etc.

L'application $\delta$ associe à chaque tâche du graphe sa durée. Dans tout ce problème, on supposera que la durée d'une tâche est unitaire : $\delta(u) = 1$.

L'ensemble $D \subseteq T \times T$ est un ensemble d'arcs (dirigés) entre ces tâches ($D$ pour *dépendance*, voir plus loin). L'existence d'un arc $(u, v)$ dans $D$ est notée $u \to v$. On suppose qu'il n'y a pas d'arc entre une tâche et elle-même : $D \cap \{(u, u) | u \in T\} = \emptyset$.

Il y a un arc dirigé $(u, v)$ d'une tâche $u$ vers une autre tâche $v$ distincte, $u \neq v$, si la tâche $u$ doit être exécutée avant la tâche $v$. On dit alors que la tâche $v$ *dépend* de la tâche $u$ en ce sens que $v$ ne peut commencer qu'une fois que $u$ est terminée. On dit alors que $u$ *précède* $v$ ou que la tâche $v$ *succède* à la tâche $u$. On peut donc parler de l'ensemble des tâches qui précèdent $v$ et de l'ensemble des tâches qui succèdent à $u$. Notez que ces ensembles peuvent être vides.

Une tâche qui n'a aucun prédécesseur s'appelle une *racine*. Une tâche qui n'a aucun successeur s'appelle une *feuille*.

La *taille* d'un graphe de tâches est le nombre de ses tâches.

La figure 1 propose quelques exemples de graphes de tâches de petite taille.

**Ordonnancement.** Un *ordonnancement* $\sigma$ d'un graphe de tâches $G = (T, D, \delta)$ est une application $\sigma$ à valeurs entières qui associe une date d'exécution à chaque tâche du graphe, dans le respect des contraintes de dépendance spécifiées par la formule (1) ci-dessous.

Une tâche $u \in T$ est exécutée de manière ininterrompue par le processeur qui en a la charge aux instants $t$ tels que $\sigma(u) \leq t < \sigma(u)+\delta(u)$. Une tâche $v$ qui dépend de $u$ ne peut être exécutée qu'après la terminaison de $u$, donc à partir de l'instant $\sigma(u) + \delta(u)$. Un ordonnancement doit donc respecter la contrainte suivante :

$$\forall u, v \in T,\ (u \to v) \Rightarrow (\sigma(u) + \delta(u) \leq \sigma(v)) \tag{1}$$

L'instant $a_\sigma$ du début de l'ordonnancement $\sigma$ du graphe de tâches $G = (T, D, \delta)$ est l'instant où la première tâche est exécutée : $a_\sigma = \min_{u \in T} \sigma(u)$.

L'instant $b_\sigma$ de la fin de l'ordonnancement est l'instant où la dernière tâche est terminée : $b_\sigma = \max_{u \in T} (\sigma(u) + \delta(u))$.

La *durée d'exécution totale* de l'ordonnancement $\sigma$ est $b_\sigma - a_\sigma$.

L'ensemble $S_t$ des tâches en cours d'exécution à l'instant $t$ est défini par :

$$S_t = \{u \mid \sigma(u) \leq t < \sigma(u) + \delta(u)\}$$

Dans notre cas, $\delta(u) = 1$ pour toute tâche $u$ et cette formule se simplifie :

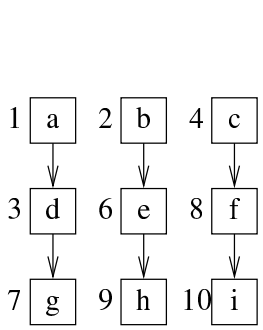
$$S_t = \{u \mid \sigma(u) = t\}$$

On dit qu'un ordonnancement utilise (au plus) $p$ processeurs si $\mathsf{cardinal}(S_t) \leq p$ à tout instant $t$.

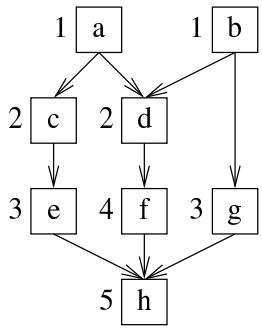
On dit qu'un ordonnancement est optimal pour un graphe de tâches $G$ avec $p$ processeurs si sa durée d'exécution est minimale parmi tous les ordonnancements possibles de $G$ avec $p$ processeurs.

La figure 2 propose quelques exemples d'ordonnancements de graphes de tâches. On rappelle que chaque tâche $u$ dure une unité de temps : $\delta(u) = 1$. La date d'exécution de chaque tâche est indiquée à gauche de cette tâche.

