Corrigé : ordonnancement de graphes de tâches (X-ENS 2015)

Partie II. Graphe de tâches acyclique

Question 1. Soit $(u = u_0, \dots, u_n = v)$ un chemin de dépendance de longueur $n \geqslant 1$. Pour tout $k \in [\![0, n-1]\!]$ on a $u_k \to u_{k+1}$ donc d'après la contrainte de dépendance (1) nous avons $\sigma(u_k) + 1 \leqslant \sigma(u_{k+1})$. Par télescopage on en déduit :

$$\sigma(u_n) - \sigma(u_0) = \sum_{k=0}^{n-1} (\sigma(u_{k+1}) - \sigma(u_k)) \geqslant \sum_{k=0}^{n-1} 1 = n$$

donc $\sigma(v) \geqslant \sigma(u) + n > \sigma(u)$.
L'existence d'un cycle dans G se traduirait donc par l'existence d'une tâche u telle que $\sigma(u) < \sigma(u)$, ce qui est absurde. G est nécessairement acyclique.

Question 2. Notons n la taille du graphe G et supposons tout d'abord que G n'ait pas de feuille :

$$\forall u \in \mathrm{T}, \quad \exists v \in \mathrm{T} \,\Big|\, u \to v.$$

Considérons une tâche u_0 (G est non vide par définition d'un graphe de tâches). La propriété ci-dessus permet de construire un chemin de dépendance $u_0 \to u_1 \to \cdots \to u_n$ de longueur $n+1$. Mais G est de taille n donc ce chemin comporte un cycle, ce qui est absurde.

Supposons maintenant que G n'ait pas de racine, ce qui se traduit par :

$$\forall v \in \mathrm{T}, \quad \exists u \in \mathrm{T} \,\Big|\, u \to v.$$

Cette hypothèse permet cette fois de construire un chemin de dépendance $u_n \to u_{n-1} \to \cdots \to u_0$ de longueur $n+1$ qui là encore contient un cycle.

On en déduit qu'un graphe de tâches acyclique possède au moins une racine et une feuille.

Question 3. On définit les fonctions :

```
let count_tasks g = vect_length (get_tasks g) ;;

let count_roots g =
  let nb = ref 0 in
  do_vect (function t -> if vect_length (get_predecessors t g) = 0 then incr nb) (get_tasks g) ;
  !nb ;;
```

Question 4. On adapte la définition de **`count_roots`** en remplissant un tableau au fur et à mesure de la découverte de nouvelles racines :

```
let make_root_array g =
  let n = count_tasks g in
  let roots = make_vect n Empty_task in
  let k = ref 0 in
  do_vect (function t -> if vect_length (get_predecessors t g) = 0
                         then (roots.(!k) <- t; incr k)) (get_tasks g) ;
  roots ;;
```

La référence **`k`** désigne la prochaine case à remplir en cas de découverte d'une nouvelle racine.

Partie III. Ordonnancement par hauteur

Question 5. On utilise le principe de l'évaluation paresseuse et du rattrapage d'une exception pour définir :

```
let check_tags_predecessors t g =
  let rec aux tab i = has_tag tab.(i) && aux tab (i+1)
  in
  try aux (get_predecessors t g) 0
  with Invalid_argument "vect_item" -> true ;;
```

Question 6. La fonction suivante utilise deux références : **n** compte le nombre de tâches qui ont reçu une étiquette, **k** désigne le rang de l'itération. La fonction **ready** détermine la liste des tâches prêts à recevoir une étiquette.

```
let label_height g =
  let k = ref 0 in
  let n = ref 0 in
  let t = get_tasks g in
  let rec ready = function
    | i when i = vect_length t -> []
    | i when not has_tag t.(i) && check_tags_predecessors t.(i) g -> i::(ready (i+1))
    | i -> ready (i+1)
  in
  while !n < vect_length t do
    do_list (function i -> set_tag !k t.(i) ; incr n) (ready 0) ;
  incr k
  done ;;
```

Question 7. Soit n la taille de G, et u une tâche de G. Montrons par récurrence sur n que P_u est non vide.

