Examen Algorithmique I (2020-21)

On insistera sur la correction et la clarté des réponses. Toute affirmation devra être justifiée explicitement, par un théorème du cours, par un numéro de question, par une référence à une règle. Pour tout raisonnement par récurrence, on devra dire sur quoi est effectuée la récurrence. N'inventez pas vos propres notations, et réutilisez les miennes, même si elles ne vous plaisent pas. Je veux une copie au propre, pas un brouillon : pas de rature, pas de questions dans le désordre, notamment. Je corrigerai très probablement toutes les copies en corrigeant d'abord toutes les questions 1 de tout le monde, puis toutes les questions 2, et ainsi de suite : ne comptez pas sur le fait que je me souvienne de ce que vous avez écrit dans une question précédente.

Je me réserve le droit de ne pas chercher à comprendre une réponse illisible, insuffisamment claire, ou plus longue ou compliquée que nécessaire.

1 Algorithme de Johnson

Soit $G \stackrel{\text{def}}{=} (S, A, c\colon A \to \mathbb{R})$ un graphe valué. On fabrique un nouveau graphe valué $G_* \stackrel{\text{def}}{=} (S_*, A_*, c_*)$ où :

- S_* est l'union disjointe de S et d'un nouveau sommet $*$;
- $A_* \stackrel{\text{def}}{=} A \cup \{(*, u) \mid u \in S\}$, autrement dit les arcs de G_* sont ceux de G, plus un arc de $*$ à chacun des sommets de G ;
- $c_*(e) \stackrel{\text{def}}{=} c(e)$ pour tout arc $e \in A$, et $c_*(*, u) \stackrel{\text{def}}{=} 0$ pour tout $u \in S$. (Note : j'écris $c_*(u, v)$ plutôt que l'inélégant $c_*((u, v))$.)

Question 1 Montrer que si G n'a pas de circuit de coût strictement négatif, alors G_* non plus.

> *Il n'y a aucun arc entrant sur $*$. Donc tous les circuits de G_* sont en fait des circuits de G.*

Dans la suite, on suppose que G, et donc G_*, n'a pas de circuit de coût strictement négatif. On note $h(u)$, pour tout $u \in S_*$, le coût minimum d'un chemin de $*$ à u dans G_*. On note que ceci est bien défini.

On définit un nouveau graphe $G_{**} \stackrel{\text{def}}{=} (S_*, A_*, c_{**})$, de mêmes sommets et mêmes arcs que G_*, mais avec $c_{**}(u, v) \stackrel{\text{def}}{=} c_*(u, v) + h(u) - h(v)$.

Question 2 Montrer que, sous l'hypothèse que nous avons faite que G n'a pas de circuit de coût strictement négatif, G_{**} n'a pas d'*arc* de coût strictement négatif.

*Comme G n'a pas de circuit de coût strictement négatif, G_* non plus par la **Question 1**, et donc la fonction h est bien définie... mais on l'a déjà dit dans l'énoncé.*

Soit $u \to v$ un arc quelconque de G_{**} (ou de G_*, c'est pareil). Il existe un chemin $* \to^* u$ dans G_* de coût $h(u)$. Le chemin $* \to^* u \to v$ obtenu en concaténant l'arc $u \to v$ a un coût $h(u) + c_*(u, v)$, qui est $\geq h(v)$, par minimalité de $h(v)$. Mais cette inégalité signifie que $c_{**}(u, v) \geq 0$.

Question 3 Soit d la fonction qui à tout couple (u, v) de sommets de G associe le coût minimum d'un chemin de u à v dans G. Soit D la fonction qui à tout couple (u, v) de sommets de G_{**} associe le coût minimum d'un chemin de u à v dans G_{**}. Montrer que, pour tous $u, v \in S$, $d(u, v) = D(u, v) + h(v) - h(u)$.

