Epreuve d'informatique

sujet 2004 – Solution

Présentation du sujet

L'objet de la première partie est de trouver un algorithme linéaire pour le problème d'accessibilité dans les graphes ET/OU. Ce problème se retrouve sous diverses formes dans la littérature, par exemple le problème du vide pour les automates d'arbres, le problème de satisfaisabilité d'un ensemble de clauses de Horn propositionnelles ou encore (dans sa version accessibilité répétée) le problème du vide pour les automates de Büchi alternants. Ce problème d'accessibilité est aussi un prototype de problème PTIME-complet. La première partie peut être complétée par des problèmes d'accessibilité répétée de diverses formes, correspondant aux conditions d'acceptance par des automates de mots infinis. Ces questions n'ont pas été incluses pour garder un énoncé court.

La deuxième partie est consacrée aux automates alternants de mots, dans une version très simple. Une définition plus jolie (et aussi plus pratique) des automates alternants permet des transitions vers des combinaisons booléennes positives (ou même arbitraires) d'états. Nous avons préféré ici nous limiter à des conjonctions ou des disjonctions d'états pour éviter d'avoir à utiliser la relation de satisfaction en calcul propositionnel. La question 10 montre que le complémentaire peut être calculé en temps linéaire. L'intersection et l'union peuvent aussi être calculés en temps linéaire, mais ce n'était pas demandé ici car la question, dans le formalisme de l'énoncé, est difficile. La question 11 fait le lien avec la première partie : dans le cas d'un alphabet à une lettre, le problème du vide est un problème d'accessibilité dans un graphe ET/OU. Cette correspondance est utilisée notamment en vérification de modèle, mais avec des automates d'arbres ou des automates de mots infinis. Ici, on pouvait aussi donner une preuve directe, les graphes étant acycliques. La question 12 montre que les automates alternants ont même pouvoir d'expression que les automates non-déterministes (et aussi les automates déterministes dans le cas des mots finis qui est celui de l'énoncé). Cependant, comme le montre la question 13, ils peuvent permettre une représentation exponentiellement plus succincte. En fait, le résultat de la question 13 se généralise à des automates déterministes arbitraires, mais la question n'est pas posée par souci de simplicité. (On peut se reporter à l'article de synthèse suivant et aux références qui y sont mentionnées : Sheng Yu, *Regular Languages* in Handbook of formal languages, Rozenberg and Salomaa eds, Springer Verlag, 1997).

1 Accessibilité dans les graphes ET/OU

Les trois premières questions avaient pour objet de montrer l'existence de l'application $A_{\mathcal{G}}$ compatible avec $\mathcal{G}$ et telle que, pour toute application g compatible avec $\mathcal{G}$ et tout sommet

s, $A_{\mathcal{G}}(s) \subseteq g(s)$. Une autre preuve d'existence est donnée dans la question 5, en construisant un algorithme.

Malheureusement, la question 2 est grossièrement fausse (ainsi que la question 1). D'autre part, sans s'appuyer sur la question 2, la question 3, bien que correcte, devient trop difficile. Ceci n'a pas de conséquence sur les questions 4 à 7 puisque, précisément, elles donnent d'autres preuves du même résultat. La façon de corriger les copies et de tenir compte de ces problèmes est détaillée dans le rapport du jury.

Question 1

Si l'un des sommets "ou" n'a pas de successeurs et le graphe comporte au moins deux sommets, alors l'application qui à tout sommet associe V n'est pas compatible. Sinon, il suffit de vérifier que, pour tout $s \in V$,

$$V = \{s\} \cup \bigcup_{(s,s') \in E} V = \{s\} \cup \bigcap_{(s,s') \in E} V$$

Question 2

Un contre-exemple simple est donné par le graphe suivant, dont tous les noeuds sont des noeuds OU : $V = \{1, 2, 3, 4\}, E = \{(1, 2), (3, 2), (3, 4), (2, 2), (4, 4)\}$ et les fonctions g_1 et g_2 :

	1	2	3	4
g_1	$1, 2$	$1, 2$	$1, 2, 3, 4$	4
g_2	$1, 2$	2	$1, 2, 3, 4$	$1, 4$

g_1 et g_2 sont compatibles, en effet : $g_i(1) = \{1\} \cup g_i(2)$, $g_i(2) = \{2\} \cup g_i(2)$, $g_i(3) = \{3\} \cup g_i(2) \cup g_i(4)$, $g_i(4) = \{4\} \cup g_i(4)$, pour $i = 1, 2$.

Par contre, $g_1 \cap g_2(3) = \{1, 2, 3, 4\}$ alors que $\{3\} \cup (g_1 \cap g_2(2)) \cup (g_1 \cap g_2(4)) = \{2, 3, 4\}$. Ce qui montre que $g_1 \cap g_2$ n'est pas compatible.