- Si $n = 1$, u est une racine car $u \to u \notin T$ et le chemin (u) appartient à P_u.
- Si $n > 1$, supposons le résultat acquis au rang $n-1$. Si u est une racine, le chemin (u) est élément de P_u. Si u n'est pas une racine, il possède un prédécesseur v. Considérons alors le graphe G′ obtenu en supprimant de G la tâche u ainsi que toutes les dépendances faisant intervenir u. Ce graphe G′ reste acyclique, donc par hypothèse de récurrence $P_v \neq \emptyset$. Notons $(u_0, \dots, u_p = v)$ un chemin de P_v ; alors u_0 est une racine de G′ et $(u_0, \dots, u_p, u)$ un chemin de G. Ce chemin n'est pas un cycle, donc $u \to u_0$ ne fait pas partie des dépendances qui ont été supprimées lorsqu'on a construit G′. Ceci montre que u_0 est une racine de G et ce chemin appartient donc à P_u.

Supposons maintenant que P_u contienne un chemin $(u_0, \dots, u_n)$ de longueur n. Alors il existe $i < j$ tel que $u_i = u_j$, et le chemin $(u_i, \dots, u_j)$ est un cycle, ce qui est absurde par hypothèse. Tous les chemins de P_u ont donc une longueur majorée par n, ce qui assure l'existence d'un chemin critique amont pour u.

Question 8. Commençons par justifier un *principe d'optimalité : si* $(u_0, \dots, u_n)$ *est un chemin critique amont, il en est de même de* $(u_0, \dots, u_{n-1})$.
En effet, s'il existait un chemin $(u'_0, \dots, u'_p)$ entre une racine u'_0 et $u'_p = u_{n-1}$ avec $p > n-1$ alors $(u'_0, \dots, u'_p, u_n)$ serait un chemin amont de u qui serait plus long que $(u_0, \dots, u_n)$, et ce dernier ne serait pas de longueur maximale dans P_u.

Considérons alors u une tâche de G et k la longueur commune des chemins critiques amont de u. Montrons par récurrence sur k que u reçoit l'étiquette k par l'algorithme 1.

- Si $k = 0$, (u) est un chemin critique amont de P_u donc u est une racine ; il reçoit bien l'étiquette 0 par l'algorithme 1.
- Si $k > 0$, supposons le résultat acquis jusqu'au rang $k-1$, et considérons un chemin critique amont $(u_0, \dots, u_k = u)$. D'après le principe d'optimalité $(u_0, \dots, u_{k-1})$ est un chemin critique amont pour u_{k-1}. Par hypothèse de récurrence, u_{k-1} a été étiqueté à l'étape $k-1$ donc u ne peut avoir été étiqueté avant l'étape k.
  Supposons maintenant qu'à l'étape k, u possède un prédécesseur v non encore étiqueté. Compte tenu de l'hypothèse de récurrence, la longueur des chemins critiques amont de v sont de longueur $j \geqslant k$; considérons-en un $(v_0, \dots, v_j)$. Alors $(v_0, \dots, v_j, u)$ est un chemin de P_u de longueur $j+1 > k$, ce qui contredit la définition de k. Ceci montre qu'à l'étape k tous les prédécesseurs de u sont étiquetés et que u se voit donc attribuer l'étiquette k.

Question 9. Si le graphe G est acyclique, la question 7 montre que toutes les tâches admettent un chemin critique amont, et la question 8 permet d'en déduire que toutes ces tâches se voient attribuer une étiquette. L'algorithme se termine donc. Si le graphe G contient un cycle $(u_0, \dots, u_p)$, montrons par récurrence sur k qu'aucune des tâches qui le compose ne peut recevoir l'étiquette k.

- Si $k = 0$, aucune de ces tâches n'est une racine, donc aucune d'elles ne se voit attribuer l'étiquette 0.
- Si $k > 0$, supposons le résultat acquis jusqu'au rang $k-1$. Au début de l'étape k aucune de ces tâches n'a reçu une étiquette, donc toutes possèdent au moins un prédécesseur sans étiquette, et ne peuvent donc se voir attribuer l'étiquette k.

Ceci prouve la non terminaison de l'algorithme lorsque G contient un cycle.
Par exemple, dans le cas du graphe représenté figure 1c, les tâches a et b se voient attribuer l'étiquette 0, la tâche g se voit attribuer l'étiquette 1, puis l'algorithme ne fournit plus aucune étiquette.

