Ordonnancement parallèle de tâches sur un graphe orienté acyclique

Contexte

Un graphe orienté $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ 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*. La documentation du module pour les graphes orientés est en ligne à l'URL `http://ocamlgraph.lri.fr/doc/Pack.Digraph.html`.

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.). La Section 1 de ce projet aborde plusieurs de ces exemples.

Le reste du projet (Sections 2-4) s'intéresse plus en détail au cas des réseaux de calcul ; 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 $e_{ji} = (v_j, v_i)$; cf. Figure 4). 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. Les ressources correspondent à des processeurs qui peuvent exécuter des tâches en parallèle. 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 Compréhension et modélisation

Application : programmes de jeu d'échecs

En 1997, le superordinateur Deep Blue bat le champion d'échecs Garry Kasparov. Cet ordinateur était conçu pour jouer aux échecs, avec des circuits spécifiques permettant d'améliorer sa puissance de calcul. Lors de la première partie contre Kasparov, Deep Blue joua un coup destabilisant pour Kasparov, qui

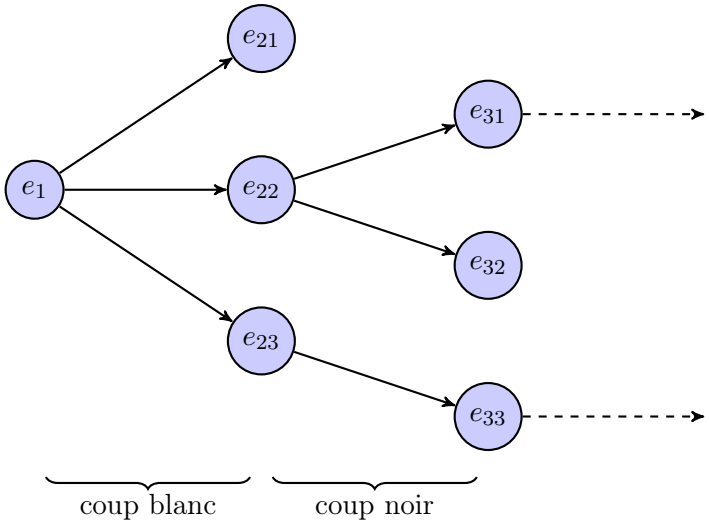

Figure 1 – Représentation sous forme de DAG des coups possibles à partir de l'état e_1. À chaque état du plateau de jeu, on associe un score, qui permettra de déterminer le coup à jouer.

n'arrivait pas à comprendre la stratégie de l'ordinateur... Coup qui était en réalité dû à un bug.

En effet, nous ne pouvons pas parler de stratégie au sens d'un joueur d'échecs, pour décrire le fonctionnement de Deep Blue. L'ordinateur se contentait en effet de prévoir un certain nombre de demi-coups (un coup complet étant constitué d'un demi-coup blanc et d'un demi-coup noir) en avance (*cf.* Figure 1). On attribue un score à l'état du pateau après chaque demi-coup possible, ce qui va permettre à l'ordinateur de choisir le meilleur demi-coup à jouer. Ce score peut être estimé en fonction de l'état du plateau suivant le demi-coup, ou bien en fonction des scores suivants.

Question 1:
Nous nous intéressons à la modélisation du problème des échecs par un DAG. Deep Blue prévoyait en moyenne 12 demi-coups à l'avance. On dispose d'une fonction S, qui à un état du plateau e attribue un score $S(e)$, via une heuristique. Plus la valeur de $S(e)$ est élevée, plus l'état du plateau est considéré comme avantageux par Deep Blue.