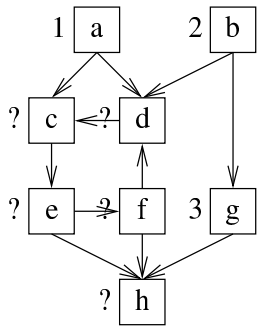
1. L'ordonnancement présenté en figure 2a a une durée d'exécution totale de 10 avec $p = 1$ processeur. Il n'est pas optimal. En effet, aucune tâche n'est exécutée à l'instant 5 où la tâche $e$ est prête ; il serait donc possible de réduire la durée d'exécution totale à 9. Ce serait alors optimal puisqu'il y a 9 tâches, chacune de durée 1.
2. L'ordonnancement présenté en figure 2b a une durée d'exécution totale de 5 avec $p = 2$ processeurs. Comme il y a 8 tâches pour 2 processeurs, tout ordonnancement a une durée au moins égale à 4. Cependant, les contraintes de dépendance ne permettent pas de réaliser un ordonnancement de durée 4. Un ordonnancement de durée d'exécution totale 5 est donc optimal.
3. Le graphe de tâches de la figure 2c comporte un cycle : $c \to e \to f \to d$. Aucune tâche d'un cycle ne peut être exécutée en respectant les contraintes de dépendance.




(a) Un ordonnancement pour le graphe de tâches de la figure 1a avec $p = 1$ processeur.




(b) Un ordonnancement pour le graphe de tâches de la figure 1b avec $p = 2$ processeurs.




(c) Un graphe de tâches avec un cycle n'admet pas d'ordonnancement.

FIGURE 2 – Quelques exemples d'ordonnancement de graphes de tâches.

**Objectif.** 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.

# II Graphe de tâches acyclique

Un *chemin* de dépendance dans un graphe de tâches d'une tâche $u$ à une tâche $v$ est une suite de tâches $(u_0, u_1, \ldots, u_n)$, $n \geq 0$, avec $u_0 = u$ et $u_n = v$ et en dépendance successive : $u_0 \to u_1$, $u_1 \to u_2$, …, $u_{n-1} \to u_n$.

La *longueur* du chemin est le nombre d'arcs de dépendance, c'est-à-dire l'entier $n$. La tâche initiale du chemin est $u_0$, sa tâche terminale $u_n$. Notez qu'une tâche peut apparaître plusieurs fois dans un chemin. Notez aussi qu'un chemin peut être de longueur nulle. Il a alors la même tâche initiale et terminale.

Une tâche $u$ est *atteignable* depuis une tâche $u_0$ s'il existe un chemin qui a pour tâche initiale $u_0$ et pour tâche terminale $u$.

Un *cycle* dans un graphe de tâches est un chemin $(u_0, u_1, \ldots, u_n)$ de longueur $n > 0$ qui a même tâche initiale et terminale : $u_0 = u_n$.

Un graphe de tâches est dit *acyclique* s'il ne possède pas de cycle. Les graphes de tâches des figures 1a et 1b sont acycliques, celui de la figure 1c possède un cycle.

On veut maintenant montrer qu'il n'existe pas d'ordonnancement pour un graphe de tâches qui possède un cycle.

**Question 1.** *Soit $G = (T, D, \delta)$ un graphe de tâches qui admet un ordonnancement $\sigma$. Démontrez que s'il existe un chemin de dépendance de longueur non nulle d'une tâche $u$ vers une tâche $v$ dans $G$, alors $\sigma(u) < \sigma(v)$. En déduire que $G$ est nécessairement acyclique.*

*On pourra procéder par récurrence sur la longueur $n$ du chemin après avoir soigneusement spécifié la propriété $H(n)$ démontrée par récurrence.*

```
let graph = make_graph ();;

let a = make_task 1 "a";;
let b = make_task 1 "b";;
let c = make_task 1 "c";;
let d = make_task 1 "d";;
let e = make_task 1 "e";;
let f = make_task 1 "f";;
let g = make_task 1 "g";;
let h = make_task 1 "h";;
```

```
add_task a graph;;
add_task b graph;;
add_task c graph;;
add_task d graph;;
add_task e graph;;
add_task f graph;;
add_task g graph;;
add_task h graph;;
```

```
add_dependence a c graph;;
add_dependence a d graph;;
add_dependence b d graph;;
add_dependence b g graph;;
add_dependence c e graph;;
add_dependence d f graph;;
add_dependence e h graph;;
add_dependence f h graph;;
add_dependence g h graph;;
```

TABLE 1 – Comment construire le graphe de la figure 1b avec la bibliothèque Graph.

