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

Rédigé par Jean-Pierre Becirspahic (jp.becir@info-llg.fr).

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 get_predecessors t g = [||] 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 get_predecessors t g = [||]
                         then (roots.(!k) <- t; incr k)) (get_tasks g) ;
  roots ;;
```

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 ;;
```

À mon avis il y a là une erreur d'énoncé : le type de cette fonction est *task -> graph -> bool*.

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 **aux** 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 aux = function
    | i when i = vect_length t -> []
    | i when not has_tag t.(i) && check_tags_predecessors t.(i) g -> i::(aux (i+1))
    | i -> aux (i+1)
  in
  while !n < vect_length t do
    do_list (function i -> set_tag !k t.(i) ; incr n) (aux 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 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.

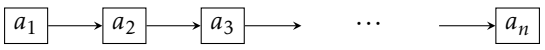
Question 10. Dans cette question, on suppose l'étiquetage produit par l'algorithme 1 sur un graphe acyclique (ce n'est pas clairement dit dans l'énoncé).
Montrons par récurrence sur k que si une tâche u porte l'étiquette k, elle ne pourra pas être ordonnancée avant k unités de temps après la première 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. La fonction ci-dessous utilise trois références : **n** compte le nombre de tâches ordonnancées ; **k** le rang d'itération dans l'algorithme 1 et **j** le nombre de tâches exécutées en parallèle dans un lot.

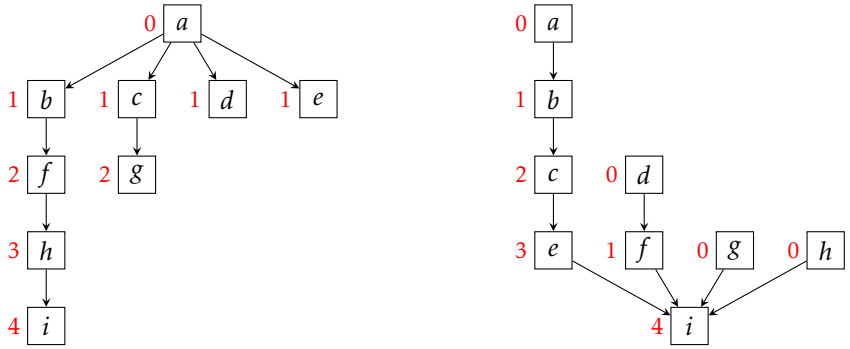
```
let schedule_height g p =
  print_string "Begin" ; print_newline () ;
  let n = ref 0 in
  let k = ref 0 in
  let j = ref 0 in
  let t = get_tasks g in
  while !n < vect_length t do
    for i = 0 to vect_length t - 1 do
      if get_tag t.(i) = !k
      then (incr j ;
            if !j = p + 1 then (j := 0 ; print_newline ()) ;
            print_string (get_name t.(i)) ; print_string " " ;
            incr n)
    done ;
    print_newline () ;
    incr k ; j := 0
  done ;
  print_string "End" ; print_newline () ;;
```

Question 12. La fonction précédente effectue $k_{\max}+1$ parcours du tableau des tâches de G en effectuant à chaque fois des opérations de coût constant donc le coût total de cette fonction est un $O(n.k_{\max})$. Par ailleurs, à chaque itération de l'étiquetage au moins une tâche reçoit une étiquette (car sinon l'algorithme ne se termine pas) donc $k_{\max} \leqslant n-1$. On en déduit que le coût de la fonction précédente est un $O(n^2)$. Cette complexité est effectivement atteinte pour un graphe de la forme ci-dessous, et ce quel que soit le nombre de processeurs :



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 (ceci a déjà été expliqué à la question précédente) ; 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.

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 ;;
```

Là encore, il y a à mon avis une erreur d'énoncé puisque le type de cette fonction est *task -> graph -> bool*.

Question 19. On définit la fonction :

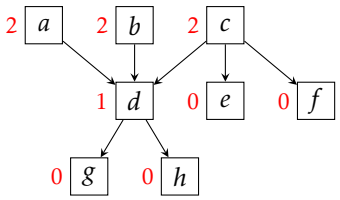
```
let schedule_Hu g p =
  print_string "Begin" ; print_newline () ;
  let n = ref 0 in
  let r = ref 0 in
  let j = ref 0 in
  let t = get_tasks g in
  let rec aux = function
    | i when i = vect_length t -> []
    | i when get_state t.(i) <> Done && is_ready t.(i) g  -> i::(aux (i+1))
    | i -> aux (i+1)
  in
  sort_tasks_by_decreasing_tags t ;
  while !n < vect_length t do
    do_list (function i -> set_state Ready t.(i) ; incr r) (aux 0) ;
    r := min !r p ;
    j := 0 ;
    while !r > 0 do
      if get_state t.(!j) = Ready
      then (print_string (get_name t.(!j)) ; print_string " " ;
            set_state Done t.(!j) ;
            incr n ; decr r) ;
      incr j
    done ;
    print_newline ()
  done ;
  print_string "End" ; print_newline () ;;
```

La référence **n** compte le nombre de tâches déjà traitées, la fonction **aux** dresse la liste des tâches prêtes à être exécutées et la référence **r** compte leur nombre.

Question 20. Le coût de la fonction auxiliaire **aux** est un $O(n)$ (parcours du tableau des tâches **t**). Dans le pire des cas, à chaque étape seule une tâche peut être exécutée (cf question 12) et dans ce cas la fonction **aux** est exécutée n fois. Sachant que le tri du tableau **t** n'est effectué qu'une fois, le coût total de cette fonction est un $O(n^2)$.

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 branches $k < p$, et dorénavant seuls 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.

Annexe : une définition élémentaire des fonctions du module `Graph`

Définition des différents types :

