Algorithme et complexité
Projet : Minimisation de l'empreinte mémoire

Consignes de rendu

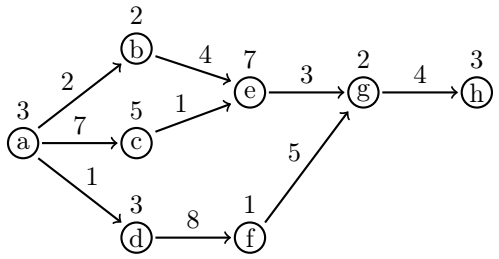
Vous devrez rendre pour ce projet :
— Un rapport.
— Le code source de votre code dans une archive. Le choix du langage est libre.

Le rapport doit contenir une réponse aux questions 1, 2, 8, 12 et 13, ainsi qu'une présentation des différentes structures de données implémentées et des fonctions associées, ainsi que leur complexités. Si vous utilisez des structures de données venant d'une librairie externe vous devrez aussi faire ce même travail en justifiant de chaque complexité. De même les différents algorithmes présentés doivent avoir leur complexité explicitée et justifiée dans le rapport même si ce n'est pas explicitement mentionné dans la question associée. Le rapport doit aussi contenir les instructions (lignes de commandes par exemple) pour compiler et exécuter votre code. Une batterie de tests devra être proposée pour chacune des fonctions principales (on précisera de la même manière les modalités de son exécution).

Si un élément n'est pas clair n'hésitez pas à me contacter par email (t.lambert [at] univ-lorraine.fr) ou à la fin d'un TD ou d'un cours pour me demander des précisions.

Sujet

On s'intéresse à la un problème de minimisation d'empreinte mémoire. Plus précisément on suppose que l'on a un programme dont l'exécution va se faire en plusieurs tâches. Ce programme va être représenté par un graphe dirigé acyclique (DAG) où les sommets représentent les tâches et les arêtes représentent des dépendances, c'est à dire le fait qu'une tâche doive être terminée avant l'exécution de la suivante. Ces DAGs sont par ailleurs pondérés, les arêtes comme les sommets possèdent un poids positif. Le poids d'un sommet représente la quantité de mémoire dont il a besoin pour l'exécution de la tâche associée, le poids d'une arête uv représente la quantité de données produite par la tâche u et qui sera utilisée par la tâche v. On supposera que l'exécution du programme est séquentielle, c'est à dire sur une seule machine/processeur. Par la suite on notera V (et n sa taille) l'ensemble des tâches, E l'ensemble des arêtes (et m sa taille), w_a les poids des arêtes et w_s le poids des sommets.



G, un exemple de DAG pondéré.

On défini ici un **ordonnancement** π comme un parcours de notre DAG respectant ses dépendances. Plus précisément si on note π_i la $i^{\text{ème}}$ tâche exécutée dans l'ordonnancement π, alors si $uv \in E$, $\pi_i = u$ et $\pi_j = v$, on a forcément $i < j$. Sur le DAG G donné en exemple, la tâche c ne peut être exécutée qu'après que la tâche a ait été exécutée, par contre elle peut s'effectuer avant ou après les tâches b, d ou f. La tâche e elle nécessite l'exécution préalable des tâches b et c.

Pour un ordonnancement donné, le **coût d'exécution d'une tâche** correspond à la quantité de mémoire utilisée lors de cette exécution. Plus précisément pour exécuter π_i, on a besoin d'une quantité de mémoire correspondant à :

— $w_s(\pi_i)$ pour l'exécution propre de la tâche.
— Au coût de stockage de toutes les données non encore utilisées (incluant les données d'entrées de la tâche en cours d'exécution).
— Au coût de stockage des données produites par la tâche.

ce qui donne, en notant $c(\pi, i)$ le coût d'exécution de π_i,

$$c(\pi, i) = w_s(\pi_i) + \sum_{\substack{\pi_j\pi_k \in E \\ j<i\leq k}} w_a(\pi_j\pi_k) + \sum_{\pi_i\pi_j \in E} w_a(\pi_i\pi_j)$$

