Pour ce devoir vous avez le choix entre 2 sujets. Vous devez en choisir un seul et indiquer clairement votre choix sur votre copie. Le devoir dure 4h.
Le premier sujet (pages 1 à 12) est tiré du sujet X-ENS 2015.
Le deuxième sujet (pages 12 à 20 est constitué de trois exercices indépendants sur les graphes, tirées de trois sujets CCINP ou Mines-Ponts distincts.

Sujet 1 : Ordonnancement de graphes de tâches X-ENS 2015

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 a 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 tache 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 noeuds 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 a laquelle l'exécution de la première tache 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. A 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 taches. La partie Il s'intéresse a quelques propriétés des graphes de taches 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 a 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 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 $n, np, \dots$ 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 δ associe à chaque tâche du graphe sa durée. Dans tout ce problème, on supposera que la durée d'une tâche est unitaire : $\delta(u) = 1$.

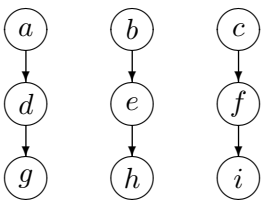
L'ensemble $D \subset T \times T$ est un ensemble d'arcs (dirigés) entre ces tâches (D pour dépendance, voir plus loin). L'existence d'un arc (u, v) dans D est notée $u \to v$. On suppose qu'il n'y a pas d'arc entre une tâche et elle-même : $D \cap \{(u, u)/\ u \in T\} = \emptyset$.

Il y a un arc dirigé (u, v) d'une tâche u vers une autre tâche v distinctes, $u \neq v$, 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.

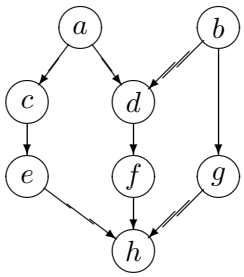
Une tâche qui n'a aucun prédécesseur s'appelle une *racine*. Une tâche qui n'a aucun successeur s'appelle une *feuille*.

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

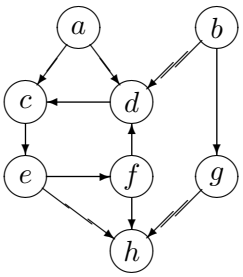
La figure 1 propose quelques exemples de graphes de tâches de petite taille.



(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

Ordonnancement. Un *ordonnancement* σ d'un graphe de tâches $G = (T, D, \delta)$ est une application σ 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_σ du début de l'ordonnancement σ 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_σ 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 σ est $b_\sigma - a_\sigma$.

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

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

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

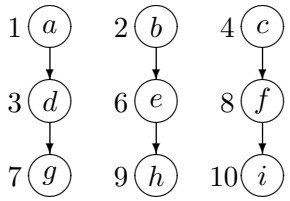
$$S_t = \{u/\ \sigma(u) = t\}$$

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

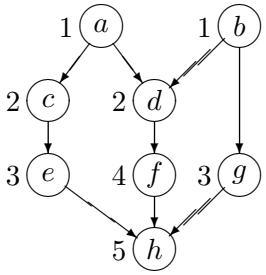
On dit qu'un ordonnancement est optimal pour un graphe de tâches G avec p processeurs si sa durée d'exécution est minimale parmi tous les ordonnancements possibles de G avec p processeurs.

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.

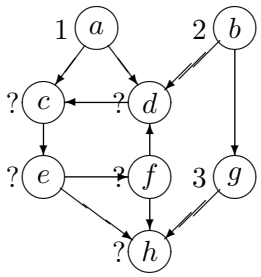
1. L'ordonnancement présenté en figure 2a a une durée d'exécution totale de 10 avec $p = 1$ processeur. Il n'est pas optimal. En effet, aucune tâche n'est exécutée a l'instant 5 où la tâche e est prête ; il serait donc possible de réduire la durée d'exécution totale à 9. Ce serait alors optimal puisqu'il 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 $p = 1$ processeurs



(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 : $u_0 \to u_1$, $u_1 u_2$, …, $u_{n-1} \to u_n$.

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 σ. 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 gr=make_graph ();;
                          add_task a gr;;  add_dependence a c gr;;
let a=make_task "a";;     add_task b gr;;  add_dependence a d gr;;
let b=make_task "b";;     add_task c gr;;  add_dependence b d gr;;
let c=make_task "c";;     add_task d gr;;  add_dependence b g gr;;
let d=make_task "d";;     add_task e gr;;  add_dependence c e gr;;
let e=make_task "e";;     add_task f gr;;  add_dependence d f gr;;
let f=make_task "f";;     add_task g gr;;  add_dependence e h gr;;
let g=make_task "g";;     add_task h gr;;  add_dependence f h gr;;
let h=make_task "h";;                      add_dependence g h gr;;
```

TABLE 1 - Comment construire le graphe de la figure 1b avec la bibliothèque `Graph`

Avant de chercher un ordonnancement d'un graphe de tâche, 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 an 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.

`make_graph : unit → graph`	Crée un graphe vide
`make_task : int → string → task`	Crée une tâche d'une durée donnée avec un nom donné. (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`. (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 : tas → 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 array`	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. (Attention, une erreur se produit si les tâches n'existent pas dans le graphe).
`get_successors : task → graph → task array`	Renvoie le tableau des tâches successeurs d'une tâche donnée. (Attention, une erreur se produit si la tâche n'existe pas dans le graphe)
`get_predecessors : task → graph → task array`	Renvoie le tableau des tâches prédécesseurs d'une tâche donnée. (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

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

- `Array.make n a` renvoie un nouveau tableau de `n` éléments qui contiennent tous la même valeur `a`.
- `Array.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`.
- `Array.sub 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 (Array.length tab)-1 do
      print_string (get_name (tab.(i))); print_string " "
   done;
   print_newline ();;
```

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 des tâches racines.

