Ordonnancement de graphes de tâches
X-ENS 2015

Graphe de tâches acyclique

Q.1 Notons $H(n)$ la proposition

pour toutes tâches u, v, s'il existe dans G un chemin de longueur n de u vers v alors
$$\sigma(u) < \sigma(v)$$

et prouvons, par récurrence, que la proposition $H(n)$ est vraie pour tout $n \geq 1$.

- Initialisation : soient u, v deux tâches telles qu'il existe un chemin de longueur 1 de u à v. On a alors $u \to v$ et donc $\sigma(u) + \delta(u) \leq \sigma(v)$. Comme δ est valeurs > 0, on en déduit que $\sigma(u) < \sigma(v)$.
- Hérédité : soit $n \geq 1$ tel que $H(1), \dots, H(n)$ soient vraies. Soient u, v des tâches telles qu'il existe un chemin de longueur $n+1$ de u à v. En notant $(u_0, \dots, u_{n+1})$ ce chemin, on a $u_0 = u$, $u_{n+1} = v$, $(u_0, \dots, u_n)$ qui est un chemin de dépendance et $u_n \to u_{n+1}$. Par hypothèse de récurrence, $\sigma(u) < \sigma(u_n)$ et par la propriété (1), $\sigma(u_n) + \delta(u_n) \leq \sigma(v)$. On a donc $\sigma(u) < \sigma(v) - \delta(u_n) < \sigma(v)$ puisque δ est à valeurs > 0. Ceci prouve le résultat au résultat $n+1$.

$H(n)$ étant vraie pour tout $n \in \mathbb{N}^*$ et tout chemin ayant une longueur $n \geq 1$, on a ainsi le résultat demandé.

Si, par l'absurde, G était cyclique, il existerait un chemin de longueur non nulle d'une tâche u vers elle même et ainsi on aurait $\sigma(u) < \sigma(u)$ ce qui est impossible. On a ainsi montré (en contraposant) que

un graphe cyclique n'admet pas d'ordonnancement

Q.2 Soit G un graphe acyclique de taille n. D'après le lemme des tiroirs, tout chemin de longueur $\geq n$ passe deux fois par au moins une tâche et contient donc un cycle, ce qui est exclus. L'ensemble des longueurs des chemins dans G est donc majoré. C'est une partie non vide de $\mathbb{N}$ (il y a des chemins de longueur nulle puisqu'il y a au moins une tâche)) qui admet donc un maximum p. On peut ainsi un trouver un chemin $u_1 \to \cdots \to u_p$ de longueur maximale. u_0 est alors une racine et u_p une feuille (puisque dans le cas contraire on pourrait agrandir le chemin).

Q.3 Il nous suffit de donner la taille du taille du tableau des tâches.

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

Une tâche est une racine si elle n'a aucun prédécesseur, c'est à dire un tableau de prédécesseur de taille nulle. Il suffit de faire le test pour chaque tâche et de gérer un compteur.

```
let count_roots g =
   let tab=get_tasks g in
   let c=ref 0 in
   for i=0 to (vect_length tab -1) do
       if vect_length (get_predecessors tab.(i))=0 then incr c
       done ;
    !c ;;
```

Q.4 La structure est très semblable. On crée un tableau pour stocker les tâches racines. Comme il y a moins de racines que de tâches, on connaît un majorant de la taille à lui donner. En l'initialisant avec la tâche vide, on respectera la contrainte de l'énoncé. Cette fois, le compteur indique le numéro de la prochaine case à remplir.

```
let make_root_array g =
   let tab=get_tasks g in
   let tabroot=make_vect (vect_length tab) empty_task in
   let c=ref 0 in
   for i=0 to (vect_length tab -1) do
       if vect_length (get_predecessors tab.(i) g)=0 then begin
```

```
            tabroot.(!c) <- tab.(!c) ;
            incr c
            end
        done ;
    tabroot ;;
```

Ordonnancement par hauteur

Q.5 On peut penser que l'énoncé a oublié de donner le graphe en argument à la fonction demandée. J'écris donc plutôt une fonction de signature
`check_tags_predecessors : task -> graph -> bool`
On calcule la liste des prédécesseurs en s'arrêtant quand on arrive au bout ou quand on trouve un prédécesseur non étiqueté. Tous les prédécesseurs sont étiquetés si et seulement si toutes les cases ont été examinées en fin de boucle.

```
let check_tags_predecessors t g =
   let pred=get_predecessors t g in
   let n=vect_length pred in
   let i=ref 0 in
   while !i<n && (has_tag pred.(!i)) do incr i done ;
   !i=n ;;
```

Q.6 Comme proposé par l'énoncé, on va écrire des fonctions auxiliaires. On choisit cependant de les définir localement afin de ne pas avoir à leur passer en argument le graphe (`g`) ou le tableau des tâches (`tab`). Ceux-ci sont donc supposés connus des fonctions. Il nous faut choisir des structures de données pour représenter les "tâches à étiqueter". On pourrait choisir, comme pour `make_root_array` un tableau éventuellement surdimensionné et complété par `empty_task`. Je choisis plutôt d'utiliser une liste de tâches et je vais donc naturellement utiliser des stratégies récursives.