Le raisonnement (inexact) initialement dans le corrigé est le suivant (dans lequel on commet l'erreur classique de ne pas renommer la variable liée) :

Pour un noeud $\vee$:

$$\begin{aligned} g_1 \cap g_2(s) &= g_1(s) \cap g_2(s) = (\{s\} \cup \bigcup_{(s,s') \in E} g_1(s')) \cap (\{s\} \cup \bigcup_{(s,s') \in E} g_2(s')) \\ &= \{s\} \cup \bigcup_{(s,s') \in E} (g_1(s') \cap g_2(s')) \\ &= \{s\} \cup \bigcup_{(s,s') \in E} g_1 \cap g_2(s') \end{aligned}$$

Question 3

Si l'on admet la question précédente, on peut faire le raisonnement qui suit :

Il suffit de choisir $A_{\mathcal{G}} = g_1 \cap \ldots \cap g_N$ si $g_1, \ldots, g_N$ sont les applications compatibles : par la question 1, il en existe au moins une, et donc $A_{\mathcal{G}}$ est défini. Par la question 2 (et par récurrence sur N) $A_{\mathcal{G}}$ est compatible. Par définition, $A_{\mathcal{G}}(s) \subseteq g_i(s)$ pour tout i et tout s, ce qui implique en particulier l'unicité.

Sinon, la question reste vraie, mais difficile. En fait, la question 5 est une façon d'y répondre.

Question 4

Bien entendu, dans chaque question, seule une des deux représentation devait être considérée. Nous donnons cependant les algorithmes dans les deux cas, par souci de complétude.

1. ```
   let rec member = function x → function
                    [] → false
                    | y :: l → (x=y) or (member x l) ;;
   ```
   Il s'agit simplement d'un parcours de liste. Le coût est $O(|L|)$, longueur de la liste dans le pire des cas.

   Pour les tableaux de Booléens : function member(x,L) = L(x).

   Le coût est $O(1)$.

2. ```
   let rec ens-incl = function
                    [] → (function l → true )
                    | x m → function l → (member x l) & (ens-incl m l) ;;
   ```
   ```
   let ens-eq = function l → function m → (ens-incl l m) and (ens-incl m l)
   ```
   Le test d'inclusion parcourt la liste L effectuant $|L|$ appels à member : son coût est $O(|L| \times |M|)$. Il en est de même pour ens-incl qui teste la double inclusion.

   Pour les tableaux de Booléens :

   ```
   function ens-eq(L,M)
   r := true ;
   i := 1 ;
   Tant que r and i ≤ N Faire
           r := r and (L(i)=M(i))
           i := i+1
   Fin Tq ;
   return r
   ```
   Le coût est $O(|V|)$.

3. ```
   let rec union =function
                    [] → (function x → x)
                    | x ::l → function m → if (member x m)then (union l m)
                                                          else x :: (union l m)
   ;;
   ```
   On parcourt la première liste et pour chacun de ses éléments on teste s'il est dans la deuxième et, le cas échéant, on ajoute un élément au résultat. Dans le pire des cas, le coût est $O(|L| \times |L'|)$.

   Pour les tableaux de Booléens :

   ```
   function union(L,M)
   R := init(false ) ;
   for i :=1 to  N do
      R(i) := L(i) or M(i) ;
   return R
   ```
   Le coût est $O(|V|)$.
   ```

4. let rec gunion = function g → function
```
                    [] → []
                    | x::l → union (gunion g L) (g x)
;;
```
Le coût est égal, dans le pire des cas, si $L = [x_1; \ldots ; x_n]$, à $|g(x_{n-1})| \times |g(x_n)| + |g(x_{n-2})| \times (|g(x_{n-1})| + |g(x_n)|) + \ldots + |g(x_1)| \times (|g(x_2)| + \ldots + |g(x_n)|)$: on montre, par récurrence sur $n = |L|$, que le coût est $O(|L| \times |g|^2)$.

Pour les tableaux de Booléens :

```
function gunion(S,g)
R := init (false )
for i :=1 to  N do
    if S(i) then R := union(R, g(i)) return R
```

Le coût est $O(|V|^2)$.

5. On suppose dans la question que V est une variable globale.

L'intersection de deux ensembles et l'intersection de tous les éléments d'un ensembles d'ensembles sont définies de manière analogue à l'union, avec les mêmes complexités :

```
let rec inter =function
                [] → (function x → [])
                | x::l → function m → if (member x m) then x ::(inter l m)
                                                      else (inter l m) ;;
```

```
let rec ginter =function g → function
                [] → V
                |x :: L1 → (inter (ginter g L1) (g x))
```

La fonction auxilliaire suivante, étant donnée g, G, f et une liste de sommets L, teste la compatibilité de g pour tous les sommets de L en appliquant la définition. Il suffit d'utiliser cette fonction avec $L = V$.

```
let is-compatible = function g → function G → function f →
    let rec aux = function
          [] → true
          x :: L → (if f(x) = ∨ then (ens-eq (union [x] (gunion g (G(x)))) (g x))
                                  else (ens-eq (union [x] (ginter g (G (x)))) (g x)))
                   and (aux L)
    in (aux V)
```

Coût : pour chacun des sommets de $v \in V$ on effectue :
– un appel à f, un appel à G et un appel à g (coût = 3)
– un appel à gunion (ou ginter) de complexité $O(|\mathsf{G}(v)| \times |g|^2)$
– un appel à union, de complexité $O(|g|)$
– un appel à ens-eq, de complexité $O(|g|^2)$
Au total, le coût est donc $O(|\mathcal{G}| \times |g|^2)$.