Avant de chercher un ordonnancement d'un graphe de tâches, il faut donc vérifier qu'il est acyclique. La propriété suivante fournit une condition nécessaire. On rappelle qu'un graphe de tâches est constitué d'un nombre fini et non nul de tâches.

**Question 2.** *Soit $G = (T, D, \delta)$ un graphe de tâches. Montrez que si $G$ est acyclique, alors $G$ a nécessairement au moins une racine et une feuille.*

Dans toute la suite du problème, on suppose qu'on dispose d'une certaine bibliothèque Caml appelée Graph qui permet de manipuler des graphes de tâches. Cette bibliothèque regroupe des fonctions qui sont disponibles pour les utilisateurs, même s'ils n'en connaissent pas le code. On pourra donc utiliser librement toutes les fonctions de cette bibliothèque sans les réécrire. Cette bibliothèque définit deux types de données : graph pour les graphes de tâches, et task pour les tâches. Elle propose les fonctions listées en table 2 pour manipuler ces deux types. La table 1 montre comment on pourrait construire le graphe de la figure 1b avec ces fonctions.

On rappelle quelques fonctions Caml Light permettant de manipuler les tableaux et d'imprimer.

- make_vect n a renvoie un nouveau tableau de n éléments qui contiennent tous la même valeur a.
- vect_length tab renvoie la longueur (le nombre d'éléments) $n$ du tableau tab. Ceux-ci sont indexés de 0 à $n - 1$ inclus.
- tab.(i) renvoie la valeur de l'élément d'indice i du tableau tab.
- tab.(i) <− a affecte la valeur a à l'élément d'indice i du tableau tab.
- sub_vect tab i n renvoie le sous-tableau de tab constitué de n éléments à partir de l'indice i inclus.
- print_int n imprime l'entier n.
- print_string s imprime la chaîne s.
- print_newline () passe à la ligne suivante.

Voici par exemple comment imprimer l'ensemble des successeurs d'une tâche :

```
let print_successors t =
  let tab = get_successors t in
  for i = 0 to (vect_length tab) − 1 do
    print_string (get_name (tab.(i))); print_string "␣"
  done;
  print_newline ();;
```

| | |
|---|---|
| make_graph: unit −> graph | Crée un graphe vide. |
| make_task: int −> string −> task | Crée une tâche d'une durée donnée avec un nom donné.<br>(Attention, une erreur se produit si le nom a déjà été utilisé pour la création d'une autre tâche.) |
| empty_task: task | Une tâche vide, différente de toutes les tâches créées par la fonction make_task.<br>(Attention, une erreur se produit si on applique les fonctions de manipulation de tâches ci-dessous autres que is_empty_task à cette tâche.) |
| is_empty_task: task −> bool | Teste si une tâche est la tâche empty_task |
| get_duration: task −> int | Renvoie la durée d'une tâche. |
| get_name: task −> string | Renvoie le nom d'une tâche. |
| add_task: task −> graph −> unit | Ajoute une tâche au graphe. |
| get_tasks: graph −> task vect | Renvoie le tableau des tâches du graphe. |
| add_dependence: task −> task −> graph −> unit | Ajoute un arc de dépendance au graphe, de la première vers la seconde tâche.<br>(Attention, une erreur se produit si les tâches n'existent pas dans le graphe.) |
| get_successors: task −> graph −> task vect | Renvoie le tableau des tâches successeurs d'une tâche donnée.<br>(Attention, une erreur se produit si la tâche n'existe pas dans le graphe.) |
| get_predecessors: task −> graph −> task vect | Renvoie le tableau des tâches prédécesseurs d'une tâche donnée.<br>(Attention, une erreur se produit si la tâche n'existe pas dans le graphe.) |

TABLE 2 – La table des fonctions de la bibliothèque Graph de manipulation de graphes de tâches.

---

**Question 3.** *Écrivez en Caml les fonctions suivantes :*

1. *count_tasks: graph −> int : renvoie le nombre de tâches d'un graphe de tâches.*
2. *count_roots: graph −> int : renvoie le nombre de ses tâches racines.*

**Question 4.** *Écrivez en Caml une fonction* make_root_array: graph −> task vect *qui renvoie le tableau des tâches racines du graphe. On pourra si nécessaire renvoyer un tableau plus grand que le nombre de racines. Il sera dans ce cas complété par la tâche* empty_task.