(a) Quelle valeur attribuer aux extrémités du DAG ? Connaissant la valeur attribuée aux successeurs d'un noeud, quelle valeur attribuer à ce noeud ? (attention à la distinction demi-coup noir et demi-coup blanc) Une fois le DAG rempli, comment décider du coup à jouer ? (un schéma sera grandement apprécié)

(b) Au jeu d'échecs, le nombre possible de demi-coups jouables est élevé, en moyenne 30 sont » plausibles « selon Claude Shannon (*cf.* Nombre de Shannon). Donner un ordre de grandeur du nombre de coup possibles à 12 demi-coups de profondeur.

(c) Face à cette monstruosité numérique, proposez une méthode de priorisation de l'exploration du DAG, pour éviter que Deep Blue ne se noie sous les calculs.

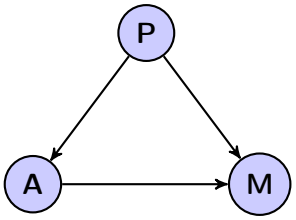

Figure 2 – Représentation sous forme de DAG d'un réseau bayésien

Application : Réseau Bayésien

En statistiques, il est possible de représenter des variables aléatoires grâce aux DAG par le biais de réseaux bayésiens. Cette représentation facilite la prise de décision en permettant d'intégrer des relations d'inférence (arcs) entre différentes variables aléatoires (noeuds). Dans cette modélisation, les arcs représentent en effet la relation causale (au sens probabiliste) entre deux phénomènes aléatoires.

Dans la Figure 2, nous avons représenté un exemple de réseau bayésien. On donne aussi à chaque noeud une table de probabilités conditionnelles dépendant des valeurs des variables aléatoires incidentes.

P représente la variable « il a plu » : $\mathbf{P}(P) = 0.5$.

A représente la variable « l'arrosoir a été mis en marche » :

Variables	**Valeurs**	**Probabilité**
P	Vrai	$\mathbf{P}(\text{A} \mid \text{P}) = 0.1$
P	Faux	$\mathbf{P}(\text{A} \mid \neg P) = 0.5$

M représente la variable « l'herbe est mouillée » :

Variables	**Valeurs**	**Probabilité**
(P,A)	(Vrai,Vrai)	$\mathbf{P}(\text{M} \mid \text{P} \wedge A) = 0.95$
(P,A)	(Vrai,Faux)	$\mathbf{P}(\text{M} \mid \text{P} \wedge \neg A) = 0.7$
(P,A)	(Faux,Vrai)	$\mathbf{P}(\text{M} \mid \neg P \wedge A) = 0.8$
(P,A)	(Faux,Faux)	$\mathbf{P}(\text{M} \mid \neg P \wedge \neg A) = 0.01$

Ce type de réseau bayésien est défini par sa loi de probabilité jointe :

$$\mathbf{P}(M \wedge A \wedge P) = \mathbf{P}(M|A \wedge P) \times \mathbf{P}(A|P) \times \mathbf{P}(P)$$

Question 2:

(a) En utilisant le graphe de la Figure 2 et les tableaux de probabilités, comment peut-on calculer $\mathbf{P}(M)$?

(b) On se propose de modéliser le problème de Monty Hall. Un jeu télévisé consiste à proposer à un candidat 3 boites de couleurs (Rouge, Verte et Bleue), l'une d'elle contient la clé d'une voiture, les deux autres ne contiennent rien. Dans un premier temps Le candidat est amené à choisir et désigné une des trois boîtes, sans l'ouvrir. Le présentateur, qui connaît l'emplacement de la clé, ouvre ensuite parmi les deux boites restantes, une qui ne contient pas de clé. Le présentateur propose alors dans un second temps au candidat de changer son choix, pour la boîte non révélée.

Pour établir le DAG correspondant au problème, identifier parmi les variables alétoires suivantes, celles qui sont indépendantes et les liens de causalités entres celles qui sont dépendantes :
— C_1 : Premier choix du candidat
— B_G : Boite gagnante contenant la clé
— C_2 : Second choix du candidat (si il décide de changer)
— B_R : Boite Révélée par le présentateur
Modéliser enfin le graphe.