Tous les chemins de u à v dans G sont aussi des chemins dans G_{**}, et tous les chemins de u à v dans G_{**} sont aussi dans G, car u et v sont dans S, et qu'il n'y a pas d'arc entrant sur $*$ dans G_{**}. En particulier, s'il n'y a pas de chemin de u à v (dans G, et dans G_{**}), alors $D(u, v) + h(v) - h(u) = +\infty = d(u, v)$. (On note que $h(v)$ et $h(u)$ ne sont pas infinis, en fait ce sont des réels négatifs ou nuls.) On suppose dorénavant qu'il existe un chemin de u à v dans G, et donc dans G_{**}.

Si $u = u_0 \to u_1 \to \cdots \to u_k = v$ est un chemin π de u à v dans G, son coût (dans G) est $\sum_{i=1}^{k} c(u_{i-1}, u_i)$, et son coût dans G_{**} est $\sum_{i=1}^{k} c_{**}(u_{i-1}, u_i) = \sum_{i=1}^{k} c(u_{i-1}, u_i) + h(u) - h(v)$ (en annulant les termes $h(u_i)$ deux à deux).

Donc si π est de coût minimal $d(u, v)$ dans G, il est de coût $d(u, v) + h(u) - h(v)$ dans G_{**}. Par minimalité de D, $D(u, v) \leq d(u, v) + h(u) - h(v)$, donc $d(u, v) \geq D(u, v) + h(v) - h(u)$.

Réciproquement, si π est de coût minimal $D(u, v)$ dans G_{**}, notons que comme u et v sont dans S et qu'il n'y a pas d'arc entrant sur $*$ dans G_{**}, tous les sommets u_i sont en fait dans S. Alors π est de coût $D(u, v) + h(v) - h(u)$ dans G, ce qui montre l'inégalité inverse $d(u, v) \leq D(u, v) + h(v) - h(u)$.

Question 4 En déduire un algorithme de calcul de la table $(d(u, v))_{u,v \in S}$ de tous les coûts minimums de u à v dans G. On rappelle que G est supposé sans circuit de coût strictement négatif. Nommez explicitement les algorithmes du cours que vous utilisez, mais ne les recopiez pas. Vous devez obtenir un algorithme d'une complexité meilleure que la complexité $O(n^3)$ de Roy-Warshall/Floyd, et strictement meilleure lorsque G est clairsemé (c'est-à-dire si $m = O(n)$). Donnez cette complexité explicitement.

On représente G par liste de successeurs. On peut alors fabriquer G_* en $O(n)$.

On calcule ensuite h par Bellman-Ford ou par Ford en $\Theta(n(m + n))$.

On met ensuite à jour les coûts des arcs $u \to v$ de G_* en leur ajoutant $h(u) - h(v)$, en temps $O(n + m)$.

Pour chaque sommet $u \in S$, on calcule ensuite tous les coûts minimums $D(u, v)$, $v \in S_*$, en temps $\Theta(m + n \log n)$ car il n'y a pas d'arc de coût strictement négatif dans G_{**} : c'est l'algorithme de Dijkstra, que l'on appelle autant de fois qu'il y a de sommets $u \in S$, pour une complexité totale de $\Theta(nm + n^2 \log n)$.

Finalement, on calcule les $d(u, v) = D(u, v) + h(v) - h(u)$ par une dernière passe en temps $\Theta(n^2)$.

La complexité totale est de $\Theta(nm + n^2 \log n)$. Lorsque $m = \Theta(n^2)$, c'est de l'ordre de n^3, ce qui n'est pas mieux que Roy-Warshall/Floyd. Mais si $m = O(n)$, c'est un

$O(n^2 \log n)$, ce qui est strictement meilleur.

Question 5 On ne suppose plus que G soit sans circuit de coût strictement négatif. Peut-on modifier l'algorithme précédant, sans augmenter sa complexité asymptotique, de sorte qu'il détecte en plus si G a un circuit de coût strictement négatif ? Justifier.

Oui. Dans la phase d'initialisation, si l'on utilise Bellman-Ford (pas Ford), on a vu en cours qu'on pouvait détecter l'existence de circuits de coût négatif (dans G_*) en seulement une itération supplémentaire, ce qui ne change pas la complexité asymptotique.
*Si G_* a un circuit de coût négatif, la **Question 1** nous dit que G en a un aussi.*
Sinon, l'algorithme des questions précédentes fonctionne sans modification.