*Un tableau est* complété *par une valeur $u$ si cette valeur n'apparaît pas dans le tableau, ou alors, si elle apparaît, elle n'apparaît pas jusqu'à un certain indice, puis seule cette valeur apparaît au-delà de cet indice.*

# III Ordonnancement par hauteur

On s'intéresse à la recherche d'un ordonnancement $\sigma$ d'un graphe acyclique donné $G$ de $n \geq 1$ tâches sur un ensemble de $p$ processeurs. On rappelle que chaque tâche $u$ dure une unité de temps : $\delta(u) = 1$.

| | |
|---|---|
| set_tag: int −> task −> unit | Affecte une étiquette à une tâche.<br>(Attention, une erreur se produit si une étiquette a déjà été affectée à cette tâche.) |
| get_tag: task −> int | Renvoie l'étiquette d'une tâche.<br>(Attention, une erreur se produit si aucune étiquette n'a été affectée à cette tâche.) |
| has_tag: task −> bool | Renvoie vrai si l'étiquette de la tâche a été définie par la fonction set_tag et faux sinon. |

TABLE 3 – La table des fonctions complémentaires pour la manipulation des étiquettes à valeurs entières des tâches.

---

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. Il est donc intéressant d'étiqueter le graphe de tâches selon ces contraintes.

Pour manipuler les étiquettes à valeurs entières des tâches, on étend la bibliothèque Graph avec les fonctions sur les tâches décrites en table 3.

Cet étiquetage fonctionne de la manière suivante : une tâche $v$ reçoit une étiquette $\mathsf{tag}(v)$ portant un numéro strictement supérieur à celui de toutes les étiquettes $\mathsf{tag}(u)$ des tâches $u$ telles que $u \to v$. Notez que plusieurs tâches peuvent recevoir la même étiquette.

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

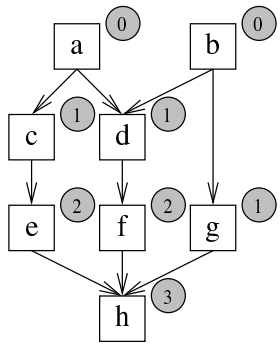
1. Initialement, aucune tâche n'est étiquetée.
2. À l'itération d'ordre $k = 0$, on parcourt l'ensemble des tâches et on affecte l'étiquette 0 aux tâches racines.
3. À l'itération $k > 0$, on parcourt l'ensemble des tâches en repérant toutes les tâches qui n'ont pas encore été étiquetées mais dont toutes 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.

On dira d'une tâche qui a reçu l'étiquette $k$ qu'elle *porte* l'étiquette $k$.

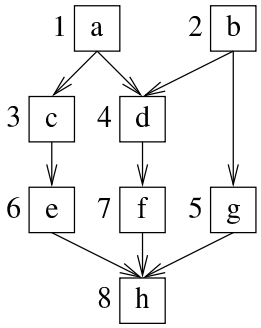
La figure 3a présente un exemple d'étiquetage selon l'algorithme 1. Les étiquettes sont notées dans les cercles grisés à droite des tâches.

**Question 5.** *Écrivez une fonction Caml* check_tags_predecessors: task −> bool *qui prend en paramètre une tâche et renvoie vrai si toutes ses tâches prédécesseurs dans le graphe de tâches sont étiquetées et faux sinon. En particulier, la fonction renvoie vrai si la tâche ne dépend d'aucune tâche (c'est une racine).*

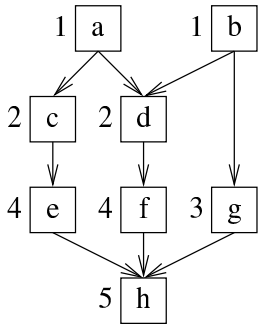
**Question 6.** *Écrivez une fonction Caml* label_height: graph −> unit *qui prend en paramètre un graphe de tâches et affecte à chaque tâche une étiquette selon l'algorithme 1. Veillez bien à ce qu'aucune erreur ne puisse se produire lors des appels des fonctions de la bibliothèque* Graph*.*




(a) Un étiquetage par hauteur du graphe de tâches de la figure 1b.




(b) Un ordonnancement pour le graphe de tâches de la figure 1b avec $p = 1$ processeurs.




(c) Un ordonnancement pour le graphe de tâches de la figure 1b avec $p = 2$ processeurs.

FIGURE 3 – Un exemple d'étiquetage par hauteur et un ordonnancement associé.

*Pour chaque valeur de l'étiquette $k$, on pourra par exemple dans un premier temps repérer l'ensemble des tâches à étiqueter, puis dans un second temps étiqueter ces tâches. Chacune de ces deux actions pourra être implémentée par une fonction auxiliaire.*

Soit $G$ un graphe de tâches. Soit $u$ une tâche de $G$. Soit $P_u$ l'ensemble des chemins de la forme $(u_0, u_1, \ldots, u_n)$, où $u_0$ est une racine de $G$ et $u_n = u$. La tâche $u$ 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. En particulier, le chemin critique amont d'une racine est de longueur nulle et il est unique.