A partir de cette définition on définit le **coût d'exécution d'un ordonnancement** comme son pic de consommation, c'est à dire la plus grande valeur atteinte lors de l'exécution d'une tâche :

$$c(\pi) = \max_{i\in[1,n]} c(\pi, i)$$

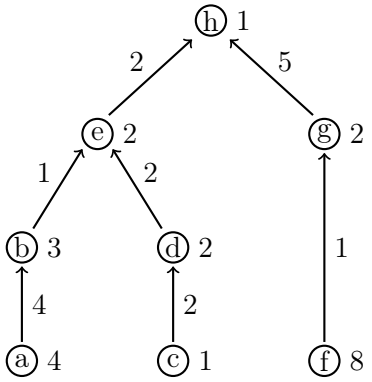
Exemple : en prenant le graphe G et l'ordonnancement $\pi = [a, d, c, f, b, e, g, h]$, on obtient les coût d'exécution suivants :

i	1	2	3	4	5	6	7	8
$c(\pi, i)$	13	21	23	17	14	20	14	7

Le coût d'exécution de π est donc 23.

1. Proposez un ordonnancement de G avec un coût d'exécution de 17.
2. Donnez la définition du problème de décision associé et démontrez son appartenance à NP.

Le cas général de ce problème (un DAG quelconque) étant NP-complet, on va s'intéresser ici à des cas spécifiques de DAGs. On va d'abord considérer les cas des arbres entrants, c'est à dire des arbres enracinés où les arêtes vont systématiquement de l'enfant vers son parent (la racine est donc forcément exécutée en dernière et la première tâche est forcément une feuille).



Un exemple d'arbre entrant A

3. Proposez une structure de donnée pour représenter un arbre entrant.
4. Écrivez une fonction `coutOrdonnancement` qui prend en entrée un arbre entrant et un ordonnancement (dont vous préciserez la représentation) et qui retourne son coût

d'exécution ou -1 si l'ordonnancement n'est pas valide (soit incomplet, soit ne respectant pas les dépendances).

Il existe un algorithme polynomial permettant d'obtenir un ordonnancement avec un coût d'exécution minimal. Pour cela on a besoin de définir deux nouveaux outils : la notion de **"collines"** et de **"vallées"**. Mais avant cela il nous faut définir le **coût résiduel d'une tâche**, c'est à dire l'utilisation de la mémoire une fois la tâche exécutée. La mémoire n'est en effet pas vidée, il reste les données produite par la tâche, mais aussi toutes celles qui n'ont pas été utilisée (les données d'entrée de la tâche sont par contre effacées). Plus précisément, ce coût résiduel cr est définit par :

$$cr(\pi, i) = \sum_{\substack{\pi_j\pi_k \in E \\ j \leq i < k}} w_e(\pi_j\pi_k)$$

En prenant l'arbre A et l'ordonnancement $\pi = [f, a, b, c, d, e, g, h]$, on obtient les coût d'exécution et résiduels suivants :

i	1	2	3	4	5	6	7	8
$c(\pi, i)$	9	9	9	5	8	8	10	8
$cr(\pi, i)$	1	5	2	4	4	3	7	0

On peut maintenant définir la notion de colline et de vallée. On notera h_j^π et v_j^π respectivement les collines et les vallées d'un ordonnancement partiel π et on les définit de la façon suivante :

— h_1^π est le coût d'exécution de l'algorithme.
— v_i^π est le plus petit coût résiduel atteint depuis la dernière fois où h_i^π a été atteint (v_i^π peut être le coût résiduel de la tâche ayant pour coût d'exécution h_i^π).
— h_i^π ($i \geq 2$) est le coût d'exécution maximal atteint depuis la précédente vallée v_{i-1}^π.

La dernière vallée est systématiquement le coût résiduel de la dernière tâche exécutée. En cas d'égalité (plusieurs tâches pouvant être la colline ou la vallée suivante) on prend la plus tardive. On définit un **segment** comme le sous-ordonnancement entre deux vallées (le premier segment va du début de l'ordonnancement jusqu'à la première vallée incluse, le second segment commence au sommet suivant cette première vallée jusqu'à la seconde vallée incluse, etc).