Question 4. Écrivez en Caml une fonction `make_root_array : graph → task array` 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`.
Attention : 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 σ 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. (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. (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 $\texttt{tag}(v)$ portant un numéro strictement supérieur à celui de toutes les étiquettes $\texttt{tag}(u)$ 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).

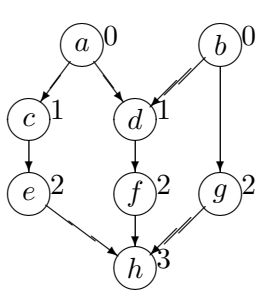
1. Initialement, aucune tâche n'est étiquetée.
2. A l'itération d'ordre $k = 0$, on parcourt l'ensemble des tâches et on affecte l'étiquette 0 aux tâches racines.
3. A l'itération $k > 0$, on parcourt l'ensemble des tâches en repérant toutes les tâches qui n'ont pas encore été étiquetées mais dont toutes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

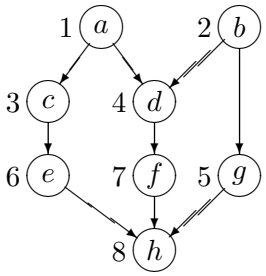
La figure 3*a* présente un exemple d'étiquetage selon l'algorithme 1. Les étiquettes sont indiquées à droite des tâches.

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).

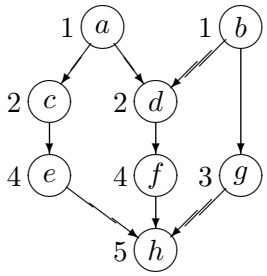
Question 6. Écrivez une fonction Caml `label_height : graph → unit` qui prend en paramètre un graphe de tâches et affecte à chaque tâche une étiquette selon l'algorithme 1. Veillez bien a ce qu'aucune erreur ne puisse se produire lors des appels des fonctions de la bibliothèque `Graph`. *Pour chaque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



(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.

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 tache e reçoit l'étiquette 4 et non 3.

Un ordonnancement sera imprimé de la manière suivante. Chaque ligne entre `Begin` et `End` liste les taches exécutées à un instant donné. Il y a une ligne par instant entre le début et la fin de l'ordonnancement. Le nombre de lignes 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.

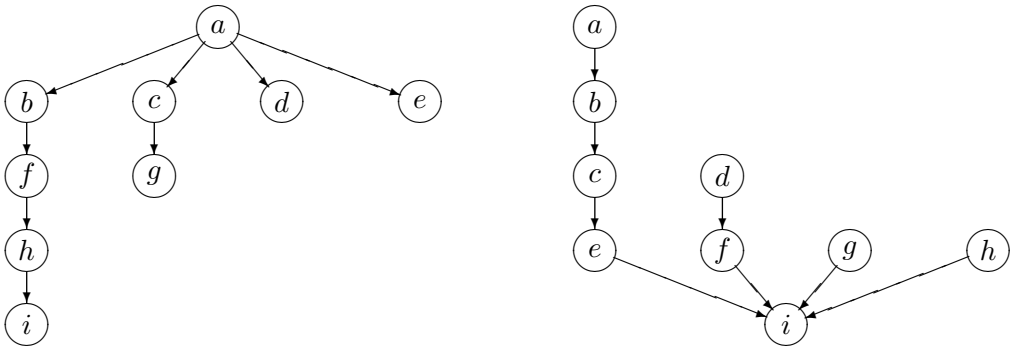
Question 11. Écrivez une fonction Caml `schedule_height : graph → int → unit` qui prend en paramètres un graphe G de tâches étiquetées par l'algorithme 1 et un nombre p de processeurs et qui imprime pour chaque instant t la liste des noms des tâches (au plus p) exécutées selon l'algorithme 2 selon le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



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

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

FIGURE 4 - Quelques exemples de graphes de tâches arborescents.