Considérons un graphe de tâches $G$ qui possède un cycle. Soit $u$ une tâche d'un cycle de $G$. Supposons que $P_u$ soit non vide. Alors, il existe une racine $u_0$ de $G$ telle que $u$ soit atteignable de cette racine. On peut produire des chemins arbitrairement longs de $u_0$ à $u$ en parcourant le cycle de manière répétée. La tâche $u$ n'admet donc pas de chemin critique amont.

**Question 7.** *Soit $G$ un graphe de tâches acyclique. Montrez que toutes ses tâches admettent des chemins critiques amont.*

**Question 8.** *Supposons que $G$ soit acyclique. Démontrez qu'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 $k$.*

**Question 9.** *En déduire que l'algorithme termine si et seulement si le graphe est acyclique. Montrez par un exemple ce qui se produit si le graphe possède un cycle.*

**Question 10.** *Démontrez que si une tâche $u$ porte une étiquette $k$, alors elle ne pourra être ordonnancée qu'au moins $k$ unités de temps après que la première tâche a été ordonnancée, quel que soit l'ordonnancement* mais aussi quel que soit le nombre de processeurs utilisés.

Soit $G$ un graphe de tâches acyclique. Soit $k_{\max}$ la valeur maximale des étiquettes attribuées aux tâches de $G$ par l'algorithme 1. Soit $T_{\max}$ l'ensemble des tâches de $G$ qui reçoivent cette étiquette $k_{\max}$. Les chemins critiques amont des tâches de $T_{\max}$ sont appelés *chemins critiques* de $G$. Ces chemins comportent $k_{\max} + 1$ tâches de durée unitaire. Selon la question 10, la *durée d'exécution totale* du graphe de tâches est donc minorée par $k_{\max} + 1$.

Une fois un graphe de tâches $G$ 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. Soit $T_k$ l'ensemble des tâches qui portent l'étiquette $k$.

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

Les figures 3b et 3c présentent des ordonnancements obtenus par cet algorithme à partir de l'étiquetage de la figure 3a, respectivement pour $p = 1$ et $p = 2$ processeurs. On notera en particulier que dans la figure 3c la tâche $e$ reçoit l'étiquette 4 et non 3.

Un ordonnancement sera imprimé de la manière suivante. Chaque ligne entre Begin et End liste les tâches exécutées à un instant donné. Il y a une ligne par instant entre le début et la fin de l'ordonnancement. Le nombre de ligne est donc la durée totale d'exécution de l'ordonnancement.

```
Begin
u v w
x y
...
z

End
```

*On pourra utiliser la fonction* get_name *de la bibliothèque* Graph *pour obtenir le nom des tâches et utiliser la fonction* print_string *pour l'imprimer.*

**Question 11.** *Écrivez une fonction Caml* schedule_height: graph −> int −> unit *qui prend en paramètres un graphe $G$ de tâches étiquetées par l'algorithme 1 et un nombre $p$ de processeurs et qui imprime pour chaque instant $t$ la liste des noms des tâches (au plus $p$) exécutées selon l'algorithme 2 selon le format ci-dessus.*

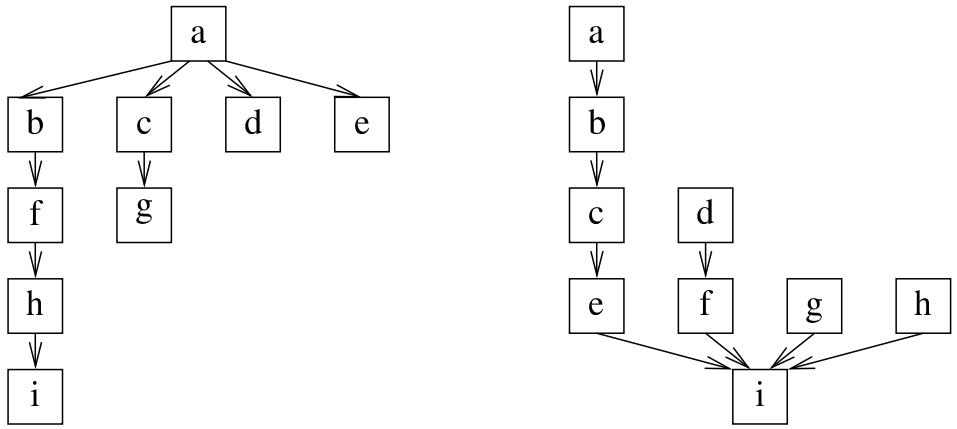
*On pourra par exemple diviser le traitement en plusieurs actions implémentées par des fonctions auxiliaires. Pour chaque valeur de l'étiquette $k$, on extrait l'ensemble des tâches portant cette étiquette. On imprime ensuite cet ensemble par lots de $p$ tâches, avec un lot par ligne, le dernier lot étant éventuellement incomplet.*

Une opération élémentaire de l'algorithme est d'accéder à une tâche par l'une des fonctions des bibliothèques présentées dans les tables 2 et 3. On suppose que chaque opération élémentaire coûte 1.