La première fonction est `reperer : int` $\rightarrow$ `task list`. Dans l'appel `reperer i`, on renvoie la liste de toutes les tâches de position $\leq i$ dans `tab` et qui sont à étiqueter (c'est à dire non étiquetées et avec tous ses prédécesseurs étiquetés). Ainsi, en notant n le nombre total de tâches, l'appel `reperer`$(n-1)$ donnera la liste de toutes les tâches à traiter. Le code de la fonction est alors quasi-immédiat.

La seconde fonction est `etiqueter : task list` $\rightarrow$ `int` $\rightarrow$ `unit`. Dans l'appel `etiqueter liste k`, on SUPPOSE que les tâches de la liste sont non étiquetées et on les étiquette avec l'entier k. Là encore, le code est immédiat.

Enfin, je propose de conclure avec une dernière fonction `iterer : int` $\rightarrow$ `unit`. Dans l'appel `iterer k`, on effectue les étapes de numéro $\geq k$ de la boucle proposée par l'énoncé (l'appel avec $k=0$ donne donc l'algorithme entier). On sait que l'on a fini (cas de base) quand la liste des tâches à traiter est vide (dans ce cas, on ne fait rien, c'est le cas du `else` non écrit dans la fonction).

```
let label_height g =
   let tab=get_tasks g in
   let n=vect_length tab in
   let rec reperer i =
      if i=(-1) then []
      else begin
            if (not (has_tag tab.(i))) && (check_tags_predecessors tab.(i) g)
                then tab.(i)::(reperer (i-1))
            else reperer(i-1)
            end in
    let rec etiqueter avoir k =
```

```
      match avoir with
      [] -> ()
     |t::q -> set_tag k t ; etiqueter q k in
  let rec iterer k =
       let avoir=reperer (n-1) in
       if avoir<>[] then begin
             etiqueter avoir k;
             iterer (k+1)
             end
   in iterer 0 ;;
```

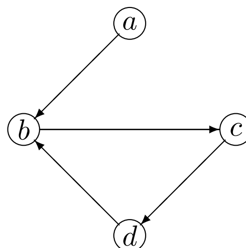
Q.7 On suppose G acyclique et u une tâche de G. L'ensemble des longueurs des chemins qui se terminent en u est non vide (il y a le chemin (u) de longueur nulle) et est majoré (on a vu en question **2** que dans un graphe acyclique de taille n, tout chemin est de longueur $\leq n-1$). Il y a donc une longueur maximale et un chemin $(u_0, u_1, \dots, u)$ associé. Par maximalité de la longueur, u_0 est une racine et le chemin est un chemin critique amont de u.

Q.8 Il nous suffit de prouver qu'à l'étape k, on étiquette exactement les tâches dont les chemins critiques amont sont de longueur k, ce que l'on fait par récurrence.

- Initialisation : une tâche à un chemin critique amont de longueur nulle si et seulement c'est une racine et à l'étape 0, on étiquette bien les racines exactement.
- Hérédité : soit $k \geq 1$ tel que le résultat soit vrai aux rangs $0, \dots, k-1$. Soit u une tâche non étiquetée, c'est à dire, par hypothèse de récurrence, dont la longueur d'un chemin critique amont $(u_0, \dots, u_p = u)$ est $p \geq k$.
  - Si $p = k$ alors tout chemin d'une racine à u est de longueur $\leq k$ et donc pour tout prédécesseur v de u, il existe un chemin d'une racine à v de longueur $\leq k-1$. Les tâches v ont donc un chemin critique amont de longeur $\leq k-1$ et sont étiquetées. A l'étape k, on étiquette donc u.
  - Sinon, u_{p-1} a un chemin critique amont de longueur $p-1 \geq k$ et n'est donc pas encore étiqueté. u ne le sera donc pas (tous ses prédécesseurs n'étant pas marqués).

  On a donc prouvé le résultat au rang k.