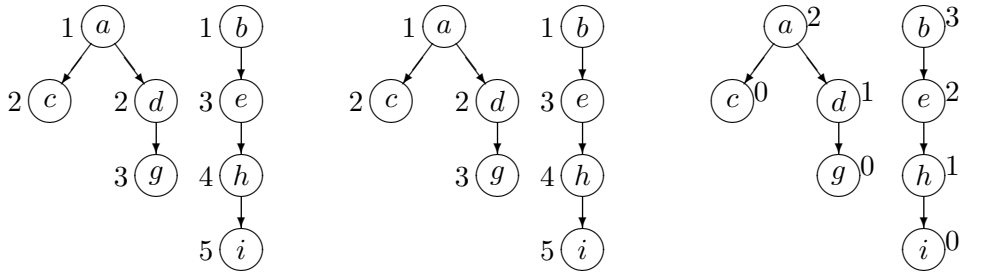
Question 14. Appliquez l algorithme 1 aux deux graphes de tâches de la figure 4 pour $p = 2$ processeurs. Quelle est la durée totale d'exécution des ordonnancements produits ? Montrez que ces ordonnancements ne sont pas optimaux pour $p = 2$ processeurs en décrivant pour chacun des graphes un ordonnancement 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. A l'itération d'ordre $k = 0$, on parcourt l'ensemble des tâches et on affecte l'étiquette 0 aux tâches feuilles.
3. A 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 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, `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 → int`	Affecte un état à une tâche.
`get_state : task → state`	Renvoie l'état d'une tâche. (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. A 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.
Attention : 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 array → task array` 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.

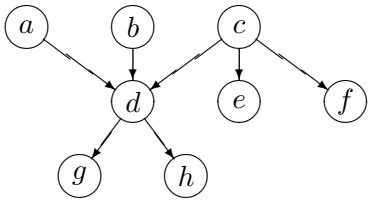


FIGURE 6 - Un graphe de tâches... un peu pathologique

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 ordonnance- ment 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.

Sujet 2 : Problèmes de graphes

Les 3 parties sont indépendantes et peuvent être traitées dans l'ordre de votre choix.

1 Coloration de graphes

La coloration d'une carte de pays consiste à attribuer une couleur à chacun des pays de manière à ce que deux pays voisins soient de couleurs différentes.

Étudier ces techniques de coloration revient de façon plus abstraite à travailler sur des graphes. Le champ d'applications de la coloration de graphes est très vaste et couvre des domaines aussi variés que le problème de l'attribution de fréquences dans les télécommunications, la conception de puces électroniques ou l'allocation de registres en compilation.

Soulignons que tous les graphes considérés dans cet exercice sont nonorientés.

1.1 Introduction sur un exemple

On cherche à colorer une carte de pays avec comme seule contrainte que deux pays ayant une frontière commune ne peuvent être de la même couleur. Comme les pays, les couleurs sont numérotées à partir de zéro.

À titre d'exemple, on considèrera la carte suivante (figure 1), comportant 8 pays numérotés de 0 à 7, que l'on représentera par le graphe G_{ex} donné en figure 2 :

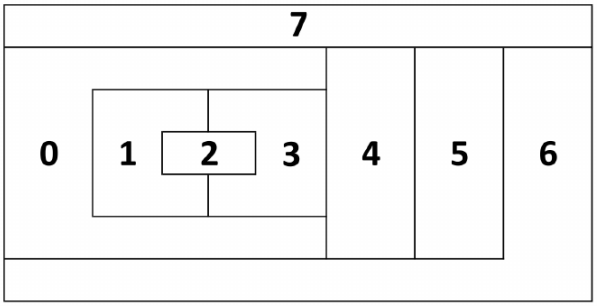


Figure 1 - Exemple d'une carte de pays

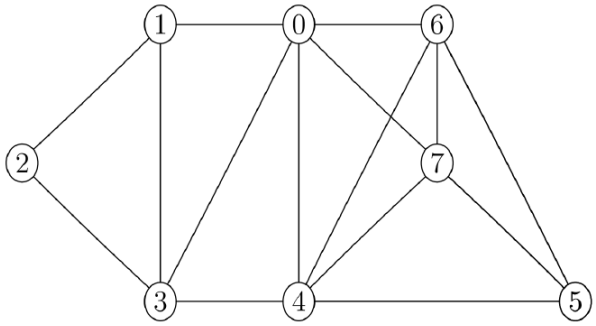


Figure 2 - Graphe G_{ex} associé à la carte de pays de la **figure 1**

1. Dessiner la matrice d'adjacence du graphe G_{ex}.
2. Dessiner la représentation sous forme de liste d'adjacence de G_{ex}.
3. Citer un avantage et un inconvénient de chaque représentations.
4. Dessiner le tableau des degrés des sommets de G_{ex}.

1.2 Tester si une coloration est valide

5. Écrire une fonction `voisins : int list array -> int -> int -> bool` qui étant donné le graphe sous forme de liste d'adjacence et deux entiers i et j, détermine si les sommets i et j sont voisins.

On représente une coloration par un tableau `c` tel que chaque case i contient la couleur du sommet i. On veut vérifier si une coloration proposée est valide, c'est à dire si aucune paire de sommets voisins n'a la même couleur.

6. Écrire une fonction `coloration_valide : int list array -> int array -> bool` qui renvoie `true` si la coloration est valide et `false` sinon.
7. Quelle est la complexité temporelle dans le pire cas de la fonction précédente ?

1.3 Un algorithme intuitif de coloration

On cherche à construire un tableau `c`. On pourra considérer `c[i]` vaut -1 si la couleur n'est pas encore attribuée et donc le tableau est initialement rempli de -1.