**Question 12.** *Estimez la complexité de votre fonction* schedule_height *pour l'ordonnancement d'un graphe de $n$ tâches étiquetées par l'algorithme 1 avec $p$ processeurs.*

**Question 13.** *Justifiez que l'ordonnancement ainsi obtenu est optimal pour un seul processeur, c'est-à-dire quand $p = 1$.*

Un graphe de tâches acyclique *arborescent sortant* est un graphe avec une unique racine dans lequel chaque tâche sauf cette racine a exactement un prédécesseur. C'est par exemple le cas du




(a) Un graphe de tâches *arborescent sortant*.

(b) Un graphe de tâches *arborescent entrant*.

FIGURE 4 – Quelques exemples de graphes de tâches arborescents.

graphe de la figure 4a. Un graphe de tâches acyclique *arborescent entrant* est un graphe avec une unique feuille dans lequel chaque tâche sauf cette feuille a exactement un successeur. C'est par exemple le cas du graphe de la figure 4b.

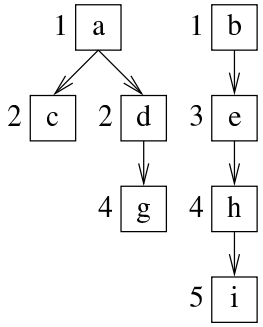
**Question 14.** *Appliquez l'algorithme 1 aux deux graphes de tâches de la figure 4 pour $p = 2$ processeurs. Quelle est la durée totale d'exécution des ordonnancements produits ? Montrez que ces ordonnancements ne sont pas optimaux pour $p = 2$ processeurs en décrivant pour chacun des graphes un ordonnancement dont la durée totale d'exécution est strictement plus courte.*

## IV Ordonnancement par profondeur : l'algorithme de Hu

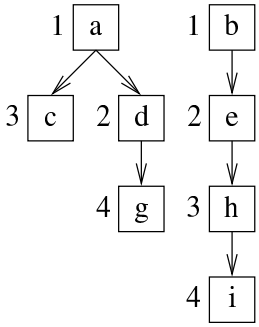
On suppose dans toute la suite du problème que tous les graphes de tâches considérés sont acycliques.

L'étiquetage par hauteur ne fournit pas assez d'informations pour ordonnancer les tâches de manière optimale car il s'appuie sur la structure du graphe en *amont* des tâches étiquetées. La figure 5 décrit par exemple deux ordonnancements du même graphe dans lequel les tâches exécutées sont exécutées dans l'ordre croissant des étiquettes de hauteur. Cependant, l'ordonnancement 5a conduit l'un des processeurs à rester inactif alors que l'ordonnancement 5b permet l'utilisation constante des deux processeurs.

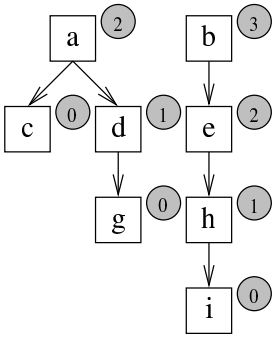
L'idée de cette partie est de mettre en place un autre étiquetage, cette fois-ci fondé sur les plus longs chemins de tâches en *aval*. En effet, la question 16 ci-dessous montrera que la longueur des plus longs chemins de tâches en aval d'une tâche limite inférieurement la durée d'exécution au-delà de 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.




(a) Un ordonnancement pour $p = 2$ processeurs obtenu avec la méthode par hauteur fournissant une durée d'exécution totale de 5.




(b) Un ordonnancement optimal pour $p = 2$ processeurs fournissant une durée d'exécution totale de 4.




(c) L'étiquetage par profondeur.

FIGURE 5 – Un graphe de tâches pour lequel la méthode d'ordonnancement par hauteur de la section III ne fournit pas un ordonnancement optimal pour $p = 2$ processeurs.

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 à chacune de ces tâches l'étiquette $k$.
4. L'algorithme termine quand toutes les tâches sont étiquetées.

La figure 5c présente un exemple d'étiquetage obtenu par l'algorithme 3.