Question 10. Dans cette question, on suppose l'étiquetage produit par l'algorithme 1 sur un graphe acyclique (ce n'est pas clairement dit dans l'énoncé).
Montrons par récurrence sur k que si une tâche u porte l'étiquette k, elle ne pourra pas être ordonnancée avant k unités de temps après la première tâche ordonnancée.

- Si $k = 0$ le résultat est évident.
- Si $k > 0$, supposons le résultat acquis au rang $k-1$ et considérons un chemin critique amont $(u_0, \dots, u_k)$ de u. La tâche u_{k-1} possède l'étiquette $k-1$ donc par hypothèse de récurrence elle ne peut avoir été ordonnancée avant $k-1$ unités de temps. On en déduit que u ne peut être ordonnancée avant k unités de temps.

Question 11. Nous allons commencer par écrire une fonction qui remplit un tableau **`ord`** de sorte que **`ord.(k)`** contienne la liste des tâches étiquetées par k. Puisqu'à chaque itération de l'étiquetage au moins une tâche reçoit une étiquette (sinon l'algorithme ne se termine pas) on a $k_{\max} \leqslant n-1$ et un tableau de taille n suffit :

```
let fill_ord g =
  let t = get_tasks g in
  let n = vect_length t in
  let ord = make_vect n [] in
  do_vect (function x -> let k = get_tag x in ord.(k) <- (get_name x)::ord.(k)) t ;
  ord ;;
```

On écrit ensuite une fonction qui imprime une liste de chaînes de caractères par lots de p :

```
let print_pack p lst =
  let rec aux i = function
    | [] -> print_newline ()
    | t::q when i > 0 -> print_string t ; print_string " " ; aux (i-1) q
    | lst -> print_newline () ; aux p lst
  in match lst with
       | [] -> ()
       | _ -> aux p lst ;;
```

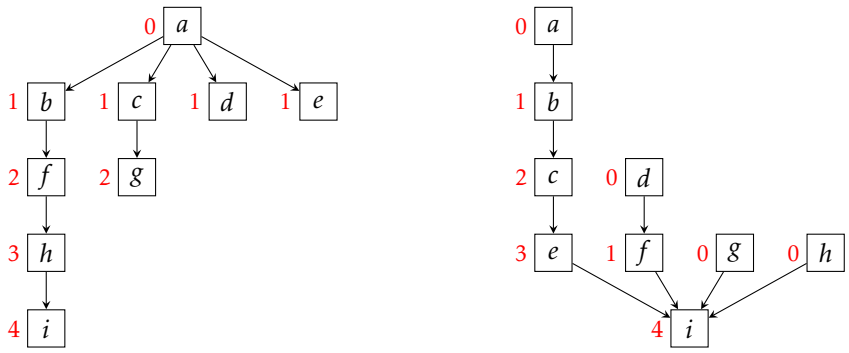
Puis enfin la fonction principale :

```
let schedule_height g p =
  print_string "Begin" ; print_newline () ;
  do_vect (print_pack p) (fill_ord g) ;
  print_string "End" ; print_newline () ;;
```

Question 12. L'insertion en tête de liste est de complexité constante donc le calcul du tableau **`ord`** se réalise en temps proportionnel au nombre de tâches, soit $O(n)$.
À chaque étape de la fonction **`print_pack`** une tâche ou un passage à la ligne est imprimé. Il y a n tâches et au plus n passages à la ligne à réaliser donc la complexité de cette fonction est elle aussi un $O(n)$.
La fonction **`schedule_height`** est donc de complexité linéaire.

Question 13. Lorsque $p = 1$ (cas d'un seul processeur) la durée d'exécution est au moins égale au nombre de tâches à effectuer, c'est-à-dire n. Or l'ordonnancement fourni par l'algorithme 2 fournit un ordonnancement imprimé sur n lignes car pour chaque valeur de $k \in [\![0, k_{\max}]\!]$ correspond au moins une tâche (à chaque itération de l'étiquetage au moins une tâche reçoit une étiquette) ; cet algorithme est donc optimal pour un processeur.