8. Écrire une fonction `colore_sommet: int array -> int -> int list array -> unit` qui prend en entrée le tableau C des couleurs attribuées, le numéro s du sommet à colorer et la liste d'adjacence caractérisant le graphe.

   Cette fonction ne renvoie rien mais modifie le tableau C en donnant à $C[s]$ la plus petite couleur possible, en fonction des couleurs des sommets voisins de s qui sont déjà colorés.
9. À l'aide de la fonction précédente, écrire une fonction `colorer1: int list array -> int array` qui renvoie le tableau de couleurs obtenu en colorant les sommets un par un dans l'ordre croissant de leur numéros.

La méthode que nous venons de décrire est rapide et fonctionne plutôt bien. Cependant, si on cherche à utiliser le nombre minimum de couleurs, l'efficacité de l'algorithme proposé ci-dessus dépend en grande partie de l'ordre dans lequel on choisit de colorer les sommets du graphe.

L'objectif des sous-parties suivantes est d'affiner la stratégie pour mieux choisir cet ordre de coloration.

1.4 Variante de Welsh-Powell

Une alternative est donnée par la variante de WelshPowell. L'idée est de parcourir l'ensemble des sommets du graphe par ordre décroissant de leurs degrés. Comme le degré d'un sommet est un entier positif, il est possible d'écrire un algorithme de tri efficace (dit par répartition).

10. Écrire une fonction `degre : int list array -> int array` avec pour argument la liste d'adjacence l d'un graphe quelconque, qui renvoie le tableau des degrés des sommets du graphe.

    Par exemple, pour un graphe de liste d'adjacence $l = [|[1,2],[0,2,3],[0,1,3],[1,2,4],[3]|]$, la fonction `degre` renverra le tableau $[|2,3,3,3,1|]$.

11. Écrire une fonction `ranger : int list array -> int list array` avec pour argument une liste d'adjacence l, qui renvoie un tableau r de même taille que l, telle que $r[i]$ soit la liste des sommets de degré i.

    Ainsi, pour l'exemple de la question précédente, l'appel `ranger(l)` renverra la liste $[|[],[4],[0],[1,2,3],[]|]$.

12. En utilisant la question précédente, écrire une fonction `trier_sommets : int list array -> int array` qui calcule le tableau des sommets dans l'ordre décroissant de degrés.

    Par exemple, pour un graphe de liste d'adjacence $l = [|[1,2],[0,2,3],[0,1,3],[1,2,4],[3]|]$, la fonction `trier_sommets` renverra le tableau de sommets $[|1,2,3,0,4|]$.

13. Quel est la complexité dans le pire cas de la fonction précédente ?

14. Écrire une fonction `colorer2 : int list array -> int array` qui colorie un graphe par le même algorithme qu'en question 9, mais cette fois en colorant les sommets dans l'ordre décroissant de leur degré.

L'amélioration proposée donne de bons résultats dans un grand nombre de cas. Cependant, il reste quelques cas où la coloration obtenue utilise trop de couleurs.

La méthode suivante, proposée en 1979 par Danier Brélaz de l'École Polytechnique Fédérale de Lausanne, raffine la détermination de l'ordre de coloration. La priorité de coloration est ainsi recalculée après chaque traitement d'un sommet et non plus une fois pour toute au départ. Au final, cette approche fournit rapidement une coloration optimale dans un très grand nombre de cas.

1.5 Algorithme DSATUR

Lorsque l'on colore un graphe, le degré de saturation d'un sommet est le nombre de couleurs différentes déjà attribuées à ses différents sommets voisins. Évidemment, ce degré de saturation est susceptible d'évoluer à chaque fois qu'un nouveau sommet est coloré.

15. Écrire une fonction `degre_satur : int list array -> int -> int array -> int` avec 3 arguments, une liste d'adjacence l, un sommet s du graphe, un tableau C de couleurs. Cette fonction renvoie le degré de saturation du sommet s.

    On ne considèrera pas que tous les sommets sont colorés, certains pouvant avoir une couleur de -1 (pas de couleur).

16. Écrire une fonction `liste_satur : int list array -> int array -> int list` avec deux arguments, une liste d'adjacence l, le tableau C des couleurs des sommets, qui renvoie la liste des sommets non colorés du graphe ayant un degré de saturation maximum parmi les sommets non colorés.

    On notera qu'il s'agit d'une liste car plusieurs sommets peuvent avoir le même degré de saturation. On supposera de plus qu'il reste au moins un sommet non coloré.

17. Écrire une fonction `colorer3 : int list array -> int array` ayant pour argument une liste d'adjacence l, qui renvoie un tableau C constituant une coloration du graphe. Cette fonction procède de la façon suivante.

    Tant qu'il reste un sommet non coloré :

— déterminer parmi les sommets non colorés ceux de degré de saturation maximale ;

— si plusieurs sommets non colorés ont un degré de saturation maximale, en choisir un parmi ceux-ci qui soit de degré maximal ;

— colorer le sommet choisi en lui attribuant la couleur disponible ayant la plus petite valeur.