2 Graphes et-ou

Un *graphe et-ou* est un graphe orienté $G \stackrel{\text{def}}{=} (S, A)$, où l'ensemble des sommets S est partitionné en deux parties disjointes $S_\wedge$ et $S_\vee$.

L'ensemble des *sommets accessibles* de G est par définition le plus petit ensemble de sommets $R_0 \subseteq S$ tel que :

- pour tout sommet $v \in S_\vee$, s'il existe un sommet $u \in R_0$ et un arc $u \to v$, alors $v \in R_0$;
- pour tout sommet $v \in S_\wedge$, si tout arc $u \to v$ d'extrémité v a sa source u dans R_0, alors $v \in R_0$.

Question 6 On se donne le graphe et-ou décrit par :

- $S_\wedge \stackrel{\text{def}}{=} \{a, b, c\}$;
- $S_\vee \stackrel{\text{def}}{=} \{u_1, u_2, u_3, u_4, u_5\}$;
- $A \stackrel{\text{def}}{=} \{(a, u_1), (u_1, u_2), (u_1, b), (u_3, b), (b, u_4), (u_2, c), (c, u_3), (c, u_5), (u_5, c)\}$.

Quel est l'ensemble de ses sommets accessibles ? Indication : ce n'est pas l'ensemble vide ! Il est conseillé de bien comprendre comment ceci fonctionne avant de passer à la suite.

On note que a n'a pas de prédécesseur, et est dans $S_\wedge$, donc il doit être dans R_0. Ceci force u_1, puis u_2, à être dans R_0. Et c'est tout. Donc $R_0 = \{a, u_1, u_2\}$.

On supposera dans la suite que `succ` (u) donne la liste des successeurs d'un sommet u. Chaque sommet v dispose aussi d'un champ (modifiable) `count`(v), qui est un entier naturel. L'idée est que `count`(v) est le nombre de prédécesseurs de v qu'il reste à marquer avant de savoir que tous les prédécesseurs de v sont marqués. Ceci n'aura de sens que lorsque $v \in S_\wedge$.

Plus précisément, on souhaite calculer l'ensemble R_0 des sommets accessibles d'un graphe et-ou donné en entrée. Pour ceci, on imite la procédure `reach_set` vue en cours, et on va accumuler dans un ensemble `marked`, initialement vide, les sommets de R_0. Les sommets de `marked` sont les sommets *marqués*. On maintient aussi une pile `work` de sommets *en attente*. On maintiendra comme invariants :

Inv1 pour tout $v \in S_\wedge$, il y a exactement `count`(v) prédécesseurs de v qui ne sont pas dans `marked` ;

Inv2 pour tout $v \in S_\wedge$, si `count`$(v) = 0$ alors $v \in$ `marked` $\cup$ `work` ;

Inv3 pour tout $v \in S_\vee$, s'il existe un prédécesseur de v dans `marked`, alors $v \in$ `marked`$\cup$`work` ;

Inv4 tous les sommets de `marked` $\cup$ `work` sont dans R_0.

Question 7 Écrire les lignes de code d'initialisation : elles doivent initialiser `marked` à $\emptyset$ et `work` à la pile vide, puis établir les invariants ci-dessus. Vous pouvez utiliser des instructions de la forme `for each` $v \in S$ pour parcourir les sommets, et de même avec $S_\wedge$ et $S_\vee$ et avec la liste `succ` (u) des successeurs d'un sommet u. Vous pouvez aussi tester par $v \in S_\wedge$ si un sommet v est dans $S_\wedge$, et similairement pour $S_\vee$. Je ne demande pas une preuve formelle en logique de Hoare, mais vous devez expliquer votre code, clairement et succinctement, et notamment quelles lignes établissent quels invariants.

```
01 marked := ∅;
02 work := empty();
03 for each u ∈ S:
04     count(u) := 0;
05 for each u ∈ S do:
06     for each v ∈ succ(u):
07         if v ∈ S_∧
08            then count(v)++;
09 for each v ∈ S_∧:
10     if count(v)=0
11        then push(v,work);
```

Les lignes 03–08 parcourent tous les arcs de G, et comptent pour chaque v fixé dans $S_\wedge$ combien il y en a d'extrémité v dans `count`(v). *Comme* `marked` *est vide, ceci établit* **Inv1**.