(c) Afin de simplifier le problème, on supposera que le candidat choisit la boîte rouge et décidera de changer de choix. Calculer alors les tableaux de probabilités conditionnelles puis calculer la probabilité que le candidat gagne : $C_2 = B_G$. Avait-il raison de changer de boîte ?

Extension aux tâches à coût inconnu

Lorsque les coûts des tâches à exécuter sont connus à l'avance, il est possible de calculer le temps d'exécution d'un ordonnancement au préalable, avant le début de l'exécution des tâches (on parle alors d'ordonnacement *statique*). Cependant, dans la pratique, il n'est pas toujours facile (voire possible) de calculer ces coûts. Parfois, on ne dispose donc que d'estimées $\widetilde{c_i} \approx c_i$ des coûts des tâches. Dans ce cas-là, le temps d'exécution réel peut différer grandement du temps prévu. Une possible stratégie est de calculer un ordonnancement statique à partir des coûts estimés. Cependant, cet ordonnancement est dans la plupart des cas sous-optimal. Pour se rapprocher de l'ordonnancement optimal, il peut alors être corrigé dynamiquement lors de l'exécution des tâches, si l'on s'aperçoit que le coût réel d'une tâche est différent de son estimée.

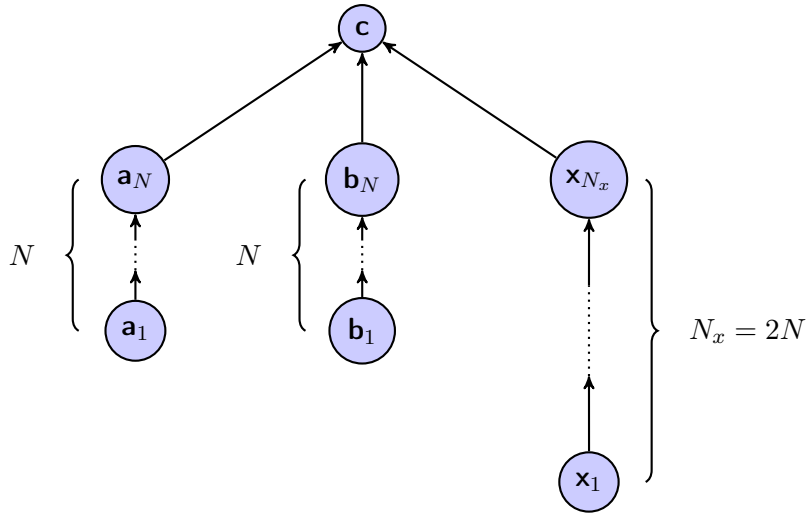

Figure 3 – Graphe des questions 3(b-d).

Question 3:

(a) Donnez deux exemples d'applications dont la modélisation sous forme de DAG mène à des tâches au coût inconnu ou incalculable précisément.

Pour les trois questions suivantes, on s'intéresse au graphe de la figure 3. On suppose que deux ressources sont disponibles (i.e., $r = 2$) et que chaque tâche a un coût égal à 1. On note $T_2(N)$ le temps total d'exécution des tâches du graphe avec 2 ressources.

(b) Si l'on suppose que le coût de la branche $\{x_i\}_{i=1:N_x}$, qui vaut $N_x = 2N$, est connu au préalable, alors l'ordonnancement optimal est facilement calculable. Calculez-le et donnez la valeur de $T_2(N)$ associée.

(c) On suppose maintenant que le côut N_x de la branche $\{x_i\}_{i=1:N_x}$ est inconnu, et que l'on dispose seulement d'estimées $\widetilde{N}_x = N$ du coûts réel qui reste inchangé et égal à $N_x = 2N$. Calculez l'ordonnancement (statique) obtenu sous l'hypothèse (fausse) $N_x = \widetilde{N}_x = N$. Cet ordonnancement prévoit un temps d'exécution égal à $\widetilde{T}_2(N) = 1.5N + 1$ qui s'avère erroné car N_x est égal à $2N$ et non N. Calculez la valeur de $T_2(N)$ réellement obtenue avec cet ordonnancement.