```
type state = Init | Ready | Done ;;
type tag = None | Tg of int ;;
type non_empty_task = {Id: string; Duration: int; mutable Tag: tag; mutable State: state} ;;
type task = Empty_task | Task of non_empty_task ;;
type graph = {mutable Tasks: task list; mutable Dependencies: (task * task) list} ;;
```

Les fonctions de la table 2 :

```
let make_graph () = {Tasks = []; Dependencies = []} ;;
```

```
let make_task d id = Task {Id = id; Duration = d; Tag = None; State = Init} ;;
```

```
let is_empty_task t = t = Empty_task ;;
```

```
let get_duration = function
  | Empty_task -> failwith "get_duration"
  | Task t -> t.Duration ;;
```

```
let get_name = function
  | Empty_task -> failwith "get_duration"
  | Task t -> t.Id ;;
```

```
let add_task t g = match t with
  | Empty_task -> failwith "add_task"
  | _ when mem t g.Tasks -> ()
  | _ -> g.Tasks <- t::g.Tasks ;;
```

```
let get_tasks g = vect_of_list g.Tasks ;;
```

```
let add_dependence t1 t2 g =
  if not (mem t1 g.Tasks && mem t2 g.Tasks)
  then failwith "add_dependence"
  else
    if not mem (t1, t2) g.Dependencies
    then g.Dependencies <- (t1, t2)::g.Dependencies ;;
```

```
let get_successors t g =
  let rec aux = function
    | [] -> []
    | (x, y)::q when t = x -> y::(aux q)
    | _::q -> aux q
  in
  if not mem t g.Tasks
  then failwith "get_successors"
  else vect_of_list (aux g.Dependencies) ;;
```

```
let get_predecessors t g =
  let rec aux = function
    | [] -> []
    | (x, y)::q when t = y -> x::(aux q)
    | _::q -> aux q
  in
  if not mem t g.Tasks
  then failwith "get_successors"
  else vect_of_list (aux g.Dependencies) ;;
```

Les fonctions de la table 3 :

```
let set_tag n = function
  | Task t when t.Tag = None -> t.Tag <- Tg n
  | _ -> failwith "set_tag" ;;
```

```
let get_tag = function
  | Empty_task -> failwith "get_tag"
  | Task t -> match t.Tag with
              | None -> failwith "get_tag"
              | Tg n -> n ;;
```

```
let has_tag = function
  | Empty_task -> failwith "get_tag"
  | Task t -> t.Tag <> None ;;
```

Les fonctions de la table 4 :

```
let set_state s = function
  | Empty_task -> failwith "set_state"
  | Task t -> t.State <- s ;;
```

```
let get_state = function
  | Empty_task -> failwith "get_state"
  | Task t -> t.State ;;
```

Et enfin une définition de la fonction **`sort_tasks_by_decreasing_tags`** utilisant l'algorithme de tri rapide (*quicksort*) :

```
let swap tab i j = let x = tab.(i) in tab.(i) <- tab.(j) ; tab.(j) <- x ;;
```

```
let sup = fun
        | Empty_task _ -> false
        | _ Empty_task -> true
        | t1 t2 -> get_tag t1 > get_tag t2 ;;
```

```
let rec segmente tab p i = function
  | j when  j < i -> j
  | j when sup tab.(i) p -> segmente tab p (i+1) j
  | j -> swap tab i j ; segmente tab p i (j-1) ;;
```

```
let sort_tasks_by_decreasing_tags tab =
  let rec aux i = function
    | j when j <= i -> ()
    | j -> let k = segmente tab tab.(i) (i+1) j in
           swap tab i k ;
           aux i (k-1) ; aux (k+1) j
  in aux 0 (vect_length tab - 1) ;;
```