Il n'est pas facile d'être certain d'avoir utilisé le nombre minimum de couleurs. On peut étudier la taille de la plus grande clique (sous-graphe entièrement connecté), qui est un minorant du nombre de couleurs à utiliser.

2 Graphe du web

Le World Wide Web, ou Web, est un ensemble de pages Web (identifiées de manière unique par leurs adresses Web, ou URL pour Uniform Resource Locators, de la forme http ://mines-ponts.fr/index.php) reliées les unes aux autres par des hyperliens. Le Web est souvent modélisé comme un graphe orienté dont les sommets sont les pages Web et les arcs les hyperliens entre pages. Le Web étant potentiellement infini, on s'intéresse à des sous-graphes du Web obtenus en naviguant sur le Web, c'est-à-dire en le parcourant page par page, en suivant les hyperliens d'une manière bien déterminée. Ce parcours du Web pour en collecter des sous-graphes est réalisé de manière automatique par des logiciels autonomes appelés Web crawlers ou crawlers en anglais, ou collecteurs en français.

Fonctions utilitaires

Nous allons tout d'abord coder certaines fonctions de manipulation de structures de données de base, qui seront utiles dans le reste de l'exercice.

1. Coder une fonction `aplatir : ('a * 'a list) list → 'a list`, telle que, si `liste` est une liste de couples $[(x_1, l_{x_1}); \ldots; (x_n, l_{x_n})]$, où chaque x_i est un élément de type `'a`, et l_{x_i} une liste d'éléments de type `'a` de la forme $[y_{i1}; \ldots; y_{ik_i}]$, `aplatir liste` est une liste d'éléments de type `'a`
$$[x_1; y_{11}; \ldots; y_{1k_1}; x_2; y_{21}; \ldots; y_{2k_2}; \ldots; x_n; y_{n1}; \ldots; y_{nk_n}]$$
2. Coder une fonction `tri_fusion : ('a * 'b) list → ('a * 'b) list` triant une liste de couples (x, y) par ordre décroissant de la valeur de la seconde composante y de chaque couple. On devra utiliser l'algorithme de tri par partition–fusion (aussi appelé « tri fusion »). Quelle est la complexité de cet algorithme ?

On va utiliser dans la suite de l'exercice un type de données `dictionnaire` qui permet de stocker des couples formés d'une chaîne de caractères (une clef) et d'un entier (une valeur). On dit que le dictionnaire associe la valeur à la clef. A chaque clef présente dans le dictionnaire est associée une seule valeur. Les fonctions suivantes sont supposées être prédéfinies :

- `dictionnaire_vide : unit → dictionnaire`.
  L'appel `dictionnaire_vide ()` crée un nouveau dictionnaire vide.
- `ajoute : string → int → dictionnaire → dictionnaire`.
  L'appel `ajoute clef valeur dict` renvoie un nouveau dictionnaire identique au dictionnaire *dict*, sauf qu'un couple $(clef, valeur)$ y a été ajouté. Cette fonction s'exécute en temps $O(\log n)$ où n est le nombre d'entrées du dictionnaire.
- `contient : string → dictionnaire → bool`.
  L'appel `contient clef dict` renvoie un booléen indiquant s'il y a un couple dont la clef est $clef$ dans le dictionnaire *dict*. Cette fonction s'exécute en temps $O(\log n)$ où n est le nombre d'entrées du dictionnaire.

- `valeur : string → dictionnaire → int`.
  L'appel `valeur clef dict` renvoie la valeur associée à la clef *clef* dans le dictionnaire *dict*. Cette fonction s'exécute en temps $O(\log n)$ où n est le nombre d'entrées du dictionnaire. Cette fonction ne peut être appelée que si la clef *clef* est présente dans le dictionnaire.

On suppose pour la suite de l'exercice que le type de données `dictionnaire` est prédéfini ; on ne demande pas de l'implémenter.

3. Coder `unique : string list → string list * dictionnaire`, qui est telle que `unique liste` renvoie un couple $(liste', dict)$ où $liste'$ est la liste des chaînes de caractères de liste distinctes (dans l'ordre de leur première occurrence dans liste) et où *dict* associe à chaque chaîne de caractères dans $liste'$ sa position dans $liste'$ (en numérotant à partir de 0). Ainsi l'appel `unique ["x";"zz";"x";"x";"zz";"yt"]` renvoie un couple formé de la liste `["x";"zz";"yt"]` et d'un dictionnaire associant à `"x"` la valeur `0`, à `"zz"` la valeur `1` et à `"yt"` la valeur `2`.
4. Quelle est la complexité de la fonction `unique` en terme de la longueur n de la liste *liste* en argument et du nombre m d'éléments distincts dans la liste *liste* ? Justifier la réponse.