Q.9 Si le graphe est acyclique, les questions **7** et **8** montrent que toutes les tâches finissent par avoir une étiquette (et même de valeur $\leq n-1$, où n est le nombre de tâches). L'algorithme se termine et la boucle est effectuée au plus $n-1$ fois.
Réciproquement (et en contraposant) supposons le graphe cyclique. L'énoncé a montré qu'il existe une tâche n'yant pas de chemin critique amont et un tel sommet ne sera jamais étiqueté ce qui empêchera la boucle de se terminer. Prenons l'exemple du graphe



A l'étape 0, la tâche a est étiquetée 0. Mais ensuite, aucune autre tâche ne sera plus étiquetée car elle possèdera toujours un prédécesseur non étiqueté.

Q.10 Soit u une tâche d'étiquette k. Elle possède donc un chemin critique amont $(u_0, \dots, u_k = u)$. Prouvons par récurrence sur i que u_i ne pourra être ordonnancée que i unité de temps après la tâche u_0 (qui est une racine et est donc étiquetée en premier lieu).

- Initialisation : c'est immédiat pour $i = 0$.
- Hérédité : soit $i \in [0, k-1]$ tel que le résultat soit vrai aux rangs $0, \dots, i-1$. u_{i-1} ne peut donc

être ordonnancé que i unité de temps après u_0. Comme $u_{i_1} \to u_i$, on a $\sigma(u_{i-1}) + \delta(u_{i-1}) \leq \sigma(u_i)$ et comme $\delta(u_{i-1}) = 1$, on en déduitv le résultat au rang i.

Q.11 Je choisis ici de ne pas suivre l'indication et donc de ne pas écrire de fonctions auxiliaires. On commence par obtenir le tableau `tab` des tâches et on gère

- une référence `c` indiquant le nombre de tâches déjà traitées ;
- une référence k indiquant le numéro du prochain lot T_k à traiter.

On doit donc continuer tant que `c`$\neq n$ (n étant le nombe des tâches). Une étape de la boucle `while` va correspondre à l'écriture de toutes les tâches du lot T_k. Dans la boucle, on gère une référence `uti` indiquant le nombre de processeurs utilisés (c'est à dire de tâche inscrites sur la ligne courante). On parcourt alors tout le tableau `tab`. Quand on rencontre une tâche d'étiquette `k`,

- si on est arrivé au bout de la ligne (`uti`$= p$), on en commence une nouvelle ;
- on imprime la tâche (en mettant à jour nos références)

Il ne faut pas oublier, en fin d'une étape de la boucle `while`, d'incrémenter k. On n'oublie pas non plus les `Begin` et `End`.

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

Remarque : pour éviter de multiples parcours de `tab`, on pourrait créer une table `T` telle que `T.`(k) contienne la liste des noms des tâches d'étiquette k. Il faut alors écrire une fonction auxiliaire prenant en argument une liste de noms de tâches et imprimant correctement ces noms. Pour gérer les sauts de ligne, la fonction écrite est de signature `ecrire : string list` $\to$ `int` $\to$ `unit`, le second argument indiquant combien d'éléments ont déjà été écrits sur la ligne courante.

Remarque : j'ai supposé (comme dans l'énoncé, que le graphe était déjà étiqueté c'est à dire que la fonction `label_height` lui avit été appliquée).

```
let schedule_height g p =
    let tab=get_tasks g in
    let n=vect_length tab in
    let T=make_vect n [] in
```

```
for i=0 to n-1 do
    let j=get_tag tab.(i) in
    T.(j) <- (get_name tab.(i))::T.(j)
    done;
let rec ecrire l j =
   match l with
   [] -> ()
  |nom::q -> if j=p then begin
                         print_newline();
                         ecrire l 0
                         end
                   else begin
                         print_string nom; print_string" ";
                         ecrire q (j+1)
                         end ;
in
let k =ref 0 in
print_string "Begin";print_newline();
while !k<n && T.(!k)<>[] do
             ecrire T.(!k) 0;
             print_newline();
             incr k
             done;
print_newline();print_string "End" ;;
```

Q.12 On néglige dans cette question l'appel à `get_tasks`.

Dans la première version, on parcourt au maximum n fois le tableau des tâches età chaque passage on effecue un nombre constant d'opérations par tâches. On a donc une complexité $O(n^2)$.

Dans la second version, un unique passage dans `tab` est utile et on a une complexité $O(n)$ (compensé par une complexité spatiale plus grande puisque l'on gère un second tableau `T`).

