Ordonnancement et parallélisation de tâches sur un graphe orienté acyclique

Contexte

Un graphe orienté $G = (V, E)$ est dit acyclique s'il ne possède pas de circuit (cycle orienté), *i.e.*, s'il n'existe pas de chemin partant de v_i qui revienne sur v_i. Un tel graphe est alors nommé DAG, de l'anglais *Directed Acyclic Graph*.

Il est souvent utile de modéliser un problème par un ensemble de tâches v_i à réaliser. Parfois, une tâche v_i ne peut être commencée que lorsque une autre tâche v_j est terminée ; on dit alors que v_i dépend de v_j. On peut représenter un tel modèle par un graphe de dépendances (les arcs) entre tâches (les nœuds). Le problème admet une solution si et seulement si deux tâches ne sont pas interdépendantes, *i.e.*, si le graphe le représentant ne possède pas de cycle. Dans ce cas, le graphe est un DAG.

Cette définition extrêmement générale permet aux DAG de modéliser divers objets dans de nombreux domaines : les circuits logiques, en électronique ; les graphes PERT, en management ; les graphes de dépendance de compilateurs, tableurs, et autres réseaux de traitement de données ; en particulier, les réseaux de calcul (*Cloud Computing*, *Grid Computing*, architectures multicœurs, etc.).

Ce projet s'intéresse plus en détail à ce dernier cas ; spécifiquement, au problème d'ordonnancement et d'exécution (éventuellement parallèle) de tâches sur réseau de calcul. Chaque nœud v_i du DAG peut donc être vu comme une tâche, dont l'exécution a un certain coût c_i, et qui ne peut être exécutée qu'après les tâches la précédant (*i.e.*, les nœuds v_j tel qu'il existe un arc $e_{ji} : v_j \to v_i$; *cf.* Figure 1). Un nœud n'ayant aucun arc entrant est appelé une source ; un nœud n'ayant aucun arc sortant est appelé un puit.

En pratique le nombre de ressources disponibles pour traiter l'ensemble des tâches est limité. Dans ce contexte, le problème posé est l'exécution de toutes les tâches du graphe, dans un ordre respectant les dépendances imposées par le graphe, et en optimisant l'utilisation des ressources disponibles de manière à minimiser le temps total d'exécution.

1 Parcours séquentiel

On considère dans un premier temps le cas séquentiel : si une tâche v_i est en cours d'exécution, les autres sont en attente. Elles ne peuvent commencer leur exécution qu'après la fin de v_i. Par simplicité, on suppose qu'une tâche ne peut pas être interrompue en milieu d'exécution.

Le cas séquentiel survient par exemple lorsqu'on ne dispose que d'une seule ressource de calcul (ex : un seul processeur).

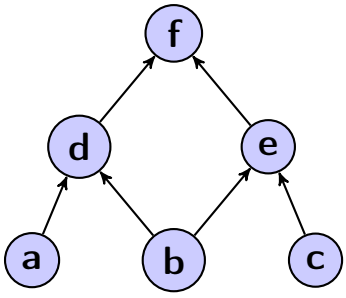

Figure 1 – La tâche **f** dépend des tâches **d** et **e**. La tâche **d** dépend des tâches **a** et **b** ; la tâche **e** dépend de **b** et **c**. Les tâches **a**, **b** et **c** sont sans dépendance : ce sont des sources. La tâche **f** est un puit.

Travail écrit :
Que peut-on immédiatement conclure sur le temps total d'exécution ? En déduire que le problème se ramène à uniquement respecter les dépendances du graphe.

1.1 Tri topologique

Un DAG, à travers l'expression de ses dépendances, définit automatiquement un ordre *partiel* sur les tâches : en effet, $(v_i, v_j) \in E \Rightarrow v_i \preceq v_j$.

Pour rappel, un ordre partiel sur V possède les mêmes propriétés qu'un ordre total (réflexivité, anti-symétrie et transitivité), mais en revanche il n'existe pas nécessairement de relation d'ordre entre tout couple d'éléments, *i.e.,* ($v_i \preceq v_j$ ou $v_j \preceq v_i$) n'est pas toujours vrai.