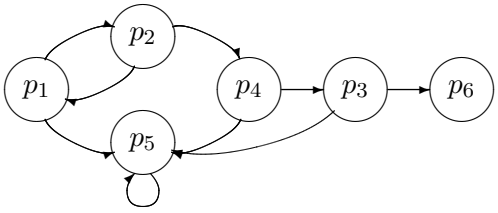
Dans cette partie on pourra utiliser le module Queue pour utiliser une structure de file (structure FIFO). On rappelle les fonctions utiles de ce module, le type `'a t` désignant une file remplie d'un type `'a` :

- `Queue.create : unit -> 'a t` crée une file vide
- `Queue.is_empty : 'a t -> bool` teste si une file est vide
- `Queue.push : 'a -> 'a t -> unit` ajoute un élément à une file
- `Queue.pop : 'a t -> 'a` retire un élément à une file et le renvoie.
- `Queue.peek : 'a t -> 'a` renvoie le premier élément sans le retirer.

Crawler simple

Nous allons maintenant implémenter un crawler simple en Caml. On suppose fournie une fonction `recupere_liens : string → string list` prenant en argument l'URL d'une page Web p et renvoyant la liste des URL des pages q pour lesquelles il existe un hyperlien de p à q, dans l'ordre lexicographique.

Pour illustrer le comportement de cette fonction, nous considérons un exemple de mini-graphe du Web à six pages et neuf hyperliens comme suit :



Dans cette représentation, p_1, p_2, etc., sont les URL de pages Web (simplifiées pour l'exemple), et les arcs représentent les hyperliens entre pages Web.

Dans ce mini-graphe, un appel à `recupere_liens "p1"` retourne la liste `["p2"; "p5"]`.

Un crawler est un programme qui, à partir d'une URL, parcourt le graphe du Web en visitant progressivement les pages dont les liens sont présents dans chaque page rencontrée, en suivant une stratégie de parcours de graphe (par exemple, largeur d'abord, ou profondeur d'abord). A chaque nouvelle page, si celle-ci n'a pas déjà été visitée, tous ses hyperliens sont récupérés et ajoutés à une liste de liens à traiter. Le processus s'arrête quand une condition est atteinte (par exemple, un nombre fixé de pages ont été visitées). Le résultat renvoyé par le crawler, que l'on définira plus précisément plus loin, est appelé un crawl.

5. Coder `crawler_bfs : int → string → (string * string list) list` qui prend en entrée un nombre n de pages et une URL u et renvoie en sortie une liste de longueur au plus n de couples (v, l) où v est l'URL d'une page visitée (les pages apparaissant dans l'ordre où elles ont été visitées) et l la liste des liens récupérés sur la page v. On demande que `crawler_bfs` parcoure le graphe du Web en suivant une stratégie en largeur d'abord (breadth-first search), c'est-à-dire en visitant en priorité les pages rencontrées le plus tôt dans l'exploration. Le crawler doit visiter n pages distinctes, et donc appeler n fois la fonction `recupere_liens` (sauf s'il n'y a plus de pages à visiter). On utilisera une variable de type dictionnaire pour se souvenir des pages déjà visitées.

   Par exemple, sur le mini-graphe, `crawler_bfs 4 "p1"` pourra renvoyer le résultat :

```
["p1", ["p2"; "p5"];
 "p2", ["p1"; "p4"];
 "p5", ["p5"];
 "p4", ["p3"; "p5"]]
```

6. Coder `crawler_dfs : int → string → (string * string list) list` qui prend en entrée un nombre n de pages et une URL u et renvoie en sortie une liste de longueur au plus n de couples (v, l) où v est l'URL d'une page visitée (les pages apparaissant dans l'ordre où elles ont été visitées) et l la liste des liens récupérés sur la page v. On demande que `crawler_dfs` parcoure le graphe du Web en suivant une stratégie en profondeur d'abord (depth-first search), c'est-à-dire en visitant en priorité les pages rencontrées le plus récemment dans l'exploration. Le crawler doit visiter n pages distinctes, et donc appeler n fois la fonction `recupere_liens` (sauf s'il n'y a plus de pages à visiter). On utilisera une variable de type dictionnaire pour se souvenir des pages déjà visitées.

   Par exemple, sur le mini-graphe, crawler_dfs 4 "p1" pourra renvoyer le résultat :

```
["p1", ["p2"; "p5"];
 "p2", ["p1"; "p4"];
 "p4", ["p3"; "p5"];
 "p3", ["p5"; "p6"]]
```

7. Coder une fonction Caml `construit_graphe : (string * string list) list → string list * int array array` telle que si *crawl* est le résultat renvoyé par un crawler (une liste de couples formés d'une URL v et de la liste des liens récupérés sur la page v), alors `construit_graphe crawl` est un couple (l, G) où l est une liste de toutes les URL de pages contenues dans la liste *crawl* et G est la matrice d'adjacence du sous-graphe partiel du Web restreint aux pages de la liste l : G_{ij} est le nombre de liens découverts dans le crawl de la page d'indice i dans l vers la page d'indice j dans l. On fera commencer les indices à 0. Pour coder la fonction `construit_graphe`, on pourra utiliser les fonctions aplatir et unique.

