Ordonnancement de graphes de tâches

1 Opérations et résultats élémentaires

1. Pour chaque lien $u \to v$ dans G, on souhaite créer un lien $v \to u$ dans G′. On va donc créer un tableau de taille n contenant des listes vides, et on va les peupler au fur et à mesure en parcourant les listes décrivant G.

```
let transpose gr =
  let n = Array.length gr in
  let gr_tr = Array.make n [] in
  for i=0 to n-1 do
    List.iter
      (fun j -> gr_tr.(j) <- i::gr_tr.(j))
      gr.(i)
  done;
  gr_tr
```

2. La i^e case du tableau obtenu en appliquant la fonction fonction `transpose` contient la liste des prédécesseurs (immédiats) de la tâche `i`.

3. Numérotons les tâches du chemin $u \rightsquigarrow v$ en écrivant $u = u_0 \to u_1 \to u_2 \to \ldots \to u_{k-1} \to u_k = v$. Puisque $u_i \to u_{i+1}$ impose $\sigma(u_i) < \sigma(u_{i+1})$, on a donc $\sigma(u) = \sigma(u_0) < \sigma(u_1) < \sigma(u_2) < \ldots < \sigma(u_{k-1}) < \sigma(u_k) = \sigma(v)$. Et donc (en admettant qu'il ne s'agit pas d'un chemin réduit à un seul sommet, soit $k > 0$), conduit à $\sigma(u) < \sigma(v)$.

4. Tout chemin $u \rightsquigarrow v$ de longueur supérieure ou égale à 1 vérifie $\sigma(u) < \sigma(v)$, ce qui impose naturellement $u \neq v$. Il ne peut donc pas y avoir de chemin fermé, donc de cycle dans G.

5. Prenons un sommet quelconque u_0 dans G, et appliquons l'algorithme suivant : tant que u_i possède au moins un successeur, on choisit u_{i+1} parmi ces successeurs (peu importe la façon dont on le choisit, on peut par exemple prendre la tâche de plus petit numéro). G étant acyclique, tous les u_i sont distincts. Comme G possède un nombre fini de sommets, la suite des sommets u_i ainsi construite ne peut être infinie. Le dernier des éléments de la suite n'a donc pas de successeur, c'est donc une feuille.

Pour le cas de la racine, on procède de même en choisissant u_{i+1} parmi les prédécesseurs de u_i (ou, alternativement, on peut dire que si G est acyclique, G′ l'est également, qu'il existe donc une feuille dans G′, et que cette feuille est une racine de G).

On évitera, dans la mesure du possible, d'abuser de preuves par l'absurde quand il est possible d'offrir des preuves constructives, plus simples à lire en général !

2 Ordonnancement par hauteur

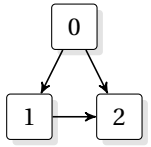
6. On construit simplement le tableau des longueurs des listes des antécédants de `gr` :

```
let nb_dependances gr =
  Array.map List.length (transpose gr)
```

7. On cherche les indices des cases du tableau obtenu précédemment contenant 0, par exemple :

```
let racines gr =
  let n = Array.length gr in
  let res = ref [] in
  let nb_dep = nb_dependances gr in
  for i = 0 to n-1 do
    if nb_dep.(i) = 0
      then res := i :: !res
  done;
  !res
```

8. On a ici une fonction un peu plus élaborée qui mérite à la fois du soin et des explications. **Il ne s'agit pas, pour des valeurs de k successives, de regarder chacun des sommets, et si leur dépendances sont étiquetées, de leur attribuer l'étiquette k**. En effet, pour le graphe ci-dessous, un tel algorithme attribuerait l'étiquette 0 à toutes les tâches, car lorsque l'on traite la tâche 1, sa dépendance est déjà étiquetée, et lorsque l'on traite la tâche 2, ses deux dépendances sont aussi déjà étiquetées !