Les lignes 09–11 établissent **Inv2** *: les sommets de* `work` *sont exactement les sommets $v \in S_\wedge$ tels que* `count`$(v) = 0$.

Dit autrement, et grâce à **Inv1**, *les sommets de* `work` *sont exactement les sommets de $S_\wedge$ qui n'ont aucun prédécesseur. Ils sont donc tous dans R_0. (C'était le but de la* ***Question 6*** *de vous faire comprendre ce point.) Donc* **Inv4** *est vrai.*

Inv3 *est trivialement vrai, parce que* `marked` *est vide.*

Question 8 Écrire une procédure `mark`(u) qui marque un sommet u passé en argument, sous les hypothèses, valables en entrée :

(a) $u \notin$ `marked` ;

(b) pour tout $v \in S_\wedge$, il y a exactement `count`(v) prédécesseurs de v qui ne sont pas dans `marked` ;

(c) pour tout $v \in S_\wedge$, si `count`$(v) = 0$ alors $v \in$ `marked` $\cup$ `work` $\cup \{u\}$;

(d) pour tout $v \in S_\vee$, s'il existe un prédécesseur de v dans `marked`, alors $v \in$ `marked` $\cup$ `work` $\cup \{u\}$;

(e) tous les sommets de `marked` $\cup$ `work` $\cup \{u\}$ sont dans R_0.

En sortie, `marked` doit être égal à l'ensemble `marked` donné en entrée, union $\{u\}$, et les invariants **Inv1**–**Inv4** doivent être vérifiés. Justifier, en citant méticuleusement les invariants utilisées, et les propriétés établies.

```
01 fun mark(u):
02     marked := marked ∪ {u};
03     for each v ∈ succ(u):
04         if v∈ S∧
05            then (count(v)--;
06                  if count(v)=0 then push(v,work))
07         else push(v,work);
```

Pour tout $v \in S_\wedge$, le fait de marquer `u` *laisse le nombre de prédécesseurs non marqués de v inchangé si v n'est pas dans* `succ(u)`*, et le fait décroître de* 1 *sinon. La ligne 05 assure donc que* **Inv1** *est vrai, grâce à (b).*

La ligne 06 assure que **Inv2** *est vrai. On doit empiler* `v` *sur* `work` *pour cela, car il est demandé de ne marquer que* `u`*. On pourrait en principe éviter la ligne 06 dans le cas où* `u=v`*, mais ceci impliquerait un test supplémentaire. Un peu plus formellement, les seuls sommets $v \in S_\wedge$ tels que* `count`$(v) = 0$ *en sortie de* `mark` *sont ceux qui étaient dans* `marked` *ou* `work` *en entrée, ou égaux à* `u`*, par (c), plus ceux qui n'avaient que* `u` *comme prédécesseur non marqué en entrée. Ces derniers sont dans* `work` *en sortie. Quant à* `u`*, il se retrouve dans* `marked` *en sortie.*

L'invariant **Inv3** *est assuré par la ligne 07 : les sommets $v \in S_\vee$ qui ont un prédécesseur dans* `marked` *en sortie sont ceux qui ont un prédécesseur dans l'ensemble* `marked` *en entrée ou qui ont* `u` *comme prédécesseur. Ces derniers sont dans* `work` *en sortie par la ligne 07. Les précédents sont dans l'ensemble* `marked` *en entrée, ou dans l'ensemble* `work` *en entrée, ou égaux à* `u` *par (d). Ils sont donc dans l'ensemble* `marked` *de sortie, ou dans l'ensemble* `work` *d'entrée, qui est inclus dans le* `work` *de sortie.*

Finalement, les sommets de `work` *à la fin de la procédure sont ceux qui y étaient avant, plus :*

- *les successeurs v de* `u` *qui sont dans $S_\vee$: par (e),* `u` *est dans R_0, donc v aussi ;*
- *les successeurs v de* `u` *dans $S_\wedge$ tels que* `count`(v) *est passé à* 0*; par* **Inv1**, *les prédécesseurs de v sont tous dans* `marked` *en sortie, donc dans le* `marked` *d'entrée ou égaux à* `u`*, et ils sont donc tous dans R_0 par (e). Donc v est dans R_0 aussi.*

Les sommets de `marked` *en sortie sont ceux qui étaient dans* `marked` *en entrée, plus* `u`*, et ils sont dans R_0 par (e). Tout ceci établit* **Inv4**.