Dans le cas des tableaux de Booléens

```
function inter (S1,S2)
R := init(true ) ; for i=1 to N do
          R(i) := S1(i) and S2(i) ; return R

function ginter(S,g)
R := init(true ) ;
for i :=1 to N do if S(i) then R := inter(R,g(i)) ;
return R

function singleton (i)
R := init(false ) ; R(i) := true ;
return R

function is-compatible(g,G,f) :
          res := true ; i := 1 ;
          Tant que i≤ N and res do
                    if f(i)= ∨
                          then res := res and ens-eq(union (singleton(i), gunion (G[i],g)), g(i))
                          else res := res and ens-eq(union(singleton(i),ginter (G[i],g)), g(i))
          Fin Tq ;
return res
```

Coût : $|V| \times (|V| + |V|^2) = O(|V|^3)$.

Question 5

1.

	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
1	1, 3	1, 2	3, 4, 6	4	5, 8	6	7, 8	8, 3	6, 8, 9
2	1, 3, 4, 6	1, 2, 3	3, 4, 6	4	3, 5, 8	6, 8	3, 7, 8	3, 4, 6, 8	3, 6, 8, 9
3	1, 3, 4, 6	1, 2, 3, 4, 6	3, 4, 6, 8	3, 4	3, 4, 5, 6, 8	3, 6, 8	3, 4, 6, 7, 8	3, 4, 6, 8	3, 4, 6, 8, 9
4	1, 3, 4, 6, 8	1, 2, 3, 4, 6	3, 4, 6, 8	3, 4, 6	3, 4, 5, 6, 8	3, 4, 6, 8	3, 4, 6, 7, 8	3, 4, 6, 8	3, 4, 6, 8, 9
5	1, 3, 4, 6, 8	1, 2, 3, 4, 6, 8	3, 4, 6, 8	3, 4, 6	3, 4, 5, 6, 8	3, 4, 6, 8	3, 4, 6, 7, 8	3, 4, 6, 8	3, 4, 6, 8, 9
6	1, 3, 4, 6, 8	1, 2, 3, 4, 6, 8	3, 4, 6, 8	3, 4, 6, 8	3, 4, 5, 6, 8	3, 4, 6, 8	3, 4, 6, 7, 8	3, 4, 6, 8	3, 4, 6, 8, 9

2. Les suites $A_n(s)$ sont croissantes (pour l'inclusion). Pour chaque sommet s, l'ensemble des entiers n tels que $A_n(s) \neq A_{n+1}(s)$ a donc au plus $|V| - 1$ éléments. L'ensemble des entiers n tels que $\exists s, A_n(s) \neq A_{n+1}(s)$ est donc borné par $|V| \times (|V| - 1)$. Comme par ailleurs, si $A_n(s) = A_{n+1}(s)$ pour tout s, alors, pour tout $k \geq n$ et pour tout s, $A_k(s) = A_{k+1}(s)$, on conclut que $M = |V| \times (|V| - 1)$ convient.
3. Tout d'abord A est compatible avec $\mathcal{G}$, en effet, pour tout sommet s, si $f(s) = \vee$,

$$A(s) = A_{M+1}(s) = A_M(s) \cup \bigcup_{(s,s') \in E} A_M(s) = A(s) \cup \bigcup_{(s,s') \in E} A(s)$$

puisque $A_{M+1} = A_M = A$. De même si $f(s) = \wedge$.
Par ailleurs, si g est une application compatible avec $\mathcal{G}$, on montre par récurrence sur n que, pour tout n, pour tout s, $A_n(s) \subseteq g(s)$: $A_0(s) = \{s\} \subseteq g(s)$ puisque $g(s) =$

$\{s\} \bigcup_{(s,s')\in E} g(s')$ ou $g(s) = \{s\} \bigcup_{(s,s')\in E} g(s')$. Si la propriété est vraie au rang n, alors, pour tout s, ou bien $f(s) = \vee$, et, dans ce cas,

$$A_{n+1}(s) = A_n(s) \cup \bigcup_{(s,s')\in E} A_n(s') \subseteq g(s) \bigcup_{(s,s')\in E} g(s')$$

par hypothèse de récurrence. D'autre part, $\bigcup_{(s,s')\in E} g(s') \subseteq g(s)$ par compatibilité de g et donc $A_{n+1}(s) \subseteq g(s)$.