On peut imposer que les dépendances soient étiquetées *avec une étiquette strictement inférieur à l'étiquette courante*. Mais on peut essayer de lire entre les lignes est regarder ce que le sujet nous suggère à demi-mots.

La fonction `racines` sert à trouver les sommets qui porteront l'étiquette 0. Pour la suite, on pourrait retirer ces sommets (et les dépendances qui y sont liées) et déterminer les racines non étiquetées encore du graphe ainsi obtenu, mais ce serait vite coûteux.

Le calcul du nombre de dépendances donne le second morceau du puzzle : une tâche devient une racine quand son nombre de dépendances non satisfaites atteint zéro. Or il

est aisé de décrémenter le nombre de dépendances des taches qui suivent une tâche que l'on vient d'étiqueter.

La dernière pièce du puzzle, si l'on souhaite obtenir une bonne complexité, est de garder en permanence les tâches non étiquetées sans dépendances, pour pouvoir les étiqueter à l'itération suivante. Il s'agit en quelque sorte d'un parcours en largeur, mais avec la contrainte toutefois de n'atteindre un sommet que par le *dernier* des arcs qui y mène.

Bien qu'il est possible de construire une fonction purement fonctionnelle, considérons tout d'abord une approche impérative. On utilise une référence `k` indiquant l'itération courante, et une référence `ready` contenant la liste des tâches non encore étiquetés au début de l'étape k et sans dépendance, qui seront donc étiquetées k (référence initialisée grâce à `racines`). Un tableau `nb_dep` mémorise le nombre de dépendances restantes pour chaque tâche, et `etiq` les étiquetages déjà effectués (et -1 pour ceux encore à déterminer).

À chaque itération, on prépare la liste des tâches à étiqueter à l'étape suivante ($k+1$), mémorisée dans la référence `next_ready`. Pour chaque tâche i à étiqueter à l'étape k, on l'étiquète (1) et on supprime la dépendance pour chaque tâches dépendante de i (2) avec un appel à une fonction `remove_dep` décrémentant le compteur de dépendances de j (3), et l'ajoutant à `next_ready` si ce compteur atteint 0 (4). Cela donne donc :

```
let etiquete gr =
  let n = Array.length gr in
  let nb_dep = nb_dependances gr in
  let etiq = Array.make n (-1) in
  let ready = ref (racines gr) in    (* Taches sans prédécesseurs *)
  let k = ref 0 in                                 (* non étiquetés *)

  while !ready <> [] do
    let next_ready = ref [] in
    let remove_dep j =
      nb_dep.(j) <- nb_dep.(j)-1;                              (* 3 *)
      if nb_dep.(j) = 0
        then next_ready := j :: !next_ready                    (* 4 *)
    in List.iter
         (fun i -> etiq.(i) <- !k;                             (* 1 *)
                   List.iter remove_dep gr.(i))                (* 2 *)
         !ready;
    ready := !next_ready;
    incr k
  done;
  etiq
```

On peut imaginer beaucoup de variantes sur le même principe, par exemple cette implémentation où une fonction récursive `explore` de signature `int -> int list -> int array` prenant en argument k et une liste des tâches à étiqueter k remplace la boucle for, et retourne le tableau d'étiquettes lorsque la récursion se termine :

```
let etiquete gr =
  let n = Array.length gr in
  let nb_dep = nb_dependances gr in
  let etiq = Array.make n (-1) in

  let rec explore k = function
    | [] -> etiq
    | ready ->
        List.iter (fun i -> etiq.(i) <- k) ready;    (* 1 *)
        let successors =
          List.concat (List.map (fun i -> gr.(i)) ready) in
        let next_ready =
          List.filter
            (fun i -> nb_dep.(i) <- nb_dep.(i)-1;    (* 3 *)
                      nb_dep.(i) = 0)                (* 4 *)
            successors
        in explore (k+1) next_ready
  in explore 0 (racines gr)
```

9. Le calcul des dépendances est en $O(n+|\mathrm{D}|)$, la construction du tableau en $O(n)$, et la boucle principale explorant le graphe en $O(n+|\mathrm{D}|)$ puisque l'on effectue des opérations en $O(1)$ pour chaque dépendance (décrémentation et test) et chaque tâche (entrée dans `next_ready`, étiquetage). La complexité est donc en $O(n+|\mathrm{D}|)$.