Question 9 On considère l'algorithme :

```
⟨ code d'initialisation de la Question 7 ⟩
while not(emptyp(work)) do
      let u = pop(work) in
          if u ∉ marked then mark(u); (* Question 8 *)
```

(Ici, `emptyp` teste la vacuité de son argument, et `pop(work)` dépile un élément de `work`.) En raisonnant sur les invariants, que l'on devra citer explicitement, justifier que, à la sortie de l'algorithme, `marked` $= R_0$.

*Les invariants sont établis après le code d'initialisation, puis maintenus entre l'entrée et la sortir du corps de boucle, par la **Question 8**. En sortie de boucle,* `work` *est vide. Alors :*

Pour tout $v \in S_\wedge$, si tous les prédécesseurs de v sont marqués, alors $\texttt{count}(v) = 0$ *par* **Inv1**, *donc v est marqué par* **Inv2** *et le fait que* `work` *est vide.*

Pour tout $v \in S_\vee$, s'il existe un prédécesseur de v qui est marqué, alors v est marqué lui aussi, par **Inv3** *et le fait que* `work` *est vide.*

Par la propriété de minimalité de R_0, $R_0 \subseteq$ `marked`. **Inv4** *implique que cette inclusion est une égalité.*

Question 10 Comme d'habitude, on note n le nombre de sommets et m le nombre d'arcs de G. Pourquoi l'algorithme de la **Question 9** ne peut-il faire qu'au plus $n + m$ tours de sa boucle `while` ? Je ne demande pas une preuve à coups d'invariants, mais une justification convaincante. Bonus : en raffinant (éventuellement) un peu vos algorithmes, montrer qu'on peut descendre à n tours de boucle `while` (ce que l'on admettra dans la suite).

D'abord, le nombre de tours de boucle est égal au nombre d'appels à `pop`, *et donc au nombre d'appels à* `push`, *qui lui est égal.*

Ensuite, `mark` *n'est jamais appelée sur chaque sommet u qu'au plus une fois, car elle n'est appelée que si u est non marqué, et le marque immédiatement. On appelle ensuite* `push` *sur un sous-ensemble des successeurs de u, donc au plus* $|\texttt{succ}(u)|$ *fois.*

Le seul autre endroit où l'on appelle `push` *est dans l'initialisation, où* `push` *est appelée au plus n fois.*

Donc le nombre total d'appels à `push`, *égal au nombre de tours de boucle, est d'au plus* $n + \sum_{u \in S} |\texttt{succ}(u)| = n + m$.

Bonus. On peut faire mieux en remplaçant `push` *par* `push_opt`, *qui n'empile v sur* `work` *que s'il n'y est pas déjà. Pour ceci, on maintient un tableau de bits disant si chaque sommet est déjà dans* `work`. *Chaque sommet n'est alors empilé, et donc dépilé, qu'au plus une fois, et le nombre de tours de boucle est majoré par n.*

Question 11 On suppose que G nous est donné dans la représentation suivante : d'abord, un graphe orienté usuel décrit par liste de successeurs, et pour chaque sommet u, un champ $\texttt{et\_ou}(u)$, qui vaut 1 si $u \in S_\wedge$ et 0 sinon. Montrer que l'algorithme de la **Question 9**, avec l'amélioration suggérée à la **Question 10**, fonctionne en temps $O(n + m)$. On précisera notamment la complexité des opérations de la **Question 7** et de la **Question 8**, les façons d'implémenter les diverses opérations non données dans la représentation de G (`for each` $v \in S_\wedge$, $v \in S_\wedge$, la gestion de l'ensemble `marked`, etc.).