De même, si $f(s) = \wedge$,

$$A_{n+1}(s) = A_n(s) \cup \bigcap_{(s,s')\in E} A_n(s') \subseteq g(s) \cup \bigcap_{(s,s')\in E} g(s') = g(s)$$

Il en résulte que $A(s) \subseteq A_{\mathcal{G}}(s)$.

D'après la définition de $A_{\mathcal{G}}$ on a donc $A = A_{\mathcal{G}}$.

4. Nous donnons deux versions, l'une purement fonctionnelle et l'autre utilisant des tableaux d'ensembles. Bien entendu, il y a beaucoup d'autres solutions utilisant d'autres représentations. L'important étant d'avoir un algorithme correct et polynômial.

   On calcule la suite A_n jusqu'à stabilisation : A_0 est initialisée par l'identité. A_1 est calculé à partir de la relation de récurrence. Tant que les deux éléments de la paire (A_{n+1}, A_n) sont distincts, on calcule (A_{n+2}, A_{n+1}).

   Plus précisément, en CAML :

```
let AG = function (S,G,f) →
    let Next = function A → s →
        if f(s) = ∨ then gunion A (G s) else ginter A (G s)
    in
    let A0 = function x → x
    in let A1 = Next A0
    in let Diff = function (A,B) →
        let rec sdiff = function
                [] → false
                s : : L → if ens-eq (A s) (B s) then sdiff L
                                                 else true
        in
        sdiff S
    in
    let rec Iter = function (A,B) →
        if Diff (A,B)then Iter (Next A, A)
                    else A
    in
    Iter (A1,A0)
```

   Coût : Il y a au plus $O(|V|^2)$ itérations, comme vu à la question 5.3. À chaque itération, on appelle Diff et Next. Diff fait au plus $|V|$ appels à ens-eq, lui-même de complexité au plus $O(|V|^2)$. Comme $|A| \leq |V|^2$, Next a pour coût (d'après la question 4) au plus $|V|^5$, ce qui donne un coût total en $O(|V|^7)$. (NB : une analyse plus fine montre que c'est en fait en $O(|\mathcal{G}|^5)$).

Avec des tableaux de Booléens, en langage algorithmique, A et B étant des tableaux d'ensembles de sommets :

```
Données : G, f
procedure Next (A,B)          % Si A = A_n, calcule dans B A_{n+1}
        for i :=1 to N do
            if f(i) = ∨ then
                            B(i) := union(A(i), gunion(G(i),A))
                      else  B(i) := union (A(i), ginter(G(i),A))
end (procedure Next)

flag := false          % drapeau indiquant si A_n = A_{n+1}
for i :=1 to  N do           % Calcul de A_0
    begin
        A(i) := init(false )
        A(i)(i) := true
    end
Tant que not flag faire           % Tant que A_n ≠ A_{n+1}
        begin
            Next(A,B)
            flag := true
            for i :=1 to N do
                flag := flag and ens-eq (A(i),B(i)) ;
            for i :=1 to N do
                A(i) := B(i)
        end
Fin Tq
return A
```

Le coût de Next est en $O(|V|^3)$. Comme il y a au plus $O(|V|^2)$ itérations, le coût total est en $O(|V|^5)$.

Question 6

On calcule $A_{\mathcal{G}}$ comme en 5.4. Puis il suffit, pour chaque $t \in V$ d'effectuer le test du vide de $S \cap A_{\mathcal{G}}(t)$. Le coût est celui du 5.4. ($O(|V|^5)$ ou $O(|\mathcal{G}|^5)$ ou $O(|V|^7)$) plus le coût du vide de l'intersection pour chaque élément de V, soit $|V|^2 \times |S|$, de toutes façons majoré par $|V|^5$ (ou $|\mathcal{G}.|^5$ ou $O(|V|^7)$).

```
let invAG = function S →
        function (V,G,f) →
        let A = AG (V,G,f) in
        let rec aux = function
                [] → []
                t::L → if (inter S AG(t)) <> [] then t::(aux L)
                                                 else (aux L)
        in
        (aux V)
```

Avec des tableaux de Booléens :

```
function invAG(S,G)
        R := init(false ) ;
        A := AG(G,f) ;
        for i :=1 to N do
            R(i) := nonvide (inter (S, AG(i))) ;
        return R
```

où nonvide est la fonction :

```
function nonvide(S)
        r := false
        for i :=1 to N do
            r := S(i) or r ;
        return r
```