Par exemple, sur la Figure 1, les nœuds **a**, **b**, **c** ne sont pas ordonnés entre eux, tout comme **d**, **e**, ainsi que **a**, **e**, ou encore **c**, **d**.

Un algorithme de tri topologique consiste à trouver un ordre topologique sur V, *i.e.*, numéroter les nœuds de manière à établir entre eux un ordre *total* qui respecte l'ordre partiel du DAG. Un tel algorithme peut prendre la forme suivante :

```
1  Y = SansDep(V)
2  Z = [ ]
3  tant que Y ≠ [ ] faire
4  |   v_i = Retirer(Y).
5  |   Numéroter(v_i).
6  |   Ajouter(v_i, Z).
7  |   ∀v_j ∈ Succ(v_i) :
8  |   si Prec(v_j) ⊂ Z alors
9  |   |   Ajouter(v_j, Y).
10 |   fin
11 fin
```

A tout moment de l'algorithme, chaque nœud du graphe est dans un des trois possibles états :
— non numéroté et avec certains de ses prédécesseurs non numérotés ;

— non numéroté et avec tous ses prédécesseurs numérotés ;
— numéroté.

Travail écrit :
Identifiez ces états avec les listes Y, Z et $X = V \backslash (Y \cup Z)$. Quelle propriété garantit que l'ordre produit par cet algorithme est topologique, c'est-à-dire qu'il respecte l'ordre partiel défini par les dépendances du graphe ? Quelle propriété garantit qu'il est total ?

Travail écrit :
En supposant que les fonctions Succ et Prec ont un coût constant, calculer l'ordre de complexité de l'algorithme en fonction du nombre de nœuds n et du nombre d'arcs m.

L'ordre topologique produit par cet algorithme dépend du format de liste utilisé pour Y. Par exemple, sur la Figure 1, il existe 16 ordres topologiques possibles. En particulier, les ordres **a**,**b**,**c**,**d**,**e**,**f** et **a**,**b**,**d**,**c**,**e**,**f** sont respectivement obtenus en utilisant un format de file, et un format de pile, pour Y.

Travail écrit :
Comment appele-t-on les parcours induits par l'utilisation d'une pile ? Et d'une file ?

Travail pratique :
Programmer un algorithme de tri topologique utilisant un format de file pour Y.

2 Parcours parallèle

On considère désormais le cas parallèle : on suppose que l'on peut traiter en même temps r tâches indépendantes, où r est le nombre de ressources de calcul (ex : processeurs) dont on dispose.

A chaque étape, on va donc avoir au plus r tâches exécutées, que l'on stocke dans une liste. On représente la trace d'exécution par la liste de chacune des étapes, *i.e.*, une liste de listes.

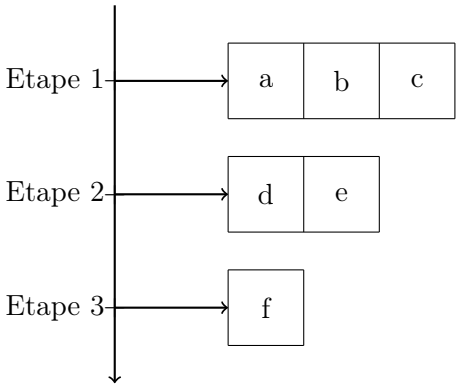

Figure 2 – Trace d'exécution des tâches du graphe de la Figure 1, en ayant un nombre de ressources au moins égal à 3, et en respectant l'ordre partiel du graphe. A chaque étape correspond une liste de tâches à exécuter, d'où la représentation en liste de listes.

2.1 Sans contrainte de ressources (ressources illimitées)

Dans un premier temps, on considère le cas $r = \infty$. Le nombre de tâches que l'on peut traiter en parallèle est donc uniquement limité par leurs dépendances, et non pas les ressources nécessaires à leur traitement.

On suppose également que chaque tâche a un coût uniforme $c_i = 1$, donc qu'elle est intégralement traitée par une ressource en une étape.

Travail écrit :
Quels nœuds sont traités à la première étape ? A la k-ième étape ? Conclure sur la valeur du temps d'exécution total (i.e., le nombre total d'étapes).