**Question 15.** *Expliquez comment adapter la fonction* label_height *de la question 6 pour obtenir une fonction* label_depth: graph -> unit *qui affecte à chaque tâche une étiquette selon l'algorithme 3.*

Soit $G$ un graphe de tâches. Soit $u$ une tâche de $G$. Soit $P'_u$ l'ensemble des chemins de la forme $(u_0, u_1, \ldots, u_n)$, où $u_0 = u$ et $u_n$ est une feuille de $G$. La tâche $u$ admet un *chemin critique aval* si $P'_u$ est non vide et si l'ensemble des longueurs des chemins de $P'_u$ est majoré. Un *chemin critique aval* de $u$ est un chemin de plus grande longueur dans l'ensemble $P'_u$.

Considérons un graphe de tâches $G$. (On rappelle que les graphes de tâches sont supposés acycliques ici.) Comme pour les chemins critiques amont, toutes les tâches $u$ de $G$ admettent des chemins critiques aval. La *profondeur* d'une tâche $u$, $\mathsf{depth}(u)$, est la longueur commune des *chemins critiques aval* de $u$.

**Question 16.** *Soit $G$ un graphe de tâches acyclique. Soit $u$ une tâche de $G$ de profondeur $h$. Supposons que $u$ soit exécutée à l'instant $t$. Montrez que l'ordonnancement de $G$ ne pourra pas terminer avant l'instant $t + h + 1$ quel que soit le nombre de processeurs utilisés.*

| | |
|---|---|
| Init, Ready, Done | Les valeurs du type state. |
| set_state: state −> task −> unit | Affecte un état à une tâche. |
| get_state: task −> state | Renvoie l'état d'une tâche.<br>(On suppose que l'état des tâches est initialisé à Init lors de leur création.) |

TABLE 4 – La table des fonctions complémentaires pour la manipulation des états des tâches.

---

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. L'ordonnancement commence à l'instant $t_0 = 1$.
2. À chaque instant $t \geq t_0$, on considère l'ensemble $R_t$ des tâches de $G$ prêtes à être exécutées. Soit $r$ le cardinal de cet ensemble.
3. Si $r \leq p$, on choisit pour être exécutées à l'instant $t$ les $r$ tâches de $R_t$ et $p - r$ processeurs restent inactifs.
4. Sinon, on trie les tâches de $R_t$ par ordre décroissant de profondeur et on choisit pour être exécutées à l'instant $t$ les $p$ premières tâches.
5. L'ordonnancement se termine quand toutes les tâches de $G$ ont été exécutées.

On notera que le caractère acyclique de $G$ garantit qu'il y a toujours au moins une tâche prête tant qu'il reste dans $G$ une tâche non exécutée.

**Question 17.** *Appliquez l'algorithme de Hu aux graphes de tâches des figures 4a et 4b pour $p = 2$ processeurs. Quelles sont les durées d'exécution totales obtenues ?*

Pour écrire l'algorithme en Caml, il sera commode de manipuler les états des tâches grâce aux fonctions de la table 4. Ces états appartiennent à un type state qui contient les valeurs suivantes. Une tâche est dans l'état initial Init si elle n'a pas encore été traitée. C'est en particulier le cas lors de sa création par la fonction make_task. Elle est dans l'état Ready si toutes les tâches dont elle dépend ont été exécutées. Elle est dans l'état Done si elle a été exécutée.

*L'état de la tâche empty_task n'est pas défini.*

**Question 18.** *Écrivez une fonction Caml is_ready: task −> bool qui renvoie vrai si toutes les tâches dont dépend la tâche passée en argument sont dans l'état Done et faux sinon. En particulier, la fonction renvoie vrai si la tâche passée en argument ne dépend d'aucune tâche.*

Pour implémenter l'algorithme 4, il faudra trier les tâches du graphe $G$ selon la valeur de leurs étiquettes, par ordre décroissant. On supposera donc qu'on dispose d'une fonction Caml sort_tasks_by_decreasing_tags: task vect −> task vect qui réalise une telle opération. Attention, une erreur se produit si l'étiquette d'une des tâches du tableau n'est pas définie, sauf si c'est la tâche spéciale empty_task. Cette tâche est considérée plus petite que toutes les autres tâches dans le tri.