En reprenant l'arbre A et les ordonnancements partiels $\pi_1 = [f, g]$ et $\pi_2 = [a, b, c, d, e]$, on obtient les coût d'exécution et résiduels et les segments suivants :

i	1	2
$c(\pi_1, i)$	9	8
$cr(\pi_1, i)$	1	5

segments	$h_i^{\pi_1}$	$v_i^{\pi_1}$
$[f]$	$h_1^{\pi_1} = 9$	$v_1^{\pi_1} = 1$
$[g]$	$h_2^{\pi_1} = 8$	$v_2^{\pi_1} = 5$

i	1	2	3	4	5
$c(\pi_2, i)$	8	8	4	7	7
$cr(\pi_2, i)$	4	1	3	3	2

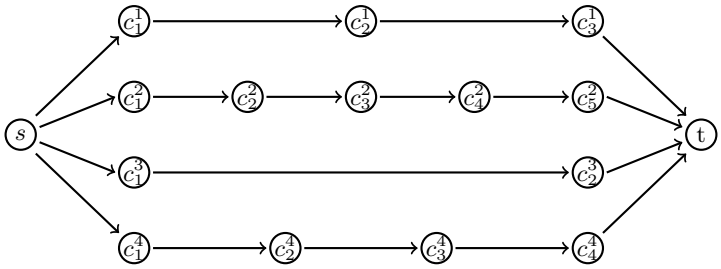
segments	$h_i^{\pi_2}$	$v_i^{\pi_2}$
$[a, b]$	$h_1^{\pi_2} = 8$	$v_1^{\pi_2} = 1$
$[c, d, e]$	$h_2^{\pi_2} = 7$	$v_2^{\pi_2} = 2$

5. Écrivez une fonction `segments` qui prend en entrée un arbre entrant et un ordonnancement et retourne la liste des collines et des vallées, ainsi que les segments correspondants.

L'algorithme pour calculer l'exécution optimale d'un arbre entrant est le suivant : on cherche un ordonnancement pour chaque enfant du sommet actuel et on calcule ensuite (ou pendant) les collines et les vallées de chacun de ces ordonnancements ainsi que les segments associés. Ensuite lorsqu'on doit choisir quel segment exécuter parmi ceux possible (un pour chaque enfant) on prend celui qui maximise $h_i - v_i$. Une fois tous les segments de tous les enfants effectués on exécute la racine. L'intuition derrière cet algorithme est que l'on préfère exécuter les plus gros pics mémoires avec un minimum de coûts résiduels venant des autres branches de l'arbre. L'optimalité de cet algorithme est admise.

6. Écrivez une fonction `ordonnancementArbre` qui prend en entrée un arbre entrant et retourne un ordonnancement minimisant le coût d'exécution. *Il est possible d'effectuer l'algorithme en $O(n^2)$ et pour le démontrer vous pouvez admettre que si $f(n) = n \log t + n + \sum_{i=i}^{t} f(n_i)$ avec $\sum_{i=1}^{t} n_i = n - 1$, alors $f(n) \leq n^2$.*

A partir de l'algorithme précédent on peut s'attaquer à une nouvelle sous-classe de graphe : celle des graphes "Fork-Join". Un graphe Fork-Join est un graphe possédant un sommet source s et un sommet cible t et ces deux sommets sont reliés par k chaînes.



Un exemple de graphe Fork-Join (non pondéré)

7. Proposez une structure de donnée pour représentez un graphe Fork-Join pondéré.

Il est possible d'obtenir un ordonnancement optimal pour un graphe Fork-Join en procédant de la manière suivante : on trouve pour chaque chaîne l'arête de poids minimal. Pour chaque autre arête de la chaîne on soustrait ce poids minimal au poids de l'arête considérée. Pour chaque sommet de la chaîne (hors s et t) on ajoute ce poids minimal au poids du sommet. On supprime ensuite l'arête. On remarque alors qu'on obtient alors deux arbres, un entrant (avec pour racine t) et un "sortant" (avec pour racine s). On calcule ensuite l'ordonnancement optimal pour chacun de ces arbres (en retournant les arêtes du second) et on admettra que la concaténation de ces deux ordonnancements (même s'il faut retourner celui de l'arbre enraciné en s) est un ordonnancement optimal.

