École polytechnique
Promotion 2007
INF431 : Programmes et Algorithmes, du séquentiel au distribué.

Contrôle de classement CC2 du mercredi 1er juillet 2009
par Philippe Jacquet et Jean-Marc Steyaert
CORRIGÉ

Les problèmes suivants sont indépendants. Il est demandé de les **rédiger sur des copies séparées** *pour permettre des corrections indépendantes.*
On accordera la plus grande importance à la clarté des explications et des algorithmes donnés en réponse. Certaines questions sont partiellement traitées dans les notes de cours : il importera de reprendre une explication globale cohérente. Sont exclusivement autorisés les polycopiés, notes de cours et de PC.

1 Problème 1 : Election de *leader* avec un canal à collision

1.1 Eléments de base et primitives

Nous considérons un réseau de n postes ($n \geq 1$) connectés à un dispositif de communication à accès multiple (par exemple une borne wifi). Nous proposons un modèle simplifié de la borne. Le temps est divisé en tranches appelées *slot*. Le slot est divisé en deux périodes : la période *montante* pendant laquelle les postes sont autorisés à émettre, et la période *descendante* pendant laquelle seule la borne est autorisée à émetttre (voir figure 1).

Slot de transmission

FIGURE 1 – Division du slot

Chaque poste est autorisé à transmettre au plus un paquet par slot, pendant la période montante. Si un poste est seul à transmettre sur un slot alors la borne reçoit le paquet sans erreur. Pendant la période descendante la borne wifi diffuse un paquet vers tous les postes pour les informer sur le *feedback* du slot. Le feedback du slot peut prendre trois valeurs *success*, *empty* et *collision*.

- **Success** : un seul poste a émis sur la période montante du slot, la borne déclare le slot *success* et répète le paquet pendant la période descendante pour qu'il soit reçu correctement par tous les postes du réseau ;
- **Empty** : aucun poste n'a émis sur la période montante du slot, la borne a perçu un canal vide (ou blanc).

– **Collision** : plusieurs postes ont émis sur la période montante du slot : les paquets se sont détruits mutuellement et aucun n'a été reçu correctement par la borne. La borne notifie tous les postes du réseau de cet échec.

Notre objectif est de construire une élection de *leader* sur ce type de canal de communication. On suppose que chaque poste a, possède un identifiant id_a qui est une suite binaire de 0 et de 1 de longueur m. L'entier m est le même pour tous les postes et on suppose que $m > \log_2 n$. Pour tout entier $0 \leq k < m$, la quantité $id_a(k)$ est le k-ième bit de id_a. Par la suite on notera $id(k) = id_a(k)$ quand il n'y aura aucune ambiguité sur le poste a.

Tous les identificateurs sont distincts, c'est à dire que si $a \neq b$, alors $\exists k : 0 \leq k < m$ et $id_a(k) \neq id_b(k)$. Par exemple $id_a = 0010$ et $id_b = 0011$ avec $m = 4$.

Chaque poste a une primitive SEND() qui attend le prochain slot et envoie sur la période montante de ce slot un paquet qui contient la séquence id de l'émetteur, et la primitive SKIP() qui attend le prochain slot sans envoyer de paquet sur sa période montante. Il a aussi la primitive GETFEEDBACK() qui retourne le feedback du slot courant notifié par la borne wifi pendant la période descendante du slot. Nous supposons que les postes commencent tous en même temps une procédure d'élection sur un slot donné, appelé slot initial, ou slot numéro zéro. Tous les postes ont alors le status *candidate*. Grâce aux propriétés du canal à collision, le premier poste qui réussira à transmettre son paquet avec succès est le gagnant, puisqu'aucun autre poste n'aura transmis en même temps son paquet. De plus tous les autres postes reçoivent l'identifiant du gagnant grâce au retour de la borne.

Le problème est que si $n \geq 2$ et que tous les postes s'obstinent à retransmettre leur paquet sur chaque slot, alors nous aurons une succession sans fin de collisions. Pour résoudre ce problème il faut mettre en oeuvre une procédure de *réduction* des collisions.

1.2 Codes et propriétés de base

Nous allons étudier la procédure ELECTION() exécutée par tous les postes à partir du slot initial, décrite par le pseudo-code suivant :

```
procedure ELECTION()
 status ← candidate
 SEND()
 feedback ← GETFEEDBACK()
 if feedback = Success  then status ← winner

   else  //  feedback = Collision
 { REDUCTION(0, 0)
 { if status = candidate  then REDUCTION(0, 1)
```

Chaque poste a exécute indépendamment le code suivant où k est un entier et i un entier pris dans $\{0, 1\}$. On rappelle que $id_a(k)$ est le k-ème bit de l'identifiant id_a du poste a.

procedure REDUCTION(k, i)
// pour le poste a
// k et i sont des entiers
if $id_a(k) = i$ **then** SEND() **else** SKIP() // $id_a(k)$ est le k-ème bit de id_a
feedback $\leftarrow$ GETFEEDBACK()
if *feedback* $=$ *Success*
then { **if** $id_a(k) = i$ **then** *status* $\leftarrow$ *winner* ; **else** *status* $\leftarrow$ *lost* }
else {
if *feedback* $=$ *Collision*
then {
if $id_a(k) = 1 - i$ **then** *status* $\leftarrow$ *lost*
else if $k < m - 1$
then { REDUCTION$(k+1, 0)$; **if** *status* $=$ *candidate* **then** REDUCTION$(k+1, 1)$ }
}
}

On remarquera que seuls les postes candidats appellent les procédures REDUCTION(k, i), et que seule une collision appelle REDUCTION$(k+1, j)$, pour $j \in \{0, 1\}$.