Question 14. L'algorithme 1 appliqué aux deux graphes de la figure 4 fournit les étiquetages suivants :



L'algorithme 2 avec $p = 2$ processeurs fournit les ordonnancements suivants pour chacun de ces deux graphes ;

```
Begin
a
b c
d e
f g
h
i
End
```

```
Begin
a d
g h
b f
c
e
i
End
```

qui dans les deux cas ont une durée d'exécution de 6 unités de temps. Mais aucun de ces deux ordonnancements n'est optimal car on peut réaliser ces tâches en suivant les ordonnancements :

```
Begin
a
b c
f d
h g
i e
End
```

```
Begin
a d
b f
c h
e h
i
End
```

qui ont une durée d'exécution de 5 unités de temps (et qui sont optimaux).

Partie IV. Ordonnancement par profondeur : l'algorithme de Hu

Question 15. Pour obtenir la fonction **`label_depth`** il suffit dans la fonction **`label_height`** de remplacer la fonction **`check_tags_predecessors`** par la fonction **`check_tags_successors`** définie ci-dessous :

```
let check_tags_succecessors t g =
  let rec aux tab i = has_tag tab.(i) && aux tab (i+1)
  in
  try aux (get_successors t g) 0
  with Invalid_argument "vect_item" -> true ;;
```

Question 16. Montrons par récurrence sur h que l'ordonnancement de G ne pourra pas se terminer avant l'instant $t+h+1$.

- Si $h = 0$ c'est évident, puisque la durée des tâches est de une unité de temps.
- Si $h > 0$, supposons le résultat acquis au rang $h-1$ et considérons un chemin critique aval $(u_0, \ldots, u_h)$ de u. D'après le principe d'optimalité le chemin $(u_1, \ldots, u_h)$ est un chemin critique aval de u_1 donc la tâche u_1 possède l'étiquette $h-1$ et ne peut être exécutée avant l'instant $t+1$. Par hypothèse de récurrence l'ordonnancement de G ne pourra se terminer avant l'instant $(t+1)+h = t+h+1$, ce qui prouve le résultat au rang h.

Question 17.

- Pour le graphe de la figure 4a :
  À la date $t = 1$, $R_1 = \{a\}$: a est exécutée ;
  À la date $t = 2$, $R_2 = \{b, c, d, e\}$: b et c sont exécutées ;
  À la date $t = 3$, $R_3 = \{d, e, f, g\}$: f et une autre tâche, par exemple g, sont exécutées ;
  À la date $t = 4$, $R_4 = \{d, e, h\}$: h et une autre tâche, par exemple e, sont exécutées ;
  À la date $t = 5$, $R_5 = \{d, i\}$: d et i sont exécutées.

– Pour le graphe de la figure 4b :
  À la date $t = 1$, $R_1 = \{a, d, g, h\}$: a et d sont exécutées ;
  À la date $t = 2$, $R_2 = \{b, f, g, h\}$: b et une autre tâche, par exemple f, sont exécutées ;
  À la date $t = 3$, $R_3 = \{c, g, h\}$: c et une autre tâche, par exemple g, sont exécutées ;
  À la date $t = 4$, $R_4 = \{e, h\}$: e et h sont exécutées ;
  À la date $t = 5$, $R_5 = \{i\}$: i est exécutée.

Dans les deux cas la durée d'exécution totale obtenue est égale à 5 (ils correspondent aux contre-exemples de la question 14).

Question 18. On définit la fonction :

```
let is_ready t g =
  let rec aux tab i = get_state tab.(i) = Done && aux tab (i+1)
  in
  try aux (get_predecessors t g) 0
  with Invalid_argument "vect_item" -> true ;;
```

Question 19. L'ordre relatif qui régit les tâches (leur profondeur) n'est pas modifié lors de l'algorithme ; je ne vais donc réaliser qu'*une seule fois* le tri du tableau des tâches par ordre décroissant de profondeur au début de l'algorithme.

Tant que toutes les tâches n'auront pas été réalisées, il va falloir parcourir le tableau des tâches à la recherche de celles qui sont prêtes à être exécutées mais qui ne l'ont pas encore été. Puisque le tableau aura été trié par ordre décroissant de profondeur, il suffit de ne sélectionner que les p premières, qui passeront de l'état **`Init`** à l'état **`Ready`** :

```
let select_ready g t p =
  let nb = ref 0 in
  let rec aux n = function
    | i when n = 0 or i = vect_length t -> !nb
    | i when get_state t.(i) <> Done && is_ready t.(i) g -> set_state Ready t.(i) ;
                                                           incr nb ; aux (n-1) (i+1)
    | i -> aux n (i+1)
  in aux p 0 ;;
```

La fonction **`select_ready`** prend en arguments un graphe, son tableau des tâches et l'entier p et renvoie le nombre de tâches prêtes à être exécutées. Ce nombre sera compris entre 0 (ce qui marquera la fin de l'algorithme) et p.