Travail pratique :
Programmer un algorithme produisant une trace d'exécution à partir d'un DAG, pour un nombre de ressources illimitées.

2.2 Avec contrainte de ressources limitées

On considère désormais un nombre de ressources de calcul fini $r \in \mathbb{N}$.

On continue de supposer que chaque tâche a un coût uniforme $c_i = 1$.

Travail écrit :
Donner une borne inférieure du temps total d'exécution en fonction du nombre de nœuds n et du nombre de ressources r. Dans le cas où cette borne inférieure est atteinte, que peut-on dire sur l'utilisation des ressources à chaque étape ?

Travail écrit :
Comment se traduit la contrainte de ressources limitées sur les listes à chaque étape ?

Travail pratique :
Adapter le programme codé en Section 2.1 pour qu'il respecte les contraintes de ressources limitées.

Lorsque le nombre de tâches prêtes à être traitées est supérieur au nombre de ressources disponibles, il faut choisir un ordre de priorité sur les tâches (i.e., choisir quelles tâches sont traitées immédiatement et quelles tâches sont reportées pour plus tard).

Trouver la stratégie optimale est un problème NP-complet. En général, les stratégies d'ordonnancement ont donc recours à une heuristique.

On appelle chemin critique le chemin le plus long reliant une source à un puit.

Travail écrit :
Proposer une heuristique pour choisir les tâches traitées en priorité (i.e., les r premières tâches de Y).

Travail pratique :
Implémenter votre heuristique. Vérifier si l'heuristique diminue le temps d'exécution (i.e., le nombre d'étapes).

Travail écrit :
Analyser l'efficacité de l'heuristique sur différents graphes, en faisant varier le nombre de ressources r.

2.3 Pondération des nœuds du graphe

On considère désormais un graphe aux nœuds pondérés, *i.e.*, on suppose que chaque tâche a un coût variable c_i. Pour traiter une tâche de coût c_i en p étapes, il faut assigner m_j ressources de calcul à l'étape j, de sorte que $c_i = \sum_{j=1}^{p} m_j$. Par simplicité, on considère des coûts entiers $c_i \in \mathbb{N}$.

Exemple : une tâche de coût $c = 3$ peut être traitée par 3 ressources en 1 étape, ou par 1 ressource en 3 étapes ; on peut également assigner 2 ressources à la première étape, et finir à la seconde étape avec 1 ressource.

Travail écrit :
Comment représenter un nœud pondéré par un graphe aux nœuds non pondérés ? En déduire une méthode simple pour résoudre le problème sur des graphes aux nœuds pondérés.

Travail pratique :
Coder une routine construisant le graphe aux nœuds non pondérés équivalent à un graphe aux nœuds pondéré.
Tester votre algorithme sur les graphes donnés.

Travail écrit :
Que remarquez-vous ? Expliquer les disparités au niveau du temps de calcul de votre programme entre les différents graphes tests fournis.

Travail pratique :
Adapter le programme codé en Section 2.2 au cas des graphes aux nœuds pondérés.

3 Evaluation du projet

Le projet sera réalisé en binôme.

Vous devez rendre un code commenté, auquel outre les tests de vos fonctions, vous ajouterez en commentaire [1] les réponses aux questions posées dans les cadres "Travail écrit". Vous rendrez une archive nommée "**Nom1_Nom2.tgz**".

Barême (approximatif) :

- Code + commentaires (14 pts)
  - Section 1 : 4 pts
  - Section 2.1 : 3 pts
  - Section 2.2 : 3 pts
  - Section 2.3 : 4 pts
- Tests (6 pts) : une séance de tests en salle de TP aura lieu le **25 Mars 2015**, de 16h à 18h. Les interfaces des fonctions demandées sont fournies sous Moodle (fichier "interface.mli").

Echéance : L'archive contenant votre code et les commentaires devront être envoyés à votre enseignant de TP avant le **23 Mars 2015**.

Pour aller plus loin :

Références

[1] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4) :406–471, December 1999.

1. Pas d'accent dans les commentaires !