Q.13 Si $p = 1$, il y a une tâche par ligne et donc n lignes. On obtient un ordonnancement de durée totale n ce qui est optimal (il y a au plus une tâche exécutée par instant puisqu'il y a un unique processeur).

Q.14 Pour le graphe ($4a$), chaque tâche est étiquetée par sa hauteur dans l'arbre et l'algorithme 2 donne l'ordonnancement suivant

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

Il n'est pas optimal car l'ordonnancement suivant est meilleur :

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

Pour le graphe (4b), l'algorithme donne

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

Il n'est pas optimal car l'ordonnancement suivant est meilleur :

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

Ordonnancement par profondeur : l'algorithme de Hu

Q.15 Lors d'une étape, on repère les successeurs au lieu des prédécesseurs. On écrit donc une fonction similaire à celle de la question **5** puis on adapte.

```
let check_tags_successors t g =
   let pred=get_successors t g in
   let n=vect_length pred in
   let i=ref 0 in
   while !i<n && (has_tag pred.(!i)) do incr i done ;
   !i=n ;;

let label_depth g =
   let tab=get_tasks g in
   let n=vect_length tab in
   let rec reperer i =
      if i=(-1) then []
      else begin
            if (not (has_tag tab.(i))) && (check_tags_successors tab.(i) g)
                then tab.(i)::(reperer (i-1))
            else reperer (i-1)
            end in
    let rec etiqueter avoir k =
       match avoir with
       [] -> ()
      |t::q -> set_tag k t ; etiqueter q k in
    let rec iterer k =
        let avoir=reperer (n-1) in
        if avoir<>[] then begin
              etiqueter avoir k;
              iterer (k+1)
              end
     in iterer 0 ;;
```

Q.16 Soit $(u_0, \ldots, u_h)$ un chemin critique amont de $u = u_0$. On a alors $\sigma(u_{i+1}) \geq \sigma(u_i) + 1$ et

donc, par récurrence simple,

$$\sigma(u_i) \geq i + \sigma(u_0)$$

en pariculier, $\sigma(u_h) \geq h + \sigma(u) = h + t$. Comme la tâche u_h s'exécute en un temps de 1, on ne terminera pas avant le temps $h + t + 1$.

Q.17 Pour le graphe (4a) on a les évolutions suivantes (on donne les éléments de R_t par ordre décroissant de profondeur)

t	R_t	tâches
1	a	a
2	b, c, d, e	b, c
3	f, g, d, e	f, g
4	h, d, e	h, d
5	i, e	i, e

ce qui donne une durée totale d'exécution égale à 5.

Remarque : l'algorithme proposé n'est en fait pas déterministe car au temps t on peut alors $r > p$ tâches de même profondeur et il faut faire un choix.

Pour le graphe (4b) on a les évolutions suivantes (on donne les éléments de R_t par ordre décroissant de profondeur)

t	R_t	tâches
1	a, d, g, h	a, d
2	b, f, g, h	b, f
3	c, g, h	c, g
4	e, h	e, h
5	i	i

ce qui donne une durée totale d'exécution égale à 5.

Q.18 La fonction est en tous points semblable à celle de la question 5. J'ai supposé, là encore, que l'énoncé avait oublié de donner l'argument `g` à la fonction.

```
let is_ready t g =
   let pred=get_predecessors t g in
   let n=vect_length pred in
   let i=ref 0 in
   while !i<n && (get_state pred.(!i)=Done) do incr i done ;
   !i=n ;;
```

Q.19 Comme en question 11, on suppose ici que l'on travaille avec un graphe correctement étiqueté (on suppose lui avoir appliqué la fonction `label_depth`).

Dans toute ma fonction, je vais travailler avec le tableau `tab` des tâches non exécutées. Celui-ci est obtenu initialement par `get_tasks` mais va évoluer. La première fonction effectue une mise à jour : les taches effectuées sont supprimées du tableau (on les transforme en des tâches vides) et celles prêtes changent d'état.

```
let mise_a_jour tab g =
   let n=vect_length tab in
   for i=0 to n-1 do
        if is_empty_task tab.(i) then ()
        else if get_state tab.(i)=Done then tab.(i) <- empty_task
        else if get_state tab.(i)=Init && (is_ready tab.(i) g)
                then set_state Ready tab.(i)
        done ;;
```

Dans la fonction principale, on gère un compteur `c` du nombre de tâches effectuées. On a donc terminé quand `c`$= n$. Une étape de la boucle principale correspond à une étape de l'algorithme :

- mise à jour du tableau des tâches ;
- création du tableau ordonné des tâches restantes (par étiquette décroissante) ;

- traitement des au plus p tâches exécutables.