Question 7

1. Tout d'abord, on montre que, pour tout $s \in V$, s ne peut être ajouté qu'au plus une fois à S : $n(s)$ est décroissant au sens large et, d'autre part, si s est ajouté à S (ligne 9), $n(s)$ vaut 1 à l'entrée de la boucle interne, et 0 à la sortie de la boucle interne.
   Chacune des instructions de la boucle interne est donc exécutée au plus $\sum_{s \in V} |prec(s)|$ fois, soit $|E|$ fois. La boucle externe est exécutée au plus $|V|$ fois. Au total, le coût de l'algorithme est ainsi au plus $O(|V|) + O(|E|)$ soit $O(|\mathcal{G}|)$.
2. Montrons maintenant que, pour tout élément $s \in S \cup T$, $s_0 \in A_{\mathcal{G}}(s)$ et que, pour tout $s \notin S \cup T$ tel que $f(s) = \wedge$, $n(s) = |G(s)| - |G(s) \cap T|$ après chaque éxécution de la boucle interne. Pour cela, on raisonne par récurrence sur le nombre d'exécutions de la boucle externe. C'est vrai initialement.
   Supposons donc maintenant que $s \in S \cup T$, $s_0 \in A_{\mathcal{G}}(s)$, et $n(s_1) = |G(s_1)| - |G(s_1) \cap T|$ dès que $f(s) = \wedge$ à l'entrée de la boucle externe. Soient T', S', n' les nouvelles valeurs de T, S, n à la sortie : $T' = s :: T$ et, pour tout $s_1 \in \mathsf{prec}(s)$, $n'(s_1) = n(s_1) - 1$. Si $f(s_1) = \wedge$. On a donc $G(s_1) \cap T' = (G(s_1) \cap T) \cup \{s\}$ et donc $|G(s_1)| - |G(s_1) \cap T'| = |G(s_1)| - |G(s_1) \cap T| - 1 = n(s_1) - 1$ (T' ne contient pas de répétition), soit $n'(s_1) = |G(s_1)| - |G(s_1) \cap T'|$.
   Par ailleurs, si $s_1 \in S' \setminus S$, alors $n(s_1) = 1$ et, ou bien $f(s_1) = \vee$ et dans ce cas, comme $(s_1, s) \in E$ et $s_0 \in A_{\mathcal{G}}(s)$, on a aussi $s_0 \in A_{\mathcal{G}}(s_1)$, par compatibilité de $A_{\mathcal{G}}$. Ou bien $f(s) = \wedge$. Dans ce cas, $n'(s_1) = 0$ entraîne $G(s_1) = G(s_1) \cap T'$ et donc $G(s_1) \subseteq T'$.Or, par hypothèse de récurrence, pour tout $s_2 \in T'$, $s_0 \in A_{\mathcal{G}}(s_2)$ donc pour tout $s_2 \in G(s_1)$, $s_0 \in A_{\mathcal{G}}(s_2)$. Par compatibilité de $A_{\mathcal{G}}$, $A_{\mathcal{G}}(s_1) \supseteq \bigcap_{s_2 \in G(s_1)} A_{\mathcal{G}}(s_2)$ et donc $s_0 \in A_{\mathcal{G}}(s_1)$.
   Réciproquement, si $s \in A_{\mathcal{G}}(s_0)$, soit n le plus petit entier tel que $s \in A_n(s_0)$ (il existe d'après la question 5.3). On montre par récurrence sur n que $s \in T$ après execution de l'algorithme. Pour cela, il suffit de montrer que s est ajouté à S à l'une des étape de calcul. Si $n = 0$, alors $s = s_0$ et donc $s \in S$ (au départ). Si la propriété est vraie pour $p < n$ et $n > 0$ supposons d'abord que $f(s) = \vee$. Dans ce cas, $A_n(s) = A_{n-1}(s) \cup \bigcup_{(s,s') \in E} A_{n-1}(s')$ et, par minimalité de n, $s_0 \in A_{n-1}(s')$ pour au moins un sommet s' tel que $(s, s') \in E$. Par hypothèse de récurrence, s' est ajouté à S à une étape de

l'algorithme. D'autre part, par définition, $s \in prec(s')$. Lorsque s' est ajouté à T, la boucle suivante ajoute s à S (puisque $n(s)$ est initialement à 1 et n'est jamais passé par 0).

Si maintenant $f(s) = \wedge$, $A_n(s) = A_{n-1}(s) \cup \bigcap_{(s,s') \in E} A_{n-1}(s')$ et, par minimalité de n, $s_0 \in A_{n-1}(s')$ pour tout s' tel que $(s, s') \in E$. Par hypothèse de récurrence, chacun des sommets s' est ajouté à S à l'une des étapes de l'algorithme. Chaque fois que l'un des sommets s' est ajouté à T, $n(s)$ est décrémenté (et seulement à ces occasions). Lorsque le dernier des sommets s' est ajouté à T, $n(s) = 0$ et s est ajouté à S.

3. On initialise prec(s) à [] pour tout s. Puis, pour $s \in V$ et pout $x \in G[s]$, on ajoute s à prec(x). Le coût est de $k \times |G| = O(|\mathcal{G}|)$. prec(s) ne contient pas de répétition car, étant donné s, $G[s]$ est sans répétition. Pour $n(s)$ on fait un autre parcours de $\mathcal{G}$.