A un slot donné, pour un poste candidat donné, on appelle *niveau* la paire (k, i) qui lui a fait appeler REDUCTION(k, i).

Question 1 Montrer qu'à chaque slot tous les postes candidats sont au même niveau. $\diamond$

solution Par récurrence sur les slot. La propriété est vraie pour le slot initial, et l'évolution du niveau ne dépend que de la succession des feedbacks des slots qui sont les mêmes pour tous les postes. A partir de maintenant on appelle niveau d'un slot le niveau de tous les postes encore candidats sur ce slot.

Le processus d'élection avec $id_a = 0010$ et $id_b = 0011$ donne le déroulement suivant :

slot	0	1	2	3	4	5
niveau	-	$(0,0)$	$(1,0)$	$(2,0)$	$(2,1)$	$(3,0)$
feedback	C	C	C		C	$S(a)$
candidats	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\emptyset$

La ligne des niveaux donne le niveau du slot au début de celui ci. Sur la ligne des feedbacks C signifie "collision", un blanc signifie un slot blanc et $S(x)$ signifie la transmission avec succès du paquet du poste x. La ligne des candidats donne l'ensemble des postes candidats à la fin du slot.

La figure 2 montre arbre des appels des procédures REDUCTION(k, i) dans l'exemple décrit précédemment. Les sommets de l'arbre correspondent aux slots et les labels dessous correspondent aux niveaux.

Question 2 On ajoute le poste c avec $id_c = 0000$. Donner le tableau du déroulement de l'élection et le nouvel arbre des appels associés. $\diamond$

solution

slot	0	1	2	3
niveau	-	$(0,0)$	$(1,0)$	$(2,0)$
feedback	C	C	C	$S(c)$
candidats	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$	$\emptyset$

Question 3 Montrer que lorsque les postes candidats appellent REDUCTION(k, i) leurs identifiants ont tous le même préfixe de longueur k, c'est à dire si a et b sont candidats alors $\forall j$ tel que $0 \leq j < k$: $id_a(j) = id_b(j)$. $\diamond$

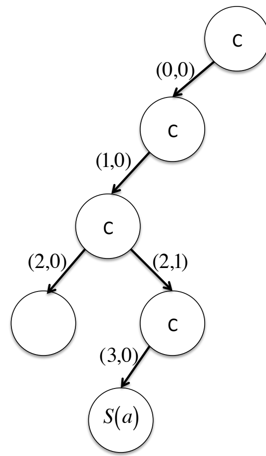

FIGURE 2 – Arbre d'appel de la procédure REDUCTION(k, i) dans la résolution $\{a, b\}$

solution Par récurrence sur l'ordre du slot. La propriété est vraie pour le slot initial et les premiers appels REDUCTION$(0, 0)$ et REDUCTION$(0, 1)$. Supposons la propriété vraie pour l'appel REDUCTION(k, i) : tous les candidats ont le même préfixe de longueur k. Si le slot courant est un succès il n'y a plus de candidats, si c'est une collision, les candidats avec $id(k) = 1 - i$ sont éliminés. Donc à l'appel de REDUCTION$(k+1, 0)$ et de REDUCTION$(k+1, 1)$ les candidats ont le même préfixe de longueur $k+1$.

1.3 Terminaison

On dit qu'un appel REDUCTION(k, i) est décisif si à la fin de l'appel il n'y a plus de candidats avec le status *candidate* et qu'il y a un seul poste avec le status *winner* et que les autres postes ont le status *lost*.

Question 4 Montrer que l'appel REDUCTION(k, i), soit est décisif, soit donne un slot vide. ◇

solution Par récurrence descendante en partant de $k = m-1$. Lors de l'appel REDUCTION$(m-1, 0)$, comme tous les candidats ont le même préfixe de longueur $m-1$, il y a au plus deux candidats : au plus un candidat a avec $id_a(m-1) = 0$ et au plus un candidat b avec $id_b(m-1) = 1$. Si il n'y a pas de candidat a avec $id_a(m-1) = 0$ l'appel donnera un slot vide et terminera sans modifier le status du possible candidat b avec $id_b(m-1) = 1$. Dans ce dernier cas le candidat b avec $id_b(m-1) = 1$ émettra avec succès dans l'appel REDUCTION$(m-1, 1)$ et l'appel sera décisif en éliminant tous les autres candidats.

Lors de l'appel REDUCTION(k, i) avec $k < m-1$, l'absence de candidat avec $id(k) = i$ termine l'appel avec un slot vide, si il y a un seul candidat avec $id(k) = i$, alors l'appel termine en éliminant tous les candidats. Sinon il y a plus d'un candidat avec $id(k) = i$ et on appelle REDUCTION$(k+1, 0)$. Cet appel soit élimine tous les candidats, soit donne un slot vide. Dans le dernier cas c'est que tous les candidats on $id(k+1) = 1$ et dans ce cas l'appel REDUCTION$(k+1, 1)$ ne peut résulter en un slot vide. D'après l'hypothèse de récurrence il est donc décisif.