10. Pour montrer qu'une tâche u admet un chemin critique amont, il suffit de montrer que P_u est non vide et que tous les chemins de P_u, dont la longueur est entière, ont une longueur majorée par une constante, ce qui assure l'existence d'un maximum.

Pour montrer que P_u est non vide, on construit un chemin menant d'une racine de G à la tâche u comme à la question 5, en choisissant itérativement des prédécesseurs jusqu'à parvenir à une racine. Ensuite, il suffit de justifier que les chemins étant sans cycles, la longueur de ces chemins est majorée par le nombre de tâches n.

Notons que l'on peut, alternativement, montrer que P_u est non vide et de cardinal fini, la deuxième condition pouvant être justifiée en remarquant que G étant acyclique, $|\mathrm{P}_u|$ est majoré par $\sum_{k=0}^{n-1} \mathrm{A}_k^{n-1}$, où A_k^{n-1} est le nombre d'arrangements de k éléments parmi $n-1$.

11. Nous allons démontrer l'équivalence suivante, attendue par l'énoncé, par une récurrence forte sur k.

H_k *: la longueur des chemins critiques amont de u est k* $\iff$ *u est étiqueté par k*

L'équivalence est vraie pour $k = 0$: les racines du graphe de tâches sont étiquetées 0 par construction, et les seules tâches étiquetées 0 sont les racines.

Supposons l'équivalence vraie pour tout entier inférieur ou égal à k, et montrons l'équivalence entre le fait d'attribuer étiquette $k+1$ et le fait que les longueurs des chemins critiques amont d'un sommet soit $k+1$. On va montrer une double implication.

① la longueur des chemins critiques amont de u est $k+1 \implies u$ est étiqueté par $k+1$

Considérons tout d'abord un sommet u quelconque dont la longueur des chemins critiques amont est $k+1$. Il n'a pas été étiqueté (sinon, il aurait été étiqueté avec une étiquette k' inférieure ou égale à k puisqu'elles sont attribuées de façon croissante, or cela signifierait d'après l'hypothèse de récurrence que la longueur des chemins critiques amont serait k').

Considérons les prédécesseurs de u. Leurs chemins critiques amont ont tous une longueur inférieure ou égale à k (s'il existe un v pour lequel ce n'est pas le cas, on peut considérer le chemin $v_1 \to v_2 \to \ldots \to v \to u$ de longueur strictement supérieure à $k+1$, ce qui n'est pas compatible avec notre hypothèse sur u). D'après l'hypothèse de récurrence, tous ces sommets ont été étiquetés, donc le sommet u pourra bien recevoir une étiquette $k+1$.

② u est étiqueté par $k+1 \implies$ la longueur des chemins critiques amont de u est $k+1$

Inversement, supposons que u soit étiquetés $k+1$. Il n'a pas pu être étiqueté à l'étape k, et l'a été à l'étape $k+1$. Un (au moins) de ses prédécesseurs a donc nécessairement été étiqueté à l'étape k. Notons ce prédécesseur v. D'après l'hypothèse de récurrence, la longueur des chemins critiques amont de v est k, donc il existe un chemin $v_1 \to v_2 \to \ldots \to v$ de longueur k. Et naturellement un chemin $v_1 \to v_2 \to \ldots \to v \to u$ de longueur $k+1$. La longueur des chemins critiques amont de u est donc supérieure ou égale à $k+1$.