Comme dans le cours, on gère l'ensemble `marked` *avec un champ booléen* $\texttt{markp}(u)$ *pour chaque sommet u. Ceci assure que le marquage, et le test $u \in$*`marked` *se réalisent en temps constant.*

*Pour la **Question 7**, le test $v \in S_\wedge$ de la ligne 07 se code en testant si* $\texttt{et\_ou}(v)$ *est vrai, en temps constant. L'énumération de la ligne 09 peut se faire en énumérant tous les sommets v de S, et en ne retenant que ceux tels que $v \in S_\wedge$. On peut aussi préconstruire une liste des sommets de $S_\wedge$, en temps $\Theta(n)$.*

En utilisant une représentation de `work` *par liste par exemple, la **Question 7** prend donc un temps $\Theta(n)$ (lignes 03-04) $+\Theta(n+m)$ (lignes 05-08) $+\Theta(n)$ (lignes 09-11), donc $\Theta(n + m)$ au total.*

La procédure `mark` *prend un temps* $\Theta(|\texttt{succ(u)}|)$*, car toutes les opérations élémentaires sont en temps constant.*

Par la ***Question 10****, on ne fait qu'au plus* n *tours de boucle* `while`*, et chaque sommet* u *est marqué au plus une fois. Le temps est donc majoré par un* $O(n)$ *plus* $O(\sum_{u\in S}(|\texttt{succ(u)}|)) = O(m)$*, soit* $O(m+n)$*. (Sans l'amélioration suggérée à la* ***Question 10****, le nombre de tours de boucle serait majoré par* m*, et le fait que chaque sommet ne peut être marqué qu'au plus une fois donne une complexité* $O(m)+O(n)+O(\sum_{u\in S}(|\texttt{succ(u)}|)) = O(n+m)$ *aussi.) On ajoute le temps* $O(m+n)$ *de l'initialisation, d'où un temps total de* $O(m+n)$*.*

Question 12 En guise d'application, on considère le problème de vacuité du langage d'une grammaire algébrique. Soit Σ un alphabet fini fixé. Une grammaire algébrique $\mathcal{G}$ est un triplet (NT, S, P) où :

- NT est un ensemble fini dit de *non-terminaux*, disjoint de Σ (les éléments de Σ sont parfois appelés les *terminaux*) ;
- S est un élément de NT, appelé *non-terminal de départ* ;
- P est un ensemble fini de *productions* ; une production est un couple formé d'un non-terminal X et d'un mot fini W sur $(\Sigma \cup NT)^*$, et on l'écrira $X \to W$. En général, W s'écrit de façon unique comme la concaténation $w_0Y_1w_1Y_2\cdots w_{k-1}Y_kw_k$ où les w_i sont des mots de Σ^* et les Y_i sont des non-terminaux, notation que l'on reprendra implicitement plus bas.

La *taille* d'une règle $X \to w$ est égale à la longueur $|w|$ de w plus 1, et la taille d'une grammaire algébrique est la somme des tailles de ses règles.

On définit le système de preuve suivant, permettant de dériver des jugements $\vdash w' \in Y$, où $w' \in \Sigma^*$ et $Y \in NT$, dont la signification intuitive est « le mot w' est produit par la grammaire au non-terminal Y ». Pour chaque production $X \to w_0Y_1w_1Y_2\cdots w_{k-1}Y_kw_k$ dans P, on a une règle de la forme :

$$\frac{\vdash w'_1 \in Y_1 \quad \cdots \quad \vdash w'_k \in Y_k}{\vdash w_0w'_1w_1w'_2\cdots w_{k-1}w'_kw_k \in X}$$

Le *langage* $L(\mathcal{G})$ de la grammaire algébrique $\mathcal{G}$ est par définition l'ensemble des mots $w \in \Sigma^*$ tels qu'on peut dériver $\vdash w \in S$ dans ce système.

Montrer que, étant donnée une grammaire algébrique $\mathcal{G}$ en entrée, on peut décider si $L(\mathcal{G})$ est vide ou non en temps linéaire en la taille de $\mathcal{G}$.