   En langage algo : Dans toute la suite, on abrégera

```
L := V
Tant queL <> [] faire
        s := car(L)
        L := cdr(L)
        i
Fin Tq
```

   dans lequel i ne modifie pas L, par

```
    Pour s ∈ L Faire i
```

   Le programme s'écrit alors :

```
Pour v ∈ V Faire Prec(v) := []
Pour v ∈ V Faire n(v) := 0
Pour v ∈ V Faire
     Pours ∈ G(v) Faire
          prec(s) := v ::prec(s)
          if f(v) = ∧ then n(v) := n(v) +1 else n(v) := 1
```

   Dans le style fonctionnel :

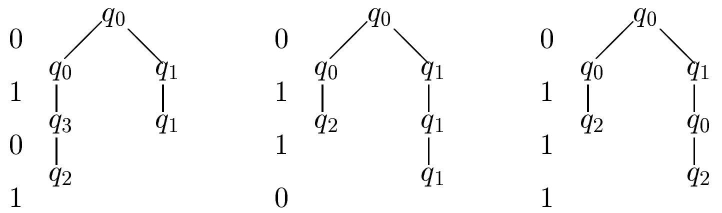
```
let prec = init []
in
let n = init 0
in
let rec CompPrec = function p → function
      [] → p
      v::L → let rec aux = function q → function
                  [] → q
                  s::M → aux (modif q s (v ::(q (s)))) M
             in
             aux p (G v)
in CompPrec prec V
```

   où modif est la fonction (supposée exister d'après l'énoncé) qui renvoie la fonction q modifiée en s par la nouvelle valeur.

4. Il suffit de mettre ensemble les questions 7.1, 7.2 et 7.3 : On calcule d'abord prec et n par 7.3. en temps $O(|\mathcal{G}|)$, puis on applique l'algorithme proposé qui, d'après 7.1. s'exécute à nouveau en $O(|\mathcal{G}|)$. D'après 7.2, à l'issue de l'exécution, $T = A_{\mathcal{G}}^{-1}(\{s_0\}) = \{t \in V | s_0 \in A_{\mathcal{G}}(t)\}$. Il suffit enfin de tester l'appartence du sommet donné à cet ensemble.

2 Automates alternants

Question 8



Question 9

$\mathcal{A}_2$ a même alphabet, mêmes états, même état initial et mêmes états finaux que $\mathcal{A}_1$. $\tau(q,a) = \vee$ pour tous q, a et $\delta_2(q,a) = \delta_1(q,a)$.

Il suffit donc de montrer que les automates reconnaissent le même langage. Montrons, par récurrence sur $|w|$ que, pour tout q, w est accepté par $\mathcal{A}_1$ avec comme seul état final q ssi w est accepté par $\mathcal{A}_2$ avec comme seul état final q. Comme $\mathcal{A}_2$ ne comporte que des états de type $\vee$, il existe un calcul réussi de $\mathcal{A}_2$ sur w avec comme état final q, si et seulement ce calcul est en fait une liste : chaque noeud a exactement un fils, excepté le noeud de profondeur $|w|$ qui est étiqueté par q. Si q' est l'état étiquetant le noeud de profondeur $n-1$, il suffit alors d'appliquer l'hypothèse de récurrence.

Le cas de base (w est le mot vide) est une conséquence de l'identité des états initiaux.

Question 10

On considère $\mathcal{A}^c$ comme dans l'énoncé : états finaux et non finaux échangés et types des transitions échangés (le calcul est bien en temps linéaire). On montre alors par récurrence sur $|w|$ que, pour tout q, si w est accepté par $\mathcal{A}$ à partir de q initial, alors w n'est pas accepté par $\mathcal{A}^c$ à partir de l'état initial q.

Si $w = \epsilon$, cela résulte de l'échange des états finaux et non finaux.

Si maintenant t est un calcul réussi de $\mathcal{A}$ sur $w = a \cdot w_0$ à partir de q, deux cas se présentent. Si $\tau(q) = \wedge$ et $\delta(q,a) = q_1 \wedge \ldots \wedge q_m$, alors, par hypothèse de récurrence, il n'existe aucun calcul réussi de $\mathcal{A}^c$ sur w_0. Or un calcul de $\mathcal{A}^c$ sur w aura sa racine étiquetée par q et un seul fils (puisque $\tau^c(q) = \vee$) étiqueté par l'un des q_i. Ce ne peut donc pas être un calcul réussi. Si $\tau(q) = \vee$, pour au moins un état $q' \in \delta(q,a)$, il existe un calcul réussi de $\mathcal{A}$ sur w_0 à partir de q'. Donc, par hypothèse de récurrence, il existe au moins un état $q' \in \delta^c(q,a)$ tel qu'il n'y a aucun calcul réussi de $\mathcal{A}^c$ sur w_0 à partir de q'. Comme $\tau^c(q,a) = \wedge$, pour tout calcul de $\mathcal{A}^c$ sur w, l'un des fils de la racine est étiqueté par q'. Ce calcul ne peut donc pas être réussi.