Mais elle ne peut être strictement supérieure à $k+1$, car s'il existait un chemin menant d'une racine à u de longueur strictement supérieure à $k+1$, alors l'avant dernier sommet sur ce chemin, un prédécesseur de u, aurait une longueur strictement supérieure à k, autrement dit ce prédécesseur n'aurait pas pu être déjà étiqueté (et par conséquent u ne pourrait pas être étiqueté à l'étape $k+1$). La longueur des chemins critique amont de u est donc exactement $k+1$.

L'hypothèse de récurrence permet de conclure.

12. Il faut ici impérativement établir un lien clair avec l'algorithme (et éventuellement avec son implémentation). On souligne ainsi que dans notre algorithme, k augmente à chaque itération. Chaque tâche est donc étiquetée à une itération bien particulière de l'algorithme, d'après la question précédente, définie par la longueur de son chemin amont.

Après au plus n itérations, toutes les tâches sont nécessairement étiquetées puisque aucun chemin amont ne peut avoir une longueur supérieure à n. L'algorithme termine donc.

Si le graphe contient un cycle, alors l'algorithme ne pourra pas étiqueter toutes les tâches. Mais son comportement dépend de choix d'implémentation : il peut terminer avec des tâches non étiquetées, ou boucler indéfiniment. Avec nos choix d'implémentation, on finira par avoir une liste vide associée à `ready`, donc l'algorithme *termine*, mais certaines tâches auront -1 pour étiquette.

13. On a vu que si $u \to v$, alors $\sigma(u) < \sigma(v)$. Mais comme les tâches portent des étiquettes entières et positives, on a nécessairement également $\sigma(u) + 1 \leqslant \sigma(v)$.

Si une tâche u porte une étiquette k, c'est qu'il existe un chemin amont $\sigma(u_0) \to \sigma(u_1) \to \sigma(u_2) \to \ldots \to \sigma(u_{k-1}) \to \sigma(u_k) = \sigma(u)$ y menant.

Avec l'égalité précédente, on en déduit $\sigma(u_0) + k \leqslant \sigma(v)$. Mais comme $\sigma(u_0) \geqslant 0$ par définition, on a nécessairement $\sigma(v) \geqslant k$.

2.1 Construction de l'ordonnancement

14. Afin de faciliter la lecture de ce que l'on va écrire, il est bienvenu de décomposer l'algorithme autant que possible.

Commençons par définir une fonction retournant le plus grand élément d'un tableau ne contenant que des entiers positifs :

```
let rec max_array =
  Array.fold_left max 0
```

Ensuite, on écrit une fonction prenant un étiquetage (sous la forme d'un tableau) et retournant un tableau de listes de taille $k_{\max} + 1$ tel que la k^{e} case de ce tableau contient toutes les tâches portant l'étiquette k :

```
let partition etiq =
  let steps = Array.make (max_array etiq + 1) [] in
    for i = 0 to Array.length etiq - 1 do
      steps.(etiq.(i)) <- i::steps.(etiq.(i))
    done;
  steps
```

L'algorithme suggère de traiter les listes du tableau obtenu avec la liste précédente une à une, ce qu fait la boucle for de la fonction ci-dessous. La fonction `label` est ainsi appelée avec chacune des listes. Elle prend deux arguments : un entier `avail` indiquant combien de processeurs sont encore libres à l'instant courant (toujours désigné par `time`) et la liste des tâches portant l'étiquette considérée restant à étiqueter.