On fabrique un graphe et-ou G *dont les sommets sont :*

- *les non-terminaux, qui sont des sommets « ou » :* $S_\vee \stackrel{def}{=} NT$ *;*
- *les sommets « et » sont les productions :* $S_\wedge \stackrel{def}{=} P$*.*

Les arcs sont, pour chaque production $u \stackrel{def}{=} (X \to w_0Y_1w_1Y_2\cdots w_{k-1}Y_kw_k)$ *:*

- k *arcs des* Y_i *vers* u*,* $1 \le i \le k$ *;*
- *un arc de* u *vers* X*.*

Par récurrence sur la dérivation d'un jugement $\vdash w \in Y$*, on montre de façon immédiate que* Y *est accessible dans* G*.*

Réciproquement, on montre que si Y est accessible dans G, alors il existe un mot $w \in \Sigma^*$ tel que $\vdash w \in Y$ soit dérivable. Une démonstration formelle est la suivante. Soit $A_\vee$ l'ensemble des non-terminaux Y tels qu'on peut dériver $\vdash w \in Y$ pour au moins un mot $w \in \Sigma^*$, et soit $A_\wedge$ l'ensemble des productions $u \stackrel{def}{=} X \to w_0 Y_1 w_1 Y_2 \cdots w_{k-1} Y_k w_k$ telles que pour tout i, $1 \leq i \leq k$, il existe des mots $w_i \in \Sigma^*$ tels qu'on peut dériver $\vdash w_i \in Y_i$. Soit $A \stackrel{def}{=} A_\vee \cup A_\wedge$. Alors :

- *pour tout sommet v de $S_\vee$, s'il existe un sommet $u \in A$ et un arc $u \to v$, c'est que v est un non-terminal X, que u est une production $X \to w_0 Y_1 w_1 Y_2 \cdots w_{k-1} Y_k w_k$, et qu'il existe des mots w'_i tels que l'on puisse dériver $\vdash w'_i \in Y_i$, pour tout i, $1 \leq i \leq k$; on peut alors dériver $\vdash w_0 w'_1 w_1 w'_2 \cdots w_{k-1} w'_k w_k \in X$, donc X est dans A ;*
- *pour tout sommet v de $S_\wedge$, si tout arc $u \to v$ d'extrémité v a sa source u dans A, alors v est dans A aussi. En effet, v est une production $X \to w_0 Y_1 w_1 Y_2 \cdots w_{k-1} Y_k w_k$, les arcs d'extrémité v sont tous les arcs de sources Y_1, ..., Y_k, et comme ces derniers sont tous dans A, on peut peut trouver des mots w'_i tels que l'on puisse dériver $\vdash w'_i \in Y_i$, pour tout i, $1 \leq i \leq k$; donc v est dans A.*

Comme l'ensemble des sommets accessibles dans G est le plus petit qui satisfait ces deux conditions, il est inclus dans A. En particulier, tout sommet de $S_\vee$, c'est-à-dire tout non-terminal Y accessible dans G est tel qu'il existe un mot $w \in \Sigma^*$ tel que $\vdash w \in Y$ soit dérivable.

Ceci implique que $L(\mathcal{G})$ est non vide si et seulement si S est accessible dans G. Or les questions précédentes montrent qu'on peut le décider en temps $O(m+n)$, et $m+n$ est inférieur ou égal à la taille de $\mathcal{G}$. (On doit aussi ajouter au temps de calcul celui de la conversion de la grammaire en graphe, qui est clairement linéaire.)

3 Parcours en profondeur de graphes non orientés

Un graphe non orienté $G \stackrel{\text{def}}{=} (S, A)$ est vu, comme en cours, comme un graphe orienté dont les arcs sont les couples (u, v) (notés $u \to v$) tels que $u - v$ soit une arête de G. (On note $u - v$ une arête, c'est-à-dire la paire $\{u, v\} \in A$.) Comme $u - v$ et $v - u$ sont la même arête, ceci signifie que si $u \to v$ est un arc de G, alors $v \to u$ aussi.

Dans la suite, on suppose G connexe, en tant que graphe non orienté, et on se fixe un sommet $*$ quelconque de G.

Question 13 Pourquoi $*$ est-il une source de G vu comme graphe orienté ? Autrement dit, pourquoi tout sommet de G est-il accessible depuis $*$?