Comme $(\mathcal{A}^c)^c = \mathcal{A}$, on déduit de ce qui précède que les langages reconnus par $\mathcal{A}$ et $\mathcal{A}^c$ sont disjoints. Reste à montrer que tout mot est accepté par l'un des deux automates. On procède à nouveau par récurrence sur w : on montre que, pour tout état q et tout mot w, w est accepté à partir de q par l'un des deux automates. Dans le cas de base, il suffit de remarque que q est un état final de l'un des deux automates.

Soit maintenant $w = a \cdot w_0$. Si $\tau(q,a) = \wedge$, ou bien pour tout état q' de $\delta(q,a)$, il existe un calcul réussi de $\mathcal{A}$ sur w_0 à partir de q, ou bien, par hypothèse de récurrence, il existe un état $q' \in \delta(q,a)$ et un calcul réussi de $\mathcal{A}^c$ sur w_0 à partir de q'. Dans le premier cas on construit un calcul réussi de $\mathcal{A}$ sur w. Dans le deuxième cas on construit un calcul réussi de $\mathcal{A}^c$ sur w.

Si $\tau(q,a) = \vee$, ou bien il existe un état q' de $\delta(q,a)$ tel que w_0 est accepté par $\mathcal{A}$ à partir de q' (et w est accepté par $\mathcal{A}$), ou bien pour tout état $q' \in \delta(q,a)$, w_0 est accepté par $\mathcal{A}^c$ à partir de q' et dans ce cas, w est accepté par $\mathcal{A}^c$.

Question 11

On construit un graphe ET/OU à partir de w et $\mathcal{A}$ comme suit : Les sommets sont $V = \{(q,i) \mid q \in Q, 1 \leq i \leq |w|\} \cup \{(q_0,0)\} \cup \{r\}$. Pour $0 \leq i \leq |w|-1$ et $(q,i) \in V$, $\mathsf{G}(q,i) = \{(q_1,i+1),\ldots,(q_m,i+1)\}$ si $\delta(q,w(i+1)) = \{q_1,\ldots,q_m\}$, $\mathsf{f}(q,i) = \tau(q,w(i+1))$. $\mathsf{G}(q,|w|) = \{r\}$ et $f(q,|w|) = \vee$, si $q \in Q_f$ et est vide sinon. $G(r) = \{r\}$ et $f(r) = \vee$.

Montrons que r est accessible dans $\mathcal{G}$ à partir de $(q_0,0)$ si et seulement si w est accepté par $\mathcal{A}$. Il suffira alors d'appliquer la question 7.4.

On montre ce résultat par récurrence sur la longueur de w. Si w est le mot vide, $\mathcal{A}$ accepte w si et seulement si q_0 est un état final, si et seulement si il existe une arête de $(q_0,0)$ vers r. Si maintenat $w = a \cdot w_0$. Si $\tau(q_0,a) = \vee$, w est accepté par $\mathcal{A}$ si et seulement si il existe un état $q \in \delta(q,a)$ tel que w_0 est accepté par $\mathcal{A}$ à partir de q, si et seulement si, par hypothèse de récurrence, $r \in A_{\mathcal{G}}(q,1)$ (en fait, pour être rigoureux, $r \in A_{\mathcal{G}'}(q,0)$ où $\mathcal{G}'$ est obtenu en se restreignant aux sommets de $\mathcal{G}$ dont la deuxième composante est non nulle et en retranchant 1 à toutes les deuxièmes composantes). Mais, par compatibilité de $A_{\mathcal{G}}$, $r \in A_{\mathcal{G}}(q,1)$ entraîne $r \in A_{\mathcal{G}}(q_0,0)$ et, par minimalité de $A_{\mathcal{G}}$, si $r \in A_{\mathcal{G}}(q_0,0)$, alors $r \in A_{\mathcal{G}}(q,1)$ pour $(q,1) \in \mathsf{G}(q,0)$. Dans le cas où $\tau(q_0,0) = \wedge$ le raisonnement est semblable.

Noter que dans ce cas particulier, $\mathcal{G}$ est sans cycle. En fait, on se sert des graphes cycliques pour les applications aux automates de mots infinis, par exemple.

Question 12

L'ensemble des états de $\mathcal{A}_2$ est 2^Q, l'état initial est $\{q_0\}$, les états finaux sont les parties non vides de Q_f. Les transitions sont données par : $(S,a,S') \in \delta_2$ si et seulement si, $S' \supseteq \delta(q,a)$ pour tout état $q \in S$ tel que $\tau(q,a) = \wedge$ et $S' \cap \delta(q,a) \neq \emptyset$ pour tout état $q \in S$ tel que $\tau(q,a) = \vee$.