Question 5 Montrer que l'appel REDUCTION$(k, 1)$ est décisif ◇

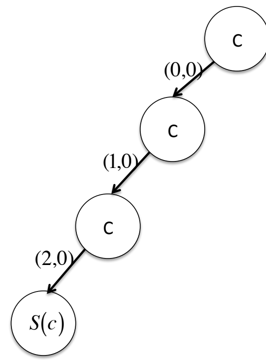

FIGURE 3 – Arbre d'appel de la procédure REDUCTION(k, i) dans la résolution $\{a, b, c\}$

solution L'appel REDUCTION$(k, 1)$ ne peut se faire que si tous les candidats ont tous $id(k+1) = 1$, et dans ce cas l'appel ne peut résulter en un slot vide et donc éliminer tous les candidats.

On dit que la paire (x_1, y_1) est lexicographiquement supérieure à (x_2, y_2) si $x_1 > x_2$, ou $x_1 = x_2$ et $y_1 > y_2$.

Question 6 Montrer que la succession des niveaux des slots avant le slot terminal est une suite lexicographiquement croissante. ◇

solution Dans l'appel REDUCTION(k, i), si il y a un appel à REDUCTION$(k+1, 0)$ cet appel est strictement supérieur. Si il donne un slot vide il sera suivi par un appel à $(k+1, 1)$ qui est supérieur, ou sinon les appels dans REDUCTION$(k+1, 0)$ seront supérieur à $(k+1, 0)$ et il n'y aura pas d'appel à REDUCTION$(k+1, 1)$.

Question 7 Montrer que la procédure ELECTION$()$ est décisive. ◇

solution La suite des niveaux (k, i) des slots est lexicographiquement croissante et est bornée supérieurement par $(m-1, 1)$. Elle doit donc terminer en un nombre fini. Soit (k, i) la plus grande valeur atteinte. Si l'appel REDUCTION(k, i) n'élimine pas les candidats, c'est que $i = 0$, mais dans ce cas il donne un slot vide et tous les candidats appellent REDUCTION$(k, 1)$. Donc $i = 1$ et dans ce cas REDUCTION(k, i) termine en éliminant les candidats.

Question 8 Quelle propriété présente la séquence id_a du vainqueur$a$? ◇

solution C'est la séquence la plus petite dans l'ordre lexicographique. Ou, ecrit en binaire, c'est celui qui représente le plus petit nombre.

1.4 Optimisation de l'élection

Question 9 Donner une borne supérieure du nombre de slots nécessaires pour terminer la procédure ELECTION$()$. ◇

solution Il y a au plus $2m$ slot avant d'atteindre le niveau maximal $(m-1,1)$. Donc il y a $2m+1$ slot en comptant l'appel ELECTION(). En fait on ne peut pas atteindre $(m-1,1)$ puisque la collision en $(m-2,i)$ implique un candidat avec $id(m-1)=0$. Donc la borne supérieure est $2m$, atteinte avec deux postes ayant tout deux le même préfixe de longueur $m-1$ et constitué uniquement de 1.

Question 10 Montrer que le premier slot de REDUCTION$(k,1)$, quand cette procédure est appelée, est toujours une collision. ◇

solution Pour appeler REDUCTION$(k,1)$, il faut que dans l'appel REDUCTION$(k-1,i)$ le premier slot soit une collision, et que l'appel REDUCTION$(k,0)$ ne termine pas en éliminant tous les candidats. Donc l'appel précédent donne un slot vide et tous les candidats se reportent sur l'appel REDUCTION$(k,1)$ qui aura comme premier slot une collision.

Question 11 Ecrire un pseudo-code d'une procédure OPT-REDUCTION(k,i) qui évite la première collision de RESOLUTION(k,i) quand $i=1$. ◇

solution **procedure** OPT-REDUCTION(k,i)

```
next ← false
if i = 1 then next ← true
else {
    if id_a(k) = 0 then SEND() else SKIP()
    feedback ← GETFEEDBACK()
    if feedback = Success
    then {
        if id_a(k) = i then status ← winner
        else status ← lost
    }
    else {
        if feedback = Collision
        then {
            if id_a(k) = 1 then status ← lost
            else next ← true
        }
    }
}
if next = true and k < m − 1
then {
    OPT-REDUCTION(k + 1, 0)
    if status = candidate then OPT-REDUCTION(k + 1, 1)
}
```

Question 12 Donner une borne supérieure du nombre de slots nécessaires pour terminer l'élection optimisée. ◇

solution Il s'agit de $m+1$ slots atteint pour deux postes avec le même préfixe nulle de longueur $m-1$.

1.5 Extension des codes

Nous proposons le pseudo-code suivant sans status *lost*. On notera que la procédure PRINCIPAL() est structurellement comme ELECTION() sauf qu'elle appelle SECONDAIRE(k, i) qui change REDUCTION(k, i) :

procedure PRINCIPAL()
status ← *candidate*
SEND()
feedback ← GETFEEDBACK()
if *feedback* = *Success* **then** *status* ← *winner*
else {
 if *feedback* = *Collision*
 then {
  SECONDAIRE$(0, 0)$
  if *status* = *candidate* **then** SECONDAIRE$(0, 1)$

procedure SECONDAIRE(k, i)
if $id_a(k) = i$ **then** SEND() **else** SKIP()
feedback ← GETFEEDBACK()
if *feedback* = *Success* **and** $id_a(k) = i$ **then** *status* ← *winner*
else {
 if *feedback* = *Collision*
 then {
  SECONDAIRE$(k + 1, 0)$
  if *status* = *candidate* **then** SECONDAIRE$(k + 1, 1)$