On notera que l'on commence avec 0 processeurs disponibles. **Il faut faire attention, si le nombre de tâches portant une étiquette k divisible par p, que `time` ne soit pas incrémenté deux fois de suite** (quand on atteint p processeurs occupés et quand on passe

à l'étiquette $k+1$). Pour éviter ce problème, on a choisi ici de n'incrémenter `time` que quand on n'a plus de processeurs de libres, et on commence, pour une étiquette k, avec aucun processeur de libre, ce qui forcera l'incrémentation de `time` lorsque l'on passe de l'étiquette k à l'étiquette $k+1$. Mais puisque `time` sera alors incrémentée avant le choix de la première étiquette, on le fait débuter à -1.

```
let ordonne gr p =
  let steps = partition (etiquete gr)
  and ord = Array.make (Array.length gr) (-1)
  and time = ref (-1) in

  let rec label avail = function
    | [] -> ()
    | lst when avail = 0 -> incr time; label p lst
    | t::q -> ord.(t) <- !time; label (avail-1) q
  in

  for i = 0 to Array.length steps - 1 do
    label 0 steps.(i)
  done;
  ord
```

15. La boucle **for** de la fonction `ordonne`, en incluant l'appel à `label`, effectue des opérations en $O(1)$ sur chaque tâche. La complexité est donc $O(n+k_{\max})$, mais comme $k_{\max} \leqslant n$ (pour toute étiquette k vérifiant $0 \leqslant k \leqslant k_{\max}$, il y a forcément au moins une tâche portant cette étiquette), cela se réduit simplement à $O(n)$.

`partition` et la construction du tableau `ord` sont tous deux en $O(n)$. Quant à `etiquete`, on a vu que la complexité était en $O(n+|\mathrm{D}|)$.

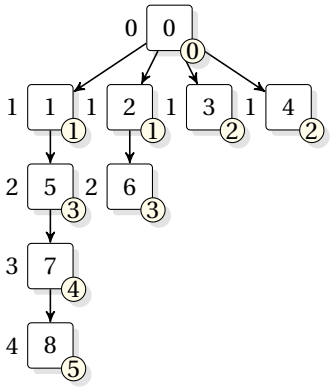
La fonction `ordonne` a donc une complexité en $O(n+|\mathrm{D}|)$ qui vient directement de l'étape d'étiquetage.

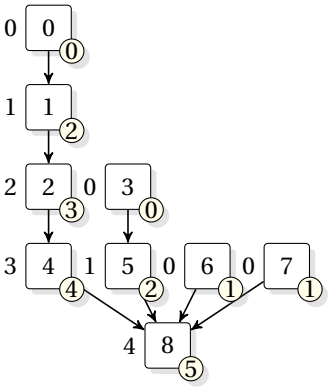
16. Il est impossible, avec l'algorithme proposé, d'avoir un instant où le processeur n'est pas actif. En effet, il ne peut y avoir de « trou » dans l'ensemble des étiquettes (si une tâche u porte une étiquette $k>0$, alors il y a forcément une étiquette $k-1$, il suffit de considérer les prédécesseurs de la tâche u dans les chemins critiques amont de u).

Donc le processeur est actif à tous les instants jusqu'à ce que toute les tâches soient effectuées, ce qui donne une durée totale d'ordonnancement égale au nombre de tâches, et pour $p=1$, il s'agit nécessairement d'une durée optimale.

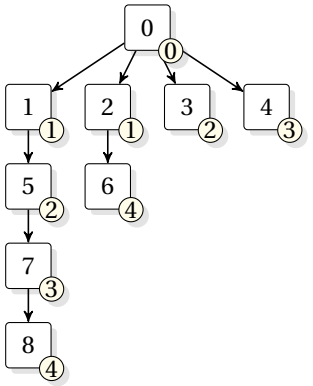
2.2 Limitations

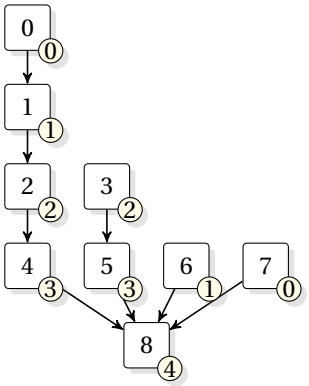
17. L'algorithme 2 conduit par exemple aux ordonnancements suivants dans le cas $p=2$ processeurs :