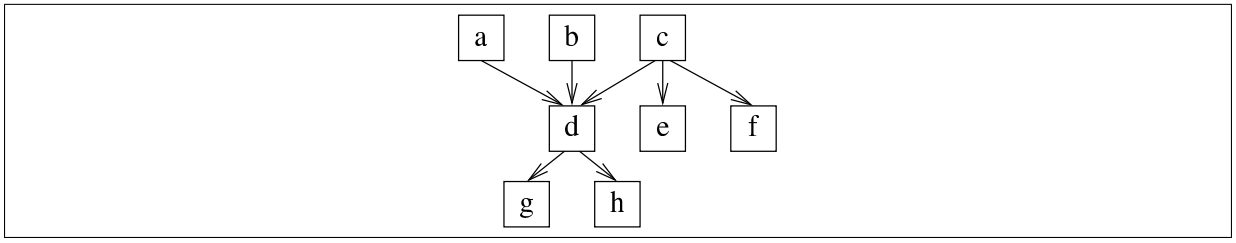



FIGURE 6 – Un graphe de tâches... un peu pathologique.

**Question 19.** *Écrivez une fonction Caml* schedule_Hu: graph −> int −> unit *qui prend en paramètres un graphe $G$ de tâches étiquetées par l'algorithme 3 et un nombre $p$ de processeurs et qui imprime pour chaque instant $t$ la liste des tâches (au plus $p$) exécutées selon l'algorithme de Hu. L'impression doit avoir la même forme que pour la question 11.*

*On pourra par exemple diviser le traitement en plusieurs actions implémentées par des fonctions auxiliaires.*

Une opération élémentaire de l'algorithme est d'accéder à une tâche par l'une des fonctions des bibliothèques présentées dans les tables 2, 3 et 4. On suppose que chaque opération élémentaire coûte 1. On suppose que l'appel à la fonction sort_tasks_by_decreasing_tags sur un tableau de taille $n$ coûte $n \times log_2(n)$.

**Question 20.** *Estimez la complexité de votre fonction* schedule_Hu *pour l'ordonnancement d'un graphe de $n$ tâches étiquetées par l'algorithme 3 avec $p$ processeurs.*

On peut montrer que l'algorithme de Hu est optimal pour les graphes de tâches *arborescents entrants* comme celui de la figure 4b 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.

**Question 21.** *Soit $G$ un graphe de tâches* arborescent entrant *avec $p$ processeurs. Montrez que dans l'algorithme de Hu le cardinal de l'ensemble $R_t$ de tâches prêtes dans $G$ ne peut pas croître au cours de l'algorithme.*

Cette propriété est spécifique du caractère arborescent entrant comme le montre l'exemple de la figure 6.

**Question 22.** *Montrez que l'algorithme de Hu ne conduit pas nécessairement à un ordonnancement optimal avec $p = 2$ processeurs pour le graphe de tâches de la figure 6.*

*On montrera qu'il existe réarrangement trié des tâches de ce graphe qui conduit à un ordonnancement non optimal.*

**Question 23.** *Montrez que l'algorithme de Hu est optimal pour les graphes de tâches* arborescents entrants *avec $p$ processeurs.*

*Cette preuve est délicate. Un approche possible est d'examiner l'activité des processeurs au cours d'un ordonnancement. On peut montrer qu'à partir de l'instant où les processeurs ne sont plus tous actifs, l'exécution est conditionnée par un chemin critique du graphe. Ainsi, l'ordonnancement utilise toujours au mieux les processeurs disponibles.*

# V Conclusion

Ce problème a été initialement étudié par *T. C. Hu* du centre de recherche IBM de Yorktown Heights en 1961. À l'époque, il s'agissait d'organiser le travail d'ouvriers sur une ligne de montage. La preuve d'optimalité dont ce sujet s'est inspiré a été proposée par *James A. M. McHugh* en 1984.

L'algorithme de Hu consiste en fait à définir une mesure de priorité sur les tâches à ordonnancer. La mesure considérée est la profondeur de la tâche. Les tâches sont ordonnancées en privilégiant celles de plus grande priorité. L'exemple de la figure 6 montre que cette approche n'est cependant pas adaptée à des graphes où des tâches ont des dépendances multiples. Dans ce cas, il est important de privilégier certaines tâches par rapport à d'autres parmi toutes les tâches de même profondeur.

En 1972, *E. G. Coffman, Jr.* et *R. L. Graham* ont proposé une amélioration de cette mesure de priorité. L'idée est de départager les tâches de profondeurs égales en privilégiant parmi elles celles qui ont les successeurs les plus profonds en un certain sens. Cet algorithme conduit à un ordonnancement optimal pour les graphes de tâches quelconques, pour $p = 2$ processeurs.

Malheureusement, il ne l'est pas pour le cas $p \geq 3$. Cependant, on peut montrer que tous les algorithmes étudiés ci-dessus conduisent à des ordonnancements dont la durée d'exécution totale n'est pas plus du double de la durée optimale. Cette approximation est heureusement suffisante dans un grand nombre d'applications.

⋆ ⋆ ⋆