Question 13 Que fait ce code ? ◇

solution Question annulée

Nous proposons le pseudo-code séquentiel suivant

procedure DOSOMETHING()
$R \leftarrow 0$
$k \leftarrow 0$
status ← *candidate*
while *status* = $candidate$
do {
 if $R = 0$ **then** SEND()
  else SKIP()
 feedback ← GETFEEDBACK()
 if $R = 0$
 then {
  if *feedback* = *Success* **then** *status* ← *winner*
  if *feedback* = *Collision*
  then {
   $R \leftarrow R + id_a(k)$
   $k \leftarrow k + 1$
 else {
  if *feedback* = *Collision* **then** $R \leftarrow R + 1$
  else $R \leftarrow R - 1$

Question 14 Que fait ce code ?

solution Il fait en sorte que tous les postes transmettent leur paquet dans l'ordre lexicographique de leur id.

Question 15 Modifier ce code pour qu'il fasse la même chose que le code élection. ◇

solution **procedure** ELECTION2()

```
R ← 0
k ← 0
status ← candidate
while status = candidate
do {
  if R = 0  then SEND()
    else SKIP()
  feedback ← GETFEEDBACK()
  if R = 0
    then {
      if feedback = Success  then status ← winner
      if feedback = Collision
        then { R ← R + id_a(k)
               k ← k + 1
    }
    else {
      if feedback = Collision
        then { R ← R + 1
               if R ≥ 2  then status ← lost
      if feedback = Success  then status ← lost
      if feedback =
      empt  then R ← R − 1
    }
}
```

1.6 Election aléatoire sans identificateur fixe

Le canal à collision permet des élections sans identificateur. C'est à dire que les postes n'ont pas d'identificateur permanent et à la fin de l'élection, un seul poste sait qu'il est le gagnant et les autres savent qu'ils sont perdants.

Le principe de l'élection aléatoire est qu'au fur à mesure chaque poste choisit au hasard les bits de la séquence identifiante qu'il utilise dans la procédure RESOLUTION(k, i). En conséquence il n'y a pas de limite sur la longueur de la séquence aléatoire identifiante. Pour cela les postes disposent d'une primitive TOSS() qui donne 0 ou 1 au hasard avec probabilité $(\frac{1}{2}, \frac{1}{2})$. On suppose que la procédure TOSS() est du "vrai" aléatoire" et quelle donne des séquences différentes selon les postes.

Question 16 Donner le pseudo-code de la procédure RAND-REDUCTION(k, i). On prendra soin que chaque poste choisisse utilise la fonction TOSS() une seule fois au plus pour le même bit $id(k)$. ◇

solution **procedure** RAND-REDUCTION(k, i)

bit $\leftarrow 0$
if $i = 0$ **then** *bit* $\leftarrow$ TOSS()
if *bit* $= 0$ **then** SEND() **else** SKIP()
feedback $\leftarrow$ GETFEEDBACK()
if *feedback* $=$ *Success*

$$\textbf{then} \begin{cases} \textbf{if } bit = 0 \ \textbf{ then } status \leftarrow winner \\ \\ \quad \textbf{else } status \leftarrow lost \end{cases}$$

$$\textbf{else} \begin{cases} \textbf{if } feedback = Collision \\ \\ \quad \textbf{then} \begin{cases} \textbf{if } bit = 1 \ \textbf{ then } status \leftarrow lost \\ \\ \quad \textbf{else if } k < m - 1 \\ \\ \quad \textbf{then} \begin{cases} \text{RAND-REDUCTION}(k+1, 0) \\ \textbf{if } status = candidate \ \textbf{ then } \text{RAND-REDUCTION}(k+1, 1) \end{cases} \end{cases} \end{cases}$$

Remarquer que le champ k est inutile.

2 Problème 2

On considère un réseau sur lequel on va résoudre divers problèmes de routage et de configuration. Dans ce problème on donnera des solutions en algorithmique séquentielle classique sauf dans la dernière question, où il s'agira d'une solution distribuée.
Le réseau est représenté par un graphe *orienté* $G = (V, E)$ où V est l'ensemble des stations ($|V| = n$) et $E \subset V \times V$ est l'ensemble des liaisons directes entre stations ($|E| = e$) ; on suppose qu'il n'a pas d'arête de type $\langle x, x \rangle$ pour $x \in V$.
Chaque arête du graphe pourra se voir attribuer une valuation $w(\langle x, x \rangle)$ que l'on pourra interpréter comme une longueur, une bande passante, une couleur, etc. selon les circonstances.

Question 17 Accessibilité
On considère le graphe $G1$ à $n = 8$ sommets défini par $V1 = \{a, b, c, d, e, f, g, h\}$ avec pour arêtes $E1 = \{\langle a, b \rangle, \langle a, c \rangle, \langle b, d \rangle, \langle c, g \rangle, \langle d, a \rangle, \langle d, c \rangle, \langle e, b \rangle, \langle f, e \rangle, \langle h, e \rangle, \langle h, g \rangle\}$ (voir figure 4).