18. Aucun des deux n'est optimal :





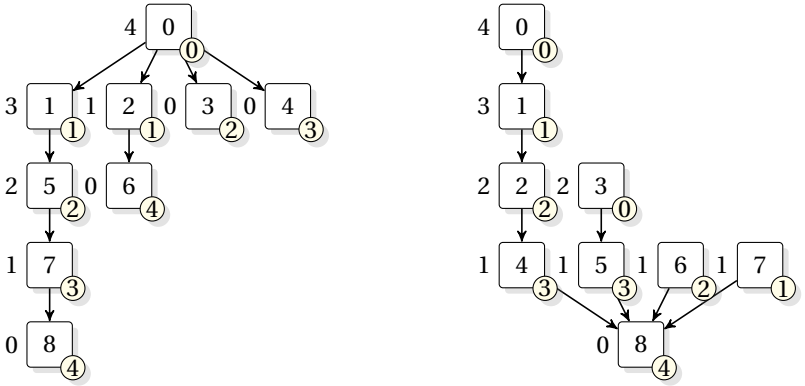
3 Algorithme de Hu

19. On utilise simplement le retournement du graphe pour parcourir les liens dans l'autre sens :

```
let etiquete_bis gr =
  etiquete (transpose gr)
```

20. Si une tâche a une profondeur $\mathrm{depth}(u)=k$, alors il existe un chemin $u \to v_1 \to v_2 \to \ldots \to v_k$ de longueur k. On a $\sigma(u) < \sigma(v_1) < \sigma(v_1) < \ldots < \sigma(v_k)$, ce qui conduit, puisque les $\sigma(v)$ sont entiers, à $\sigma(v_k) \geqslant \sigma(u)+k = t+\mathrm{depth}(u)$. La durée totale d'ordonnancement excède donc nécessairement $t+\mathrm{depth}(u)$.

21. Cela donne, par exemple :



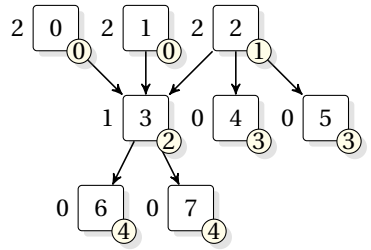
On remarquera que, cette fois, l'algorithme de Hu a fourni des ordonnancement optimaux pour les graphes de tâches considérés !

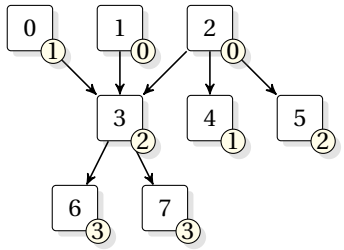
22. Si une tâche non étiquetée ne se trouve pas dans R lors de la i^{e} itération de l'algorithme mais qu'elle est présente à l'itération suivante, c'est nécessairement que son prédécesseur a étiqueté lors de la i^{e} itération.

Or, dans le cas d'un graphe de tâches entrant, toute tâche a au plus un seul successeur. L'étiquetage d'une tâche ne peut donc libérer, et par conséquent faire entrer dans R, qu'une seule tâche. Si n tâches sont étiquetés lors de la i^{e} itération, alors $n' < n$ tâches entrent dans R pour l'itération suivante. Au bilan, le cardinal de R évolue de $n' - n \leqslant 0$, donc diminue nécessairement à chaque itération.

23. L'algorithme de Hu peut ordonner les tâches comme ci-après à gauche, avec un instant $t = 1$ où l'un des deux processeurs est inactif (et de même ensuite à $t = 2$). Cela conduit à une durée totale d'ordonnancement de 5.

Pourtant, une durée totale de 4 est possible, comme sur l'exemple ci-après à droite (qui peut également être le résultat obtenu avec l'algorithme de Hu, cela dépend quelles tâches sont choisies pour la première itération parmi toutes les tâches de la première ligne, et qui portent toutes la même étiquette).





Résultats