(d) Bonus : On cherche à corriger dynamiquement l'ordonnancement statique précédemment calculé à partir de ce coût estimé. Même si la valeur réelle de $N_x = 2N$ n'est découverte qu'à la fin de l'exécution des N_x tâches de la branche $\{x_i\}_{i=1:N_x}$, l'ordonnancement peut cependant être corrigé dès que l'on réalise que l'estimé $\widetilde{N}_x = N$ est incorrect. Calculez l'ordonnancement corrigé dynamiquement et la valeur de $T_2(N)$ associée.

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

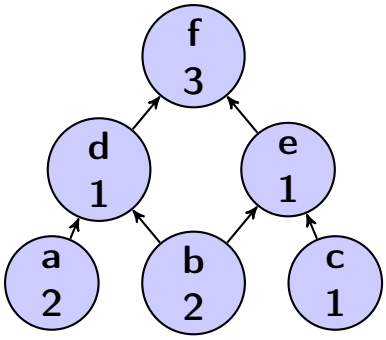

Figure 4 – La tâche **f** dépend des tâches **d** et **e**. La tâche **d** dépend des tâches **a** et **b** ; la tâche **e** dépend de **b** et **c**. Les tâches **a**, **b** et **c** sont sans dépendance : ce sont des sources. La tâche **f** est un puit. Les nombres en dessous des noms des nœuds correspondent à leur coût c_i.

Tri topologique

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

Pour rappel, un ordre partiel sur $\mathcal{V}$ 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 4, 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 $\mathcal{V}$, 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 = Sources(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.

Question 4:

(a) Identifiez ces états avec les listes Y, Z et $X = \mathcal{V} \backslash (Y \cup Z)$. 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 ?

(b) En supposant que les fonctions Succ et Prec ont un coût de c, calculer l'ordre de complexité de l'algorithme en fonction de c, 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 4, 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.

Question 5:
Comment appele-t-on les parcours induits par l'utilisation d'une pile ? Et d'une file ?

Question 6:
Programmez un algorithme de tri topologique utilisant un format de file pour Y.

3 Ordonnancement 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.

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 $c_i = \sum_{j=1}^{p} m_j$. 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.

Question 7:

(a) Donnez 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 ?

(b) Comment se traduit la contrainte de ressources limitées sur les listes à chaque étape ?

Question 8:
Programmez un algorithme produisant une trace d'exécution à partir d'un DAG, pour un nombre de ressources donné r.

Question 9:
Analysez l'efficacité de votre algorithme sur différents graphes, en faisant varier le nombre de ressources r.

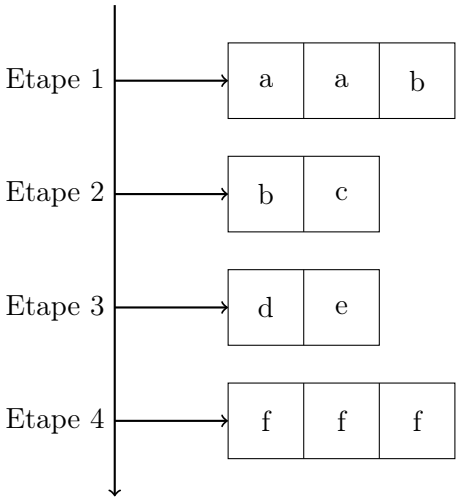

Figure 5 – Trace d'exécution des tâches du graphe de la Figure 4, en ayant un nombre de ressources égal à 3, et en respectant l'ordre partiel du graphe. A chaque étape correspond une liste de tâches à exécuter, d'où la représentation en liste de listes.

Bonus : amélioration de l'ordonnancement avec heuristique de priorité

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 un ordonnancement optimal 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.