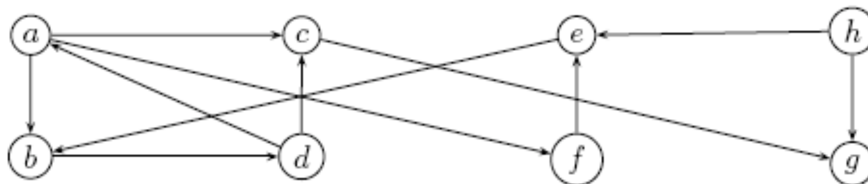

FIGURE 4 – Graphe $G1$

Sur ce réseau, on définit les relations logiques :
— *Arête*(x, y) vraie si et seulement si $\langle x, y \rangle \in E1$;
— *Chemin*(x, y) qui est la plus petite relation définie par
Arête$(x, y) \implies$ *Chemin*(x, y) et
Arête$(x, y) \wedge$ *Chemin*$(y, z) \implies$ *Chemin*(x, z).

La relation *Chemin*(x,y) est donc la fermeture transitive de la relation *Arête*(x,y).
Montrer comment on peut appliquer la *méthode de résolution*, vue en cours, pour prouver que *Chemin*(h,a) est vrai. Comment peut-on prouver que *Chemin*(a,h) est faux ? On donnera dans les deux cas une solution complète.
Comment gère-t-on l'existence de cycles comme (a,b,d,a) ?

solution La relation *Arête*(x,y) prend donc la valeur vrai pour les paires suivantes : *Arête*(a,b), *Arête*(a,c), *Arête*(a,f) (omise dans la liste mais présente sur le dessin), *Arête*(b,d), *Arête*(c,g), *Arête*(d,a), *Arête*(d,c), *Arête*(e,b), *Arête*(f,e), *Arête*(h,e), *Arête*(h,g).

Ces formules produisent selon la méthode de résolution avec la formule universelle *Arête*$(x,y) \implies$ *Chemin*(x,y) une série de 11 formules identiques avec le prédicat *Chemin*(x,y), dont les premières sont : *Chemin*(a,b), *Chemin*(a,c), etc. Dans le premier cas on a unifié $x=a, y=b$ et dans le second $x=a, y=c$, etc.

Le processus se poursuit en calculant les résolvantes de la formule universelle
$\neg$*Arête*$(x,y) \vee \neg$*Chemin*$(y,z) \vee$ *Chemin*(x,z) avec les faits prouvés ci-dessus, par exemple *Arête*(b,d) et *Chemin*(d,a), dont la résolvante est *Chemin*(b,a), avec pour unificateur $x=b, y=d, z=a$. De même avec *Arête*(h,e) et *Chemin*(e,b), on a la résolvante *Chemin*(h,b) ; puis on produit *Chemin*(h,d) et enfin *Chemin*(h,a), ce qui prouve l'existence du chemin.

En procédant de façon systématique, on va donc calculer tous les chemins de longueur 2, puis tous les chemins de longueur 3, etc. On arrête le calcul quand on a calculé tous les chemins de longueur $n-1$, longueur maximale d'un chemin sans boucle.

En l'occurrence on constatera que la formule *Chemin*(a,h) ne fait pas partie de la liste des chemins prouvés et on en déduit que h n'est pas accessible de a.

L'existence de cycles ne pose pas de problème particulier dans cet algorithme. On pourrait l'arrêter plus tôt dès que l'on constate qu'on n'ajoute plus de nouvelle formule au cours d'un tour calculant les chemins de longueur $k+1$.

Question 18 Calcul de la relation

Expliciter le pseudo-code d'un algorithme qui calcule toutes les paires (x,y) telles que *Chemin*(x,y) est vrai : on pourra utiliser une structure intermédiaire qui stocke les résultats acquis. Quel est le coût de cet algorithme ?
Comment l'implémenter en Java ?
Dans une première étape on explicitera les structures de données utilisées par le programme. On supposera que les sommets sont déjà codés comme des entiers de 0 à $n-1$; par exemple pour le graphe de la question 17, le sommet a est codé par 0, b par 1, h par 7.
Dans une seconde étape, on traduira le pseudo-code sur cette structure.
Comment se compare-t-il aux algorithmes du cours ?
On suppose de plus que la fonction w associe à chaque arête son temps de transmission. Comment modifier le code pour qu'il trouve le chemin le plus économique entre chaque paire de points ?

solution Comme indiqué dans la question précédente, la solution la plus simple consiste à lister les chemins par longueur croissante. On crée donc des ensembles `L1` comme vu ci-dessus, puis `L2`, `L3`, etc. en calculant les résolvantes de la formule $\neg$*Arête*$(x,y) \vee \neg$*Chemin*$(y,z) \vee$ *Chemin*(x,z) avec une formule de type *Arête*(α,β), issue de l'ensemble des arêtes, et une formule de type *Chemin*(β,γ) prise dans l'ensemble `Li`, résolvantes qui constitueront l'ensemble `Li+1`. On arrête le processus à `Ln-1`.

D'où le pseudocode :

```
1. Constituer l'ensemble LA des ar\^etes du graphe
2. L1 <-- LA
3. pour i:=1 \`a n-2 faire {
          Li+1 <-- nul;
          pour tous (x,y) dans LA et (y,z) dans Li faire {
```

```
            ajouter (x,z) \'a Li+1
        }
    }
```

Comme chaque ensemble `Li` peut finir par avoir $O(n^2)$ éléments, le coût total peut être $O(n^5)$, ce qui excède le coût de la méthode par produit de matrices.

