X/ENS Informatique A MP 2015 — Corrigé

Ce corrigé est proposé par Benjamin Monmege (enseignant-chercheur à l'université) ; il a été relu William Aufort (professeur en CPGE).

Ce sujet traite de l'ordonnancement d'un ensemble de tâches : si certaines ne peuvent être faites qu'après d'autres, dans quel ordre faut-il les exécuter ? Si chacune est associée à une durée, comment les répartir entre des agents travaillant en parallèle afin de minimiser la durée totale ? Ce problème a des application concrètes dès que l'on travaille en équipe, donc notamment en entreprise (dans quel ordre faut-il construire un avion ?), mais aussi dans nos ordinateurs quand différentes applications s'exécutent en parallèle sur plusieurs processeurs. Le sujet modélise le problème à l'aide d'un graphe de dépendances ayant les tâches pour sommets et les relations de dépendances entre tâches comme arcs orientés (un arc (u, v) signifie que la tâche u doit être terminée avant que la tâche v ne puisse commencer). Le sujet se compose de trois parties.

- Dans la première partie, le modèle de graphes de tâches est étudié. En particulier, on y montre qu'on ne peut produire un ordonnancement que dans les graphes de tâches acycliques. Deux algorithmes sont alors étudiés dans les deux parties suivantes.
- Un algorithme d'ordonnancement par hauteur est proposé ; il s'exécute en temps linéaire mais ne produit à coup sûr des ordonnancements optimaux que dans le cas trivial d'un seul processeur.
- La dernière partie étudie l'algorithme de Hu, utilisant des calculs plus élaborés de profondeurs, afin de garantir l'optimalité sur une classe de graphes de tâches ayant la forme d'arbres, quel que soit le nombre de processeurs. Cette optimalité s'obtient au prix d'une complexité quadratique, plutôt que linéaire.

Ce sujet plutôt long alterne des questions d'implémentation (dont certaines sont délicates et demandent aux candidats de structurer eux-mêmes leur code) et des démonstrations, en particulier pour prouver la correction ou la complexité des algorithmes étudiés. C'est donc un sujet très riche qu'il est intéressant d'étudier pour réviser les graphes et la programmation tant impérative que récursive. Il est à noter qu'une librairie de graphes, spécifique à cette épreuve, est imposée, ce qui permet de s'extraire de détails bas niveau.

Dans le corrigé, les codes sont écrits en OCaml plutôt qu'en Caml Light, comme le demandait pourtant l'énoncé, car c'est le langage qui est devenu obligatoire aux concours depuis la session 2019.

Indications

Partie II

1 Suivant l'indication de l'énoncé, montrer la propriété

$$\mathrm{H}(n) : \quad \text{« s'il existe un chemin de dépendance de longueur } n \text{ d'une tâche } u \text{ vers une tâche } v \text{ dans G, alors } \sigma(u) + n \leqslant \sigma(v) \text{ »}$$

par récurrence sur $n \in \mathbb{N}$.

2 En raisonnant par l'absurde, montrer que si le graphe ne contient pas de racine, il est possible de construire un chemin de longueur $n+1$, avec n le nombre de tâches du graphe. Pour la propriété sur les feuilles, considérer le graphe dual où le sens des arcs est inversé.

3 Utiliser les fonctions `get_tasks` et `get_predecessors` de la librairie mise à disposition.

Partie III

5 Il y a une erreur dans l'énoncé : il faut aussi le graphe en second argument de la fonction `check_tags_predecessors` qui a donc pour type `task -> graph -> bool`.

6 Utiliser une première fonction auxiliaire produisant la liste de tâches prêtes à être étiquetées à l'étape courante. Ensuite, écrire une autre fonction auxiliaire récursive qui étiquette les tâches de cette liste.

7 Pour chaque tâche u, montrer que P_u est non vide en adaptant le raisonnement employé dans la question 2. Montrer ensuite que la longueur maximale d'un chemin de P_u est borné, grâce à l'hypothèse d'acyclicité.

8 Montrer cette propriété par récurrence forte sur k, en considérant, pour une tâche u d'étiquette $k+1$, les étiquettes de ses prédécesseurs.

9 La condition nécessaire est un corollaire des questions 7 et 8. Pour la condition suffisante, supposer l'existence d'un cycle et montrer que l'algorithme ne peut pas terminer, faute de pouvoir étiqueter les tâches du cycle. Utiliser le graphe de la figure 1(c) pour produire un exemple.

10 Montrer cette propriété par récurrence sur k.

11 Commencer par produire un tableau de listes de tâches, dont la i-ième case contient les tâches d'étiquette i. Utiliser ensuite deux fonctions récursives : l'une pour visiter les cases du tableau et l'autre pour parcourir la liste contenue dans cette case (et afficher les tâches).

12 La fonction produisant le tableau de listes de tâches, dont la i-ième case contient les tâches d'étiquettes i, peut être écrite de sorte qu'elle soit de complexité linéaire en n. Le reste de la fonction aussi peut s'écrire avec une complexité linéaire.

14 Il ne s'agit pas seulement d'appliquer l'algorithme 1, mais aussi l'algorithme 2 d'ordonnancement par hauteur... Trouver ensuite deux ordonnancements de durée d'exécution totale de 5 unités de temps.