8. Soit G un graphe Fork-Join pondéré à k branches et a^i_{min} l'arête de poids minimal de chaque branche. Soit S et T les arbres créés par l'algorithme. Soit π un ordonnancement de T et π' un ordonnancement de G où les sommets de S sont exécutés avant ceux de T et ceux de T sont exécutés dans le même ordre que dans π. Démontrez que

$$c(\pi, i) + \sum_{i=1}^{k} w_a(a^i_{min}) = c(\pi', i + |S|).$$

9. Écrivez une fonction `ordonnancementForkJoin` qui prend en entrée un graphe Fork-Join et retourne un ordonnancement minimisant le coût d'exécution.

Pour finir on va s'attaquer à une dernière classe de graphe dite "Série-Parallèle" (qui inclut au passage les graphes "Fork-Join"). Un graphe Série-Parallèle est défini récursivement de la façon suivante :

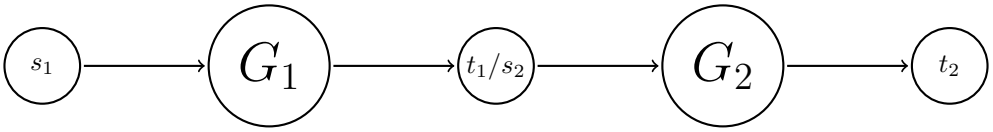
— Soit c'est un graphe composé uniquement d'une source s et d'une cible t, avec une arête allant de s vers t.

— Soit c'est la composition en série de deux graphes Série-Parallèle G_1 et G_2 (de sources et cibles respectives s_1, t_1, s_2, t_2), c'est à dire que sa source sera s_1, sa cible t_2 et on fusionne t_1 à s_2.

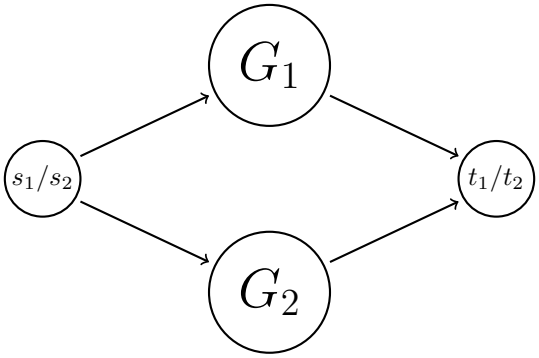
— Soit c'est la composition en parallèle de deux graphes Série-Parallèle G_1 et G_2 (de sources et cibles respectives s_1, t_1, s_2, t_2), c'est à dire qu'on fusionne s_1 et s_2 et t_1 et t_2.

Cas de base

Composition en série de G_1 et G_2



Composition en parallèle de G_1 et G_2



10. Proposez une structure de donnée pour représentez un graphe Série-parallèle pondéré.

On se propose maintenant de concevoir un algorithme optimal (et polynomial) pour les graphes Série-Parallèle. Pour cela on va se baser sur l'algorithme précédent.

12. Donner en, le justifiant, l'ordonnancement optimal du cas de base et de la composition en série, en supposant dans ce second cas que l'on connaît l'ordonnancement optimal des deux graphes que l'on compose.

Pour obtenir l'ordonnancement optimal lors de la composition en parallèle on va procéder de la manière suivante : on calcule l'ordonnancement optimal pour chacun des deux sous-graphes puis on les linéarise (on transforme chacun de ces deux graphes en chaînes respectant l'ordre d'exécution de leur ordonnancement optimal) et on calcule l'ordonnancement optimal du graphe Fork-join à deux chaînes ainsi obtenu. L'optimalité de l'ordonnancement obtenu ainsi est admis.

13. Écrire une fonction `lineariser` qui étant donné un graphe Série-Parallèle et un ordonnancement transforme le premier en une chaîne. Vous préciserez dans le rapport le poids donné aux sommets et aux arêtes de cette chaînes lors de cette transformation et vous justifierez qu'ils assurent la conservation des coûts d'exécution et des coûts résiduels de l'ordonnancement donné en entrée.

14. Écrivez une fonction `ordonnancementSerieParallele` qui prend en entrée un graphe Série-Parallèle et retourne un ordonnancement minimisant le coût d'exécution.