   Par exemple, sur le mini-graphe, si crawl est une variable contenant le résultat de l'appel `crawler_bfs 4 "p1"` (voir question 5), alors `construit_graphe crawl` doit renvoyer :

```
["p1"; "p2"; "p5"; "p4"; "p3"],
[|[|0; 1; 1; 0; 0|];
  [|1; 0; 0; 1; 0|];
  [|0; 0; 1; 0; 0|];
  [|0; 0; 1; 0; 1|];
  [|0; 0; 0; 0; 0|]|]
```

   En particulier :
   - p_3 apparaît même s'il n'a pas été visité dans le crawl ;
   - p_6 n'apparaît pas car il n'a pas été découvert dans le crawl ;
   - l'hyperlien de p_3 à p_5 n'apparaît pas car p_3 n'a pas été visité.

Calcul de PageRank

PageRank est une manière d'affecter un score à l'ensemble des pages du Web, imaginée par Sergey Brin et Larry Page, les fondateurs du moteur de recherche Google. L'introduction de PageRank a révolutionné la technologie des moteurs de recherche sur le Web. Nous allons maintenant implémenter le calcul de PageRank.

Etant donnée une partie du Web (où l'ensemble des pages est indexé entre 0 et $n-1$), la matrice de surf aléatoire dans cette partie du Web est la matrice M de taille $n \times n$ définie comme suit :

- S'il n'y a aucun lien depuis une page Web d'indice i, alors pour tout j, $M_{ij} := 1/n$.
- Sinon, s'il y a k_i liens depuis la page Web d'indice i, alors pour tout j, on a $M_{ij} := (1-d) \times G_{ij}/k_i + d/n$, où G_{ij} est le nombre de liens depuis la page d'indice i vers la page d'indice j et d est un nombre réel fixé appartenant à $[0,1]$ (on prend souvent $d = 0,15$).

Cette matrice peut être vue comme décrivant la marche aléatoire d'un surfeur sur le Web. à chaque fois que celui-ci visite une page Web :

- Si cette page ne comporte aucun lien, il visite une page Web arbitraire, choisie aléatoirement de façon uniforme.
- Si cette page comporte au moins un lien, il visite avec une probabilité égale à $1/d$ un des liens sortants de cette page, et avec une probabilité égale à d une page Web arbitraire, choisie aléatoirement de façon uniforme.

8. Coder `surf_aleatoire : float → int array array → float array array` telle que si d est un nombre entre 0 et 1, et si G est la matrice d'adjacence d'un sous-graphe partiel du Web, alors `surf_aleatoire d G` renvoie la matrice M de surf aléatoire dans ce sous-graphe.
9. Coder `multiplie : float array → float array array → float array`, une fonction prenant en argument un vecteur ligne v de taille n et une matrice M de taille $n \times n$ et renvoyant le vecteur ligne w de taille n résultant du produit de v par la matrice M : $w = vM$. En d'autres termes, pour tout j, $w_j = \sum_i v_i M_{ij}$.

Le PageRank des pages d'un sous-graphe du Web à n pages se calcule par des multiplications successives d'un vecteur ligne par la matrice de surf aléatoire M de ce sous-graphe. Plus précisément, soit θ un nombre réel strictement positif (par exemple, $\theta = 10^{-4}$) et soit $v^{(0)}$ le vecteur ligne de taille n dont toutes les composantes valent $1/n$. On pose pour un entier naturel p arbitraire $v^{(p)} := v^{(0)}M^p$. L'algorithme de PageRank calcule la suite des $v^{(p)}$ pour $p = 0, 1, \ldots$ jusqu'à ce que $\|v^{(p+1)} - v^{(p)}\|_1 \leq \theta$ et renvoie alors le vecteur $v^{(p+1)}$, considéré comme le vecteur des scores de PageRank. On peut montrer (à l'aide du théorème de Perron–Frobenius) que l'algorithme termine dès lors que d est strictement positif.

PageRank est utilisé pour affecter un score d'importance aux pages du Web. Le vecteur de scores v retourné par l'algorithme de PageRank donne dans v_i le score d'importance de la page d'indice i. Les pages de plus haut score de PageRank sont considérées comme les plus importantes.

10. Coder `pagerank : float → float array array → float array`, une fonction prenant en argument un nombre $\theta > 0$ et une matrice M de surf aléatoire d'un sous-graphe du Web et renvoyant le vecteur des scores de PageRank pour θ et M. La fonction `pagerank` devra faire appel à la fonction `multiplie` précédemment codée.
11. Coder `calcule_pagerank : float → float → (string * string list) list → (string * float) list` telle que `calcule_pagerank d theta crawl` renvoie une liste de couples (u, s), un couple pour chaque URL découverte dans le crawl crawl, triée par valeur décroissante de s, où u est l'URL de cette page et s son score de PageRank. Ici, d et θ sont les deux paramètres nécessaires au calcul de la matrice de surf aléatoire et du PageRank respectivement. On pourra faire appel à la fonction `tri_fusion` et à l'ensemble des fonctions développées dans les questions précédentes.

3 Recherche de cliques dans un graphe

3.1 Définitions et propriétés

Définition 1. (Graphe) On appelle **graphe** un couple $G = (S, A)$ où S est l'ensemble des sommets et A est une partie $S \times S$, appelée ensemble des arêtes.

On pourra remarquer que, dans cette définition de graphe, les éléments de la forme (s, s) où $s \in S$, sont des arêtes possibles.