Partie IV

16 Montrer la propriété par récurrence sur h.

18 Là encore, il manque le graphe comme argument de la fonction `is_ready`, qui aura donc le type `task -> graph -> bool`. Calquer son fonctionnement sur celui de la fonction `check_tags_predecessors`, écrite en question 5.

19 Il faut adapter le code de la fonction `schedule_height` (question 11), en prenant soin de recalculer la liste des tâches prêtes à chaque instant.

20 Le calcul de la liste des tâches prêtes à chaque instant peut se faire avec une complexité $\mathrm{O}(n)$. L'impression des tâches à exécuter sur chaque processeur nécessite un temps $\mathrm{O}(p)$. Borner le nombre de fois que ces instructions sont exécutées pour estimer la complexité.

21 Majorer le nombre de tâches prêtes à être exécutées à l'instant $t+1$, en fonction des tâches prêtes à l'instant t et de celles effectivement exécutées à l'instant t.

23 En notant h la profondeur maximale d'une tâche de G et T_i l'ensemble des tâches de profondeur supérieure ou égale à i (pour $i \in [\![0\,;h]\!]$), commencer par minorer la durée d'exécution totale d'un ordonnancement optimal par

$$\tau_{\mathrm{G}} = \max_{0 \leqslant i \leqslant h} \left(i + \left\lceil \frac{|\mathrm{T}_i|}{p} \right\rceil \right)$$

Montrer ensuite que l'algorithme de Hu produit un ordonnancement de durée inférieure à τ_{G}. Pour cela, considérer le dernier instant k où les p processeurs traitent des tâches ayant toutes la même profondeur d. En notant d' la profondeur maximale d'une tâche restant alors à exécuter, montrer que la durée d'exécution totale de l'ordonnancement calculé par l'algorithme de Hu vaut $k+d'+1$. Montrer aussi que les k premières étapes exécutent toutes p tâches de profondeur au moins d, puis conclure.

II. Graphe de tâches acyclique

1 Montrons par récurrence que la propriété

$\mathscr{P}(n)$: « s'il existe un chemin de dépendance de longueur n d'une tâche u vers une tâche v dans G, alors $\sigma(u) + n \leqslant \sigma(v)$ »

est vraie pour tout $n \in \mathbb{N}$.

- $\mathscr{P}(0)$ est vraie puisqu'un chemin de longueur nulle de u à v implique que $u = v$ et donc $\sigma(u) = \sigma(v)$.
- $\mathscr{P}(n) \Longrightarrow \mathscr{P}(n+1)$: soient u et v deux tâches reliées par un chemin de dépendance de longueur $n+1$. Il existe donc une tâche u_1 telle que $u \to u_1$ et avec un chemin de dépendance de longueur n entre u_1 et v. Par $\mathscr{P}(n)$, $\sigma(u_1)+n \leqslant \sigma(v)$. De plus, l'arc de dépendance entre u et u_1 implique, par la formule (1), que

$$\sigma(u) + \delta(u) \leqslant \sigma(u_1)$$

c'est-à-dire $$\sigma(u) + 1 \leqslant \sigma(u_1)$$

puisque $\delta(u) = 1$. On obtient ainsi que

$$\sigma(u) + n + 1 \leqslant \sigma(u_1) + n \leqslant \sigma(v)$$

ce qui prouve $\mathscr{P}(n+1)$.

- Conclusion : $\forall n \in \mathbb{N}$ $\mathscr{P}(n)$ est vraie

Dans le cas particulier où $n > 0$, cela prouve que

> S'il existe un chemin de dépendance de longueur non nulle d'une tâche u vers une tâche v dans G, alors $\sigma(u) < \sigma(v)$.

S'il existe un cycle $(u_0, u_1, \ldots, u_n)$ avec $u_0 = u_n$, alors $\sigma(u_0) < \sigma(u_0)$. C'est impossible donc

> Si le graphe G admet un ordonnancement, il est acyclique.

2 Soit G un graphe de tâches acyclique. Par l'absurde, supposons que le graphe G ne contienne pas de racine : cela veut donc dire que toutes les tâches ont au moins un prédécesseur. Soit u une tâche quelconque. Par récurrence, on peut donc construire pour tout n un chemin de longueur n de tâche terminale u. Pour n supérieur au nombre de tâches, cela fournit un chemin de dépendances $(u_0, u_1, \ldots, u_n = u)$ comprenant $n+1$ tâches : par le principe des tiroirs, il existe donc deux occurrences u_i et u_j, avec $0 \leqslant i < j \leqslant n$ de la même tâche. Le chemin $(u_i, u_{i+1}, \ldots, u_j)$ est alors un cycle, ce qui contredit l'hypothèse d'acyclicité. Ainsi, le graphe G comporte au moins une racine.

Considérons alors le graphe dual $\mathrm{G}^{-1} = (\mathrm{T}, \mathrm{D}^{-1}, \delta)$ obtenu à partir de G en inversant le sens des arcs, c'est-à-dire en prenant $\mathrm{D}^{-1} = \{(v,u) \mid (u,v) \in \mathrm{D}\}$. Si G est acyclique, il en est de même pour G^{-1}, donc G^{-1} possède au moins une racine : une racine de G^{-1} correspondant à une feuille de G, on a finalement

> Si un graphe de tâche est acyclique, il a au moins une racine et une feuille.