Question 10:
Bonus : proposez une heuristique pour choisir les tâches traitées en priorité (i.e., les r premières tâches de Y).

4 Extension aux tâches et ressources hétérogènes

Dans cette section, on introduit la notion d'*hétérogénéité* des tâches et des ressources. Il arrive souvent de devoir exécuter des tâches de type différent en exploitant des ressources elles aussi de type différent. On peut prendre comme exemple un code de calcul devant réaliser à la fois des produits matrice-vecteur et des produits matrice-matrice, en exploitant à la fois des CPU et des GPU.

Dans les sections précédentes, il était supposé que le coût d'une tâche ne dépendait pas de son type ou du type de la ressource qui l'exécutait. Or, dans la pratique, il est courant que certains types de ressources soient plus adaptés à l'exécution de certains types de tâches. En gardant le même exemple, les GPU sont plus efficaces que les CPU pour le calcul de produits matrice-matrice mais moins pour celui de produits matrice-vecteur.

Par simplicité, on se place dans la suite dans le cas à 2 types de tâches et 2 types de ressources. On modélise la problématique de la façon suivante : on définit la *vitesse* d'une ressource comme le nombre de tâches qu'elle peut exécuter en une étape. Dans les parties précédentes, toutes les ressources avaient une vitesse de 1. Dans la suite, les ressources de type 1 ont une vitesse de 1 pour exécuter les tâches de type 1 mais une vitesse de $1/\alpha$ seulement pour exécuter les tâches de type 2. Il en va bien sûr inversement des ressources de type 2. $\alpha \in \mathbb{N}^*$ est le paramètre de d'hétérogénéité : plus il est grand, plus le problème est de nature hétérogène, c'est-à-dire plus grand est le surcoût si les tâches ne sont pas traitées par le "bon" type de ressource.

Question 11:
Si un ordonnancement calculé sans prendre en compte l'hétérogénéité d'un problème prévoit un temps d'exécution égal à t, quel peut être temps d'exécution réel dans le pire des cas ?

Question 12:
Généralisez votre programme pour pouvoir modéliser 2 types de tâches et 2 types de ressources. Votre programme prendra en argument r_1 et r_2, le nombre de ressources de type 1 et 2, ainsi que le paramètre α et rendra un ordonnancement valide, sans pour l'instant bénéficier de l'hétérogéniété (voir question 14).

Question 13:
Analysez l'efficacité de votre algorithme sur les DAGs hétérogènes fournis. Commentez sur l'influence du paramètre α.

Question 14:
Adaptez *de manière simple* votre algorithme pour que le calcul de l'ordonnancement bénéficie de l'hétérogénéité des 2 types de ressources et de tâches.

Question 15:
Analysez l'efficacité de votre algorithme adapté sur les DAGs hétérogènes fournis. Commentez sur l'influence du paramètre α.

Question 16:
Bonus : Voyez-vous d'autres stratégies plus complexes que celle que vous avez codé pour améliorer l'efficacité de votre algorithme ?

5 Evaluation du projet

Le projet sera réalisé en binôme.

Vous devez rendre un code commenté, auquel vous ajouterez les tests de vos fonctions. Vous fournirez les réponses aux questions dans un fichier au format pdf et vous rendrez une archive contenant votre code nommée "**Nom1_Nom2.tgz**".

Barême (approximatif) :

- Code + commentaires + réponses aux questions (16 pts)
  - Section 1 : 5pts
  - Section 2 : 3pts
  - Section 3 : 3pts
  - Section 4 : 5pts
- Tests (4 pts) : une séance de tests en salle de TP aura lieu à une date à définir. Les interfaces des fonctions demandées sont fournies sous Moodle (fichier "projet.mli").

Echéance : L'archive contenant votre code et vos réponses aux questions devront être déposées sur moodle avant le **21 Mars à minuit**.