Pour le troisième point, on gère une référence `i` qui correspond à la position dans le tableau des tâches et une autre `j` égale au nombre de tâches traitées lors de l'étape (qui ne doit pas excéder p). Le fait de travailler avec un tableau trié de tâches assure le respect de l'algorithme de Hu.

```
let schedule_Hu g p =
    let tab= get_tasks g in
    let n=vect_length tab in
    let c=ref 0 in
    print_string "Begin";
    while !c<n do
        mise_a_jour tab g;
        let t=sort_tasks_by_decreasing_tags tab in
        print_newline();
        let j=ref 0 in
        let i=ref 0 in
        while (!i<n) && (!j<p) do
              if not(is_empty_task t.(!i)) && get_state t.(!i)=Ready then
                     begin
                      print_string (get_name t.(!i));
                      print_string " ";
                      set_state Done t.(!i);
                      incr j;
                      incr c;
                      end;
              incr i
              done;
         done ;
     print_newline();print_string "End" ;;
```

Q.20 La boucle principale `while` est effectuée au plus n fois. La mise à jour se fait en $O(n)$ et le tri en $O(n\log(n))$. La boucle intérieure `while` est en $O(n)$. Le coût total est $O(n^2\log(n))$.

Remarque : comme en question **11** on aurait sans doute pu éviter des trajets inutiles par utilisation de listes. Mais il aurait alors sans doute fallu avoir une fonction de tri de liste de tâches ou d'insertion dans une liste triée de tâches.

Q.21 Dans un graphe arborescent entrant, effectuer une tâche libère au plus une tâche (son éventuel unique successeur). Quand on retire un élément à R_t, on en ajoute donc au plus un à R_{t+1}. Ceci explique que le cardinal de R_t ne peut croître au cours de l'algorithme.

Q.22 Avec le graphe (6) l'algorithme de Hu donne

t	R_t	tâches
1	a,b,c	a,b
2	c	c
3	d,e,f	d,e
4	f,g,h	f,g
5	h	h

On peut cependant conclure en 4 unité de temps avec l'ordonnancement suivant ;

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

Q.23 Notons T_i l'ensemble des tâches de profondeur $\geq i$. Soit t_i le plus petit temps où tous les éléments de T_i peuvent avoir été exécutés. Comme on dispose de p processeur, on a donc $t_i \geq \left\lceil \frac{|T_i|}{p} \right\rceil$ (où $|E|$ désigne le cardinal d'un ensemble E). Par minimalité, au temps t_i on a exécuté une tâche de profondeur $\geq i$ et il nous faut donc encore au moins un temps i en plus (pour terminer le chemin vers une feuille). Ceci montre que tout ordonnacement à un temps d'exécution totale plus grand que $\left\lceil \frac{|T_i|}{p} \right\rceil + i$. Si on note d la profondeur maximale d'une tâche, on a aussi un temps d'exécution totale plus grand que $d+1$. Finalement, le temps minimal pour un ordonnancement est plus grand que

$$\max\left(d+1, \max_{0\leq i\leq d}\left(\left\lceil \frac{|T_i|}{p} \right\rceil + i\right)\right)$$

où d est la profondeur maximale d'une tâche.
On pourra conclure que l'algorithme de Hu est optimal si on montre qu'il réalise ce minorant. Considérons donc l'ordonnancement obtenu par cet algorithme (dans le cas d'un graphe entrant) et distinguons deux cas.

- Si tous les processeurs sont toujours actifs sauf éventuellement lors de la dernière étape, il est immédiat que l'ordonnancement est optimal.
- Sinon, on considère le premier temps t où au moins un processeur est inactif. Aux temps $1, \dots, t-1$, les p processeurs ont tous fonctionné (conséquence de la question **21**, c'est là que l'on utilise l'hypothèse graphe en trant). Puisque l'on a perdu une tâche disponible au temps t, il existe une tâche T effectuée au temps $t-1$ dont le successeur n'a pas été libéré au temps t. Notons i la profondeur de la tâche T. Au temps t, toutes les tâches de rang i seront terminées (sinon, une tâche de rang $i+1$ n'aurait pas été exécutée au temps $t-1$ et cela nie le choix dans l'algorithme de Hu). On est donc exactement dans le cadre décrit en début de question et le temps d'exécution est $\left(\left\lceil \frac{|T_i|}{p} \right\rceil + i\right)$.

Remarque : évidemment, les explications mériteraient d'être un peu plus détaillées mais je crois que les idées principales sont présentes.