Définition 2. (Clique) Soit $G = (S, A)$ un graphe. Soit S' une partie de S. On dit que S' est une **clique** si :

$$\forall (s_1, s_2) \in S' \times S', (s_1, s_2) \in A$$

Définition 3. (Clique de célébrités, célébrité) Soient $G = (S, A)$ un graphe et C une partie de S. On dit que C est une clique de célébrités si C est une clique et :

$$\forall (c, s) \in C \times S, ((s, c) \in A) \wedge ((c, s) \in A \rightarrow s \in C)$$

Un élément de l'ensemble C est alors appelé **célébrité**.

Le terme "célébrité" provient de l'interprétation suivante : l'ensemble des sommets correspond à un ensemble de personnes et une arête (s, c) représente le fait que s connaît c. Ainsi, une célébrité est connue de tous et elle connaît uniquement les autres célébrités.

1. Dans cette question, on pose $S = \{0, 1, 2, \ldots, 6\}$. Pour chacun des graphes suivants, préciser s'ils contiennent une clique de célébrités non vide. Dans le cas où il y en a une, l'expliciter.
   (a) $G_1 = (S, A_1)$ avec $A_1 = \{(1, 2), (1, 3), (1, 5), (2, 6)\}$.
   (b) $G_2 = (S, A_2)$ avec

$$A_2 = \left\{ \begin{array}{l} (0,3),(0,5),(1,2),(1,3),(2,2),(2,3),(3,3) \\ (4,1),(4,3),(4,5),(5,1),(5,3),(6,1),(6,3) \end{array} \right\}$$

2. Soit $G = (S, A)$ un graphe quelconque. Montrer que s'il existe une clique de célébrités non vide C dans G, alors celle-ci est unique.

Dans la suite, on note C_G l'unique clique de célébrités non vide du graphe G. Dans le cas où celle-ci n'existe pas, C_G désigne alors l'ensemble vide qui est noté $\emptyset$.

3. Soient $G = (S, A)$ un graphe et p un sommet de G. On note $G' = (S \setminus \{p\}, A \cap (S \setminus \{p\} \times S \setminus \{p\}))$. Montrer les propositions suivantes :
   (a) Montrer que si $C_{G'}$ est égal à l'ensemble vide, alors $C_G \in \{\emptyset, \{p\}\}$.
   (b) Montrer que si $C_G \setminus \{p\} \neq \emptyset$, alors $C_{G'} = C_G \setminus \{p\}$
   (c) On suppose que $C_{G'}$ n'est pas l'ensemble vide et on fixe c' un élément de $C_{G'}$.
      i. Montrer que si (p, c') n'est pas une élément de A, alors $C_G \in \{\emptyset, \{p\}\}$.
      ii. Montrer que si (c', p) n'est pas un élément de A, alors $C_G \in \{\emptyset, C_{G'}\}$.
      iii. Montrer que si (p, c') et (c', p) sont des éléments de A, alors $C_G \in \{\emptyset, \{p\} \cup C_{G'}\}$.

3.2 Algorithmique et programmation

Dans la suite, l'ensemble des sommets est de la forme $\{0, 1, \ldots, n-1\}$ où n est un entier supérieur à 1 et un graphe $G = (S, A)$ est représenté en C par sa liste d'adjacence que l'on note L_G.

La liste d'adjacence sera stockée en Ocaml par le type `int list array`.

4. Écrire une fonction en OCaml : `est_clique : int list array -> int list` qui prend en entrée la liste d'adjacence L_G d'un graphe $G = (S, A)$, sa taille et un tableau R représentant une liste sans répétitions d'éléments de S et qui renvoie `true` si l'ensemble des éléments de R constitue une clique de G et `false` sinon.
5. On considère le graphe G ayant comme liste d'adjacence :

$$L_G = [[4, 1, 3, 5], [3, 0, 2], [3, 4, 6], [5, 2, 4, 5, 6], [2, 2], [4, 2, 3, 4], [4, 2, 4, 6]]$$

   Décrire l'évolution de la variable C à chaque étape de l'Algorithme 2 décrit ci-dessous.
6. Montrer par récurrence sur le nombre de sommets que si G est un graphe où C_G est non vide, alors `clique_possible_C` est égale à C_G.

Algorithme 2 - Construction d'une clique de célébrités possibles

```
1  CLIQUE_POSSIBLE_C G :
2  début
3      C ← []
4      S ← [0, 1, ..., n − 1]
       /* n est le nombre de sommet de G                                  */
5      pour chaque s élément de S faire
6          si C est vide alors
7              Ajouter s dans C
8          fin
9          sinon
10             c ← premier élément de C
11             t ← FAUX
               /* t permet de vérifier si on a effectué certaines instructions  */
12             si (s,c) n'est pas une arête de G alors
13                 C ← [s]
14                 t ← VRAI
15             fin
16             si (c,s) n'est pas une arête de G alors
17                 C ← C
18                 t ← VRAI
19             fin
20             si t = FAUX alors
21                 Ajouter s à la fin de liste C
22             fin
23         fin
24     fin
25 fin
26 retourner C
```