Améliorations :
1. On peut ne pas retenir les chemins déjà calculés dans une itération précédente. Les `Li` deviennent alors seulement les ensembles de nouveaux chemins trouvés dans chaque itération. Les chemins déjà trouvés sont répertoriés dans une table de hachage ce qui permet un test rapide. Lorsqu'un `Li` s'avère vide à la fin de l'itération, on peut arrêter le calcul.
2. On peut aussi modifier le schéma logique en prenant comme formule logique universelle fondamentale

$$\neg \mathit{Chemin}(x,y) \;\vee\; \neg \mathit{Chemin}(y,z) \;\vee\; \mathit{Chemin}(x,z)$$

et en parcourant deux fois l'ensemble des chemins déjà calculés...
3. On peut aussi prendre le schéma 2 ci-dessus en imposant le point de partage $y = 0, 1, 2, 3, ..., n-1$ successivement. On a alors redécouvert l'algorithme de Floyd-Warshall (!)

Voici des codes Java possibles pour les incrédules...

```
import java.util.Collection;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.Map;

class Couple {
  int x, y;

  Couple(int x, int y) {
    this.x = x;
    this.y = y;
  }

  // fonction indispensable pour l'utilisation d'un Set ou Collection
  public boolean equals(Object o) {
    Couple c = (Couple) o;
    return c.x == x && c.y == y;
  }

  // fonction indispensable pour l'utilisation d'un HashSet
  public int hashCode() {
    return (x * 31) ^ (y * 47);
  }

  public String toString() {
    return "(" + x + ',' + y + ')';
  }
}

public class Ensemble {
  static int a = 0, b = 1, c = 2, d = 3, e = 4, f = 5, g = 6, h = 7;

  // il suffit de changer ici pour choisir le type de structure
  static Collection<Couple> nouvelEnsemble() {
```

```
  // return new LinkedList<Couple>();      // la mauvaise
  return new HashSet<Couple>();       // la bonne
}

// Mauvaise solution, le cumul dans resultat fait exploser la taille des
// listes. L'utilisation de HashSet permet cependant de maintenir une taille
// finale correcte.
static Collection<Couple> fermetureNaive(Collection<Couple> arcs, int p) {
  Collection<Couple> resultat = arcs;
  for (int i = 1; i < p; ++i) {
    Collection<Couple> tmp = nouvelEnsemble();
    for (Couple c1 : arcs)
      for (Couple c2 : resultat)
        if (c1.y == c2.x)
          tmp.add(new Couple(c1.x, c2.y));
    for (Couple c : tmp)
      resultat.add(c);
  }
  return resultat;
}

// Solution plus raisonnable, mais il peut encore y avoir des doublons si
// on a plusieurs chemins. En utilisant des HashSet, on peut maintenir
// en O(n^3.|V|).
static Collection<Couple> fermetureBasique(Collection<Couple> arcs, int p) {
  // p = n-1
  Collection<Couple> resultat = nouvelEnsemble();
  for (Couple c : arcs)
    resultat.add(c);
  Collection<Couple> tmp1 = arcs;
  for (int i = 1; i < p; ++i) { // n-2 ite'rations
    // avec tmp1 = les chemins de longueur i
    // on calcule tmp2 = les chemins de longueur i+1
    Collection<Couple> tmp2 = nouvelEnsemble();
    for (Couple c1 : arcs)
      // O(|V|)
      for (Couple c2 : tmp1)
        // O(n^2)
        if (c1.y == c2.x)
          // Ici on suppose que add de tmp2 supprime les doublons en O(1)
          // indispensable pour rester en O(n^2).
          // Une LinkedList et un test avec tmp2.contains seraient une tre`s
          // mauvaise ide'e qui rajoute un facteur O(n^2), fatal !
          tmp2.add(new Couple(c1.x, c2.y));
    tmp1 = tmp2;
    // et cumul dans resultat
    for (Couple c : tmp2)
      resultat.add(c);
  }
  return resultat;
}

// On gagne un facteur n, car la somme des tailles des tmp2 est en O(n^2)
// complexit\'e en O(n^2.|V|).
```

```
static Collection<Couple> fermetureIncrementale(Collection<Couple> arcs, int p) {
  Collection<Couple> resultat = nouvelEnsemble();
  for (Couple c : arcs)
    resultat.add(c);
  Collection<Couple> tmp1 = arcs;
  for (int i = 1; i < p; ++i) {
    // avec tmp1 = les nouveaux chemins de longueur i
    // on calcule tmp2 = les nouveaux chemins de longueur i+1
    Collection<Couple> tmp2 = nouvelEnsemble();
    for (Couple c1 : arcs)
      for (Couple c2 : tmp1)
        if (c1.y == c2.x) {
          Couple c = new Couple(c1.x, c2.y);
          if (!resultat.contains(c))
            // ok car un HashSet fait contains en O(1)
            tmp2.add(c);
        }
    tmp1 = tmp2;
    // et cumul dans resultat
    for (Couple c : tmp2)
      resultat.add(c);
  }
  return resultat;
}

static Collection<Couple> Roy_Warshall(Collection<Couple> arcs, int n) {
  // construction des tables
  Map<Integer, Collection<Couple>> entrant = new HashMap<Integer,
  Collection<Couple>>();
  Map<Integer, Collection<Couple>> sortant = new HashMap<Integer,
  Collection<Couple>>();
  for (int k = 0; k < n; ++k) {
    entrant.put(k, nouvelEnsemble());
    sortant.put(k, nouvelEnsemble());
  }
  Collection<Couple> resultat = nouvelEnsemble();
  // initialisation des tables
  for (Couple c : arcs) {
    entrant.get(c.y).add(c);
    sortant.get(c.x).add(c);
    resultat.add(c);
  }
  // les 3 boucles en O(n)
  for (int k = 0; k < n; ++k)
    for (Couple c1 : entrant.get(k))
      for (Couple c2 : sortant.get(k)) {
        Couple c = new Couple(c1.x, c2.y);
        entrant.get(c.y).add(c);
        sortant.get(c.x).add(c);
        resultat.add(c);
      }
  return resultat;
}
```

```
  static Collection<Couple> faireCollection(int[][] arcs, int n) {
    Collection<Couple> arcs2 = nouvelEnsemble();
    for (int i = 0; i < arcs.length; ++i)
      arcs2.add(new Couple(arcs[i][0], arcs[i][1]));
    return arcs2;
  }

  public static void main(String[] args) {
    int n = 8;
    int[][] arcs = { { a, b }, { a, c }, { a, f }, { b, d }, { c, g },
        { d, a }, { d, c }, { e, b }, { f, e }, { h, e }, { h, g } };
    Collection<Couple> graphe = faireCollection(arcs, n);
    graphe = fermetureIncrementale(graphe, n-1);
    // graphe = Roy_Warshall(graphe, n);
    System.out.println(graphe.size());
    System.out.println(graphe);
  }

}
```