Parce que G est connexe. Il existe donc un chemin non orienté entre $*$ et tout sommet u, et ceci est un chemin orienté aussi.

Question 14 On se fixe un parcours en profondeur L partant de $*$ quelconque de G, vu comme graphe orienté. Montrer que L ne contient aucun arc transverse.

On utilise la classification des arcs usuelle. Si $u \to v$ est un arc transverse, alors $r(u) > r(v)$ et $fin(u) > fin(v)$. L'arc $v \to u$ est donc un arc avant (ou un arc de liaison). Mais ceci signifie qu'il existe un chemin $v \to^*_T u$ dans l'arbre couvrant T

défini par L. En conséquence, $u \to v$ est un arc arrière. Mais un arc arrière satisfait [$r(u) > r(v)$ et] $fin(u) \leq fin(v)$, contradiction.

4 Ordonnancement biprocesseur

On souhaite résoudre le problème de l'allocation statique de tâches et de bases de données sur un système à deux processeurs communiquant entre eux.

En entrée, on nous donne une liste de tâches à effectuer et une liste de bases de données. On appellera les tâches et les bases de données des *modules*. Formellement, on nous donne $t+d$ modules $M_1, \ldots, M_{t+d}$, les t premiers étant appelés des tâches, les d derniers des bases de données. On nous donne aussi les temps T_{i1} et T_{i_2} que mettrait chaque tâche M_i, $1 \leq i \leq t$ à être exécutée sur chacun des deux processeurs, ainsi que les temps de communication C_{ij} entre les modules M_i et M_j lorsqu'ils sont alloués sur des processeurs différents (communication entre tâches, accès d'une tâche à une base de donnée, notamment). Lorsque M_i et M_j sont alloués sur le même processeur, on suppose que leur temps de communication est nul.

Une *allocation* des modules est un sous-ensemble E de $\{1, \cdots, t+d\}$, qui représente les numéros des modules placés sur le processeur numéro 1. Les autres sont placés sur le processeur numéro 2. Le *coût* d'une telle allocation est la somme :

$$c(E) \stackrel{\text{def}}{=} \sum_{i \in E \cap \{1,\cdots,t\}} T_{i1} + \sum_{i \in \{1,\cdots,t\} \setminus E} T_{i2} + \sum_{i \in E, j \in \{1,\cdots,t+d\} \setminus E} C_{ij}.$$

On souhaite trouver une allocation E de coût $c(E)$ minimal. Pour ceci, on forme un réseau dont les sommets sont numérotés $1, \ldots, t+d$, en plus d'une source s et d'un puits p.

Question 15 Compléter la description du réseau, et montrer que l'on peut résoudre algorithmiquement la question du coût $c(E)$ minimal en temps $O((t+d)^4)$ (ou mieux !).

On souhaite exprimer $c(E)$ comme la valeur d'une coupe, qui sera $E \cup \{s\}$. On demande donc :

- *un arc $i \to p$ de capacité T_{i1} pour tout i avec $1 \leq i \leq t$ (première somme dans la définition de $c(E)$) ;*
- *un arc $s \to i$ de capacité T_{i2} pour tout i avec $1 \leq i \leq t$ (deuxième somme) ;*
- *un arc $i \to j$ de capacité C_{ij} pour tous i et j avec $1 \leq i, j \leq t+d$ (troisième somme).*

La question est donc de trouver la valeur de la coupe minimum du réseau. De façon équivalente, de trouver la valeur du flot maximum. Ceci peut se faire avec n'importe lequel des algorithmes de flots maximum du cours. On a $n \stackrel{def}{=} t+d+2$ sommets et $m \stackrel{def}{=} 2t + (t+d)^2$ arcs. On a donc une complexité :

- *$O(n^2 m) = O((t+d)^4)$ par Edmonds-Karp-Dinic ;*
- *$O(m^2 \log C)$, où C est le max des T_{i1}, T_{i2}, et C_{ij}, par l'algorithme par échelonnement : c'est un $O((t+d)^4 \log C)$, ce qui est moins bon ;*
- *$O(n^3) = O((t+d)^3)$ par Karzanov (préflots), ce qui est bien meilleur.*