Nous aurons aussi besoin d'une fonction qui imprime les n premières tâches portant le label **`Ready`** du tableau des tâches, tout en mettant à jour ce label :

```
let print_task t n =
  let rec aux k = function
    | i when k = 0 -> print_newline ()
    | i when get_state t.(i) = Ready -> print_string (get_name t.(i)) ; print_string " " ;
                                        set_state Done t.(i) ; aux (k-1) (i+1)
    | i -> aux k (i+1)
  in aux n 0 ;;
```

Il reste à rédiger la fonction principale :

```
let schedule_Hu g p =
  print_string "Begin" ; print_newline () ;
  let t = sort_tasks_by_decreasing_tags (get_tasks g) in
  let rec aux () =
    match select_ready g t p with
      | 0 -> print_string "End" ; print_newline ()
      | n -> print_task t n ; aux ()
  in aux () ;;
```

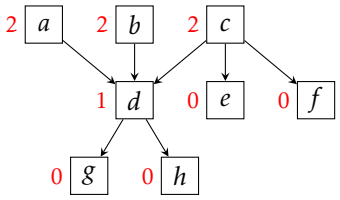
Question 20. Puisque le nombre de prédécesseurs d'une tâche est majoré par $n-1$, la complexité temporelle de la fonction **`is_ready`** est en $O(n)$. La fonction **`select_ready`** exécute au plus n fois cette fonction donc sa complexité temporelle est un $O(n^2)$.

La fonction **`print_task`** parcours tout ou partie du tableau des tâches donc a une complexité temporelle en $O(n)$.

Enfin, la fonction **`schedule_Hu`** trie une fois le tableau des tâches pour un coût en $O(n \log n)$ puis exécute au plus n fois les deux fonctions précédentes, donc on peut majorer sa complexité temporelle par un $O(n^3)$.

Question 21. Considérons une tâche de R_t ; elle possède zéro ou un fils. Son exécution permet donc à *au plus* une tâche supplémentaire de devenir prête à être exécutée. Ainsi, si k est le nombre de tâches exécutées à l'instant t on a $|R_{t+1}| = |R_t| - k + k'$ avec $k' \leqslant k$ ce qui montre que $|R_{t+1}| \leqslant |R_t|$.

Question 22. Considérons le graphe des tâches de la figure 6. Son étiquetage par profondeur depuis les feuilles est :



Plusieurs tris par étiquette décroissante sont possibles. Si l'algorithme de tri retourne l'ordre $a \geqslant b \geqslant c \geqslant d \geqslant e \geqslant f \geqslant g \geqslant h$ l'algorithme de Hu fournit l'exécution des tâches suivante :

```
Begin
a b
c
d e
f g
h
End
```

En revanche si l'ordre retourné est $c \geqslant b \geqslant a \geqslant d \geqslant e \geqslant f \geqslant g \geqslant h$ cet algorithme fournit l'exécution :

```
Begin
c b
a e
d f
g h
End
```

Cet exemple montre que l'algorithme de Hu ne garantit pas un ordonnancement optimal.

Question 23. Nous allons montrer par récurrence sur p que l'algorithme de Hu est optimal pour un graphe arborescent entrant avec p processeurs.

- Si $p = 1$ le résultat est évident : le processeur effectue une tâche à chaque itération.
- Si $p > 1$, supposons le résultat acquis jusqu'au rang $p - 1$.
  Comme le suggère l'énoncé, considérons l'instant t où pour la première fois l'un des processeurs va être inactif (si un tel instant n'existe pas l'exécution ne peut qu'être optimale).
  Si on supprime du graphe les tâches déjà exécutées, le graphe restant est toujours arborescent entrant, ce qui permet sans perte de généralité de supposer $t = 0$. Le graphe possède alors un nombre de racines $k < p$, et dorénavant seuls au plus k processeurs pourront travailler en parallèle. Autrement dit, tout se passe comme si nous n'avions plus que k processeurs à notre disposition, ce qui permet d'appliquer l'hypothèse de récurrence et conclure que l'algorithme de Hu est optimal.