Mais on n'en demandait pas tant !

Pour obtenir le plus court temps de communication on prendra cette dernière variante (Floyd-Warshall) et retiendra pour chaque nouveau chemin celui de temps minimal.

Question 19 Coloriages et messages

On suppose dans cette question que la valuation donnée aux arêtes est de type qualitatif, par exemple que chaque arête peut être colorée soit en jaune, soit en rouge. On veut ne sélectionner que les chemins qui alternent ces couleurs : une arête rouge, suivie d'une jaune, suivie d'une rouge, etc. ou semblablement en commençant par une arête jaune. Modifier le schéma logique de la question 17 en introduisant les chemins qui commencent par une arête jaune et ceux qui commencent par une arête rouge. L'algorithme de la question 18 permet-il encore de trouver les solutions ? Sinon, l'adapter.

Une autre interprétation de cette situation est de poser sur les arêtes des points "o" et des traits "—", comme en Morse ; une succession d'arêtes "o —" serait alors interprétée comme codant un "a", "— o o o" comme codant un "b", et "— o — o" comme codant un "c". On peut donc écrire des textes et un chemin prend ainsi un sens. Écrire un schéma logique qui ne retient que les chemins sur l'alphabet $\{a, b, c\}$.

solution 1. Chemins alternants

On définit les prédicats

— *AJaune*(x, y) vrai ssi (x, y) est une arête jaune ;

— *ARouge*(x, y) vrai ssi (x, y) est une arête rouge ;

— *CJaune*(x, y) vrai ssi il existe un chemin alternant de x à y commençant par une arête jaune ;

— *CRouge*(x, y) vrai ssi il existe un chemin alternant de x à y commençant par une arête rouge.

Les quatre formules logiques de base sont alors :

$$\mathit{AJaune}(x, y) \implies \mathit{CJaune}(x, y),$$

$$\mathit{ARouge}(x, y) \implies \mathit{CRougee}(x, y),$$

$$\mathit{ARouge}(x, y) \wedge \mathit{CJaune}(y, z) \implies \mathit{CRouge}(x, z).$$

Le principe de l'algorithme reste le même que dans les questions 1 et 2.

La solution la plus simple consiste à lister les chemins par longueur croissante en respectant les couleurs de la première arête. On crée donc des ensembles `LJ1` et `LR1` correspondant aux chemins jaunes et rouges de longueur 1 (à partir des ensembles `LAJ` et `LRJ` , puis `LJ2`, `LR2`, `LJ3`, `LR3`, etc. en calculant les résolvantes des formules $\neg AJaune(x, y) \ \vee \ \neg CRouge(y, z) \ \vee \ CJaune(x, z)$ avec une formule de type $AJaune(\alpha, \beta)$, issue de l'ensemble des arêtes, et une formule de type $CRouge(\beta, \gamma)$ prise dans l'ensemble `LRi`, résolvantes qui constitueront l'ensemble `LJi+1`. On arrête le processus à $n - 1$.

D'où le pseudocode :

```
1. Constituer l'ensemble LA des ar\^etes du graphe
2. LJ1 <-- LAJ; LR1 <-- LAR
3. pour i:=1 \`a n-2 faire {
        LJi+1 <-- nul;
        pour tous (x,y) dans LAR et (y,z) dans LJi faire {
             ajouter (x,z) \`a LRi+1
        };
        LRi+1 <-- nul;
        pour tous (x,y) dans LAJ et (y,z) dans LRi faire {
             ajouter (x,z) \`a LJi+1
        };
   }
```

2. Mots en Morse. Il faut procéder semblablement, et exprimer les suites de "—" et "o" qui font sens. Les arêtes sont donc valuées par les prédicats :
— $ATrait(x, y)$ vrai ssi (x, y) est une arête "—" ;
— $Apoint(x, y)$ vrai ssi (x, y) est une arête "o" ;
— $Ca(x, y)$ vrai ssi il existe un chemin codant la lettre "a" de x à y ;
— $Cb(x, y)$ vrai ssi il existe un chemin codant la lettre "b" de x à y ;
— $Cc(x, y)$ vrai ssi il existe un chemin codant la lettre "c" de x à y ;
Les formules logiques de base sont alors :

$$APoint(x, y) \ \wedge \ ATrait(y, z) \implies Ca(x, z),$$

$$ATrait(x, y) \ \wedge \ APoint(y, z) \ \wedge \ APoint(z, u) \ \wedge \ APoint(u, t) \implies Cb(x, t),$$

$$ATrait(x, y) \ \wedge \ APoint(y, z) \ \wedge \ ATrait(z, u) \ \wedge \ APoint(u, t) \implies Cc(x, t),$$

$$Ca(x, y) \implies Cvalide(x, y),$$

$$Cb(x, y) \implies Cvalide(x, y),$$

$$Cc(x, y) \implies Cvalide(x, y),$$

$$Ca(x, y) \ \wedge \ Cvalide(y, z) \implies Cvalide(x, z),$$

$$Cb(x,y) \ \wedge \ Cvalide(y,z) \implies Cvalide(x,z),$$
$$Cc(x,y) \ \wedge \ Cvalide(y,z) \implies Cvalide(x,z).$$

C'est le prédicat $Cvalide(x,z)$ qui définit les bons chemins dans le graphe.
Le principe de l'algorithme reste le même que dans les questions 1 et 2.

Question 20 Expressions régulières

Peut-on aller encore plus loin et ne retenir que les chemins satisfaisant une expression régulière donnée ? Si c'est possible, expliciter soigneusement et de façon détaillée le processus de construction du schéma logique, à partir de l'expression régulière. Quel est le coût de la validation d'un chemin ? Sinon expliquer l'impossibilité.

solution Solution esquissée : On construit le schéma logique comme à la question précédente, par récurrence sur la structure de la formule ; les formules sont identiques pour les lettres pour les mots il faut donc concaténer les formules de leurs lettres comme on l'a fait pour $Cc(x,y)$ à partir des traits et des points constitutifs. L'union ne pose pas de problème comme on l'a déjà vu, et l'étoile de Kleene d'une expression n'est que sa fermeture transitive dont le schéma a été donné en question 1.

La réponse est donc affirmative. Le coût est celui de l'implantation lourde, plutôt en $O(n^5)$.

Question 21 Voyage en Bidulie

Dans un pays lointain vivent deux communautés A et B un peu bizarres, aux frontières mal définies, et interconnectées par un réseau : pour éviter tout complot à l'intérieur d'une communauté, les messages entre deux membres de A doivent être relayés par au moins un membre de B et vice-versa. Le graphe des relations est donc biparti (voir le cours). Pour aller de A dans A, le message doit obligatoirement suivre un chemin de type $A(BA)^+$, et symétriquement pour B, car il n'existe pas d'arête directe de A dans A ni de B dans B.
Bien sûr, certains agitateurs tentent d'établir des liaisons directes entre membres de A ou entre membres de B. De plus tout citoyen peut décider de changer de communauté à condition de modifier tous ses voisins en conséquence. (S'il était membre de A ses voisins étaient tous de B, et maintenant il est membre de B et ses voisins sont tous de A.)
Il existe donc un service au ministère du Réseau qui doit vérifier en permanence que le graphe du réseau reste biparti, sans pour autant connaître les ensembles A et B a priori, mais seulement à partir de la liste des arêtes à cet instant. Proposer un algorithme pour résoudre ce problème. [On pourra s'inspirer de la question 19, mais ce n'est pas la seule façon de faire. Dans cette question on ne considère qu'un état donné du graphe et pas ses évolutions.]

solution Pour simplifier on suppose que le graphe est non orienté.

La question est donc de déterminer si un graphe est biparti en calculant la partition de ses sommets en les deux ensembles A et B. Pour ce faire, on part d'un sommet a quelconque et élément a priori de A et on calcule les distances des autres sommets à a : le dernier algorithme de la question 2. Ce faisant, on peut trouver un sommet qui serait à distance paire par un chemin et impaire par un autre, ce qui permet de conclure que le graphe n'est pas biparti.
Si tel n'est pas le cas, les sommets à distance paire sont dans A et les autres dans B. S'il reste des sommets on en reprend un au hasard et on recommence sur cette nouvelle composante connexe du graphe.

Question 22 Algorithmique distribuée

On voudrait rendre ces algorithmes distribués en installant dans les sommets du réseau des tables qui permettraient de gérer les routages des questions 19 et 20. Expliquer comment faire s'il existe une solution.
Peut-on gérer la société bidulienne de façon distribuée ? [Cette question est ouverte et admet plusieurs réponses.]

solution Il faut adapter le routage de type Floyd-Warshall décrit dans le cours. On a vu comment cet algorithme a une implantation naturelle dans le formalisme de la programmation logique.
Pour implanter des rooutages alternants, il faudra que chaque station ait pour tout autre sommet une table pour les chemins de première arête rouge et une autre pour les chemins de première arête jaune.
Si le message arrive par une arête jaune il sera dirigé selon la table rouge et sinon selon la table jaune.

La construction des tables peut se faire de fac con distribuée avec l'algorithme de Toueg, version distribuée de Floyd-Warshall.