Montrons que, s'il existe un calcul réussi de $\mathcal{A}_1$ sur w à partir de chacun des états $q_1,\ldots,q_m$, alors $\mathcal{A}_2$ accepte w à partir de $\{q_1,\ldots,q_m\}$, par récurrence sur $|w|$. Si w est le mot vide, alors $q_1,\ldots,q_m \in Q_f$, d'où le résultat puisque $\{q_1,\ldots,q_m\}$ est final dans $\mathcal{A}_2$.

Si maintenant $w = a \cdot w_0$, pour tout état q_i tel que $\tau(q_i,a) = \vee$, il existe un état $q'_i \in \delta(q,a)$ et un calcul réussi de $\mathcal{A}_1$ sur w_0 à partir de q'_i. Pour chaque état q_i tel que $\tau(q_i,a) = \wedge$ et chaque état $q' \in \delta(q_i,a)$, il existe un calcul réussi de $\mathcal{A}_1$ sur w_0 à partir de q'. Par hypothèse de récurrence, il existe donc un calcul réussi de $\mathcal{A}_2$ sur w_0 à partir de $\{q'_i \mid \tau(q_i,a) = \vee\} \cup \bigcup_{j,\tau(q_j,a)=\wedge} \delta(q_j,a)$. De plus, par définition de $\mathcal{A}_2$, il existe une transition par a depuis $\{q_1,\ldots,q_m\}$ vers cet ensemble d'états, d'où le résultat.

Montrons réciproquement que, si w est accepté par $\mathcal{A}_2$ à partir d'un ensemble S d'états, alors, pour tout $q \in S$, il existe un calcul réussi de $\mathcal{A}_1$ sur w à partir de q. À nouveau, on prouve ce résultat par récurrence sur la longueur de w. Si $|w| = 0$, alors S est un état final de $\mathcal{A}_2$ et ne contient donc que des états finaux de $\mathcal{A}_1$: pour chacun d'eux il existe un calcul réussi de $\mathcal{A}_1$ sur w. Si maintenant $w = a \cdot w'$, w étant accepté par $\mathcal{A}_2$ à partir de S, il existe une transition $S' \in \delta_2(S, a)$ telle que w' est accepté par $\mathcal{A}_2$ à partir de S', soit, par hypothèse de récurrence, w' accepté par $\mathcal{A}_1$ à partir de chacun des états de S'. Si $q \in S$, ou bien $\tau(q, a) = \vee$ et $\delta(q, a) \cap S' \neq \emptyset$: soit $q' \in S' \cap \delta(q, a)$. On construit un calcul réussi sur w à partir de q en étiquetant la racine par q, celle-ci ayant pour seul fils un calcul réussi à partir de q' sur w'. Si maintenant $\tau(q, a) = \wedge$, alors, par définition, $\delta(q, a) \subseteq S'$. On construit alors un calcul réussi sur w en étiquetant la racine par q, les fils de la racine étant des calculs réussis sur $q_1, \ldots q_m$ où $\delta(q, a) = \{q_1, \ldots, q_m\}$.

Question 13

1. Raisonnons par l'absurde et supposons que ce langage soit reconnu par un automate ayant au plus 2^n états. Alors 1^{2^n}, de longueur 2^n peut s'écrire $w_1 \cdot w_2 \cdot w_2$ avec $w_2 \neq \epsilon$ et $w_1 \cdot w_2^k \cdot w_3$ dans le langage, d'après le lemme de pompage. Or c'est absurde puisque le langage ne contient qu'un mot.
2. On supposera dans la suite sans perte de généralité que $n \geq 2$.
   $Q = \{q_i \mid i \in \{1, \ldots, n\}\} \cup \{q_{i.k} \mid 1 \leq i \leq k \leq n\} \cup \{\overline{q_i}\ i \in \{1, \ldots, n\}\} \cup \{\overline{q_{i.k}} \mid 1 \leq i \leq k \leq n\} \cup \{q_0\}$, l'état initial est q_0, les états finaux sont les $\overline{q_{i,i}}, i < n$ et $q_{n,n}$. δ donné par

$$\begin{array}{rcll}
q_0 & \to & \overline{q_1} \wedge \ldots \wedge \overline{q_n} & \\
q_k & \to & q_{1,k} \vee \ldots \vee q_{k-1,k} \vee \overline{q_{k,k}} & \text{Si } 0 < k < n \\
q_{i,k} & \to & q_i \wedge q_k & \text{Pour } i < k < n \\
q_{k,k} & \to & \overline{q_1} \wedge \ldots \wedge \overline{q_{k-1}} \wedge q_k & \text{Pour } k < n \\
\overline{q_k} & \to & \overline{q_{1,k}} \vee \ldots \vee \overline{q_{k-1,k}} \vee q_{k,k} & \text{Pour } k \leq n \\
\overline{q_{i,k}} & \to & q_i \wedge \overline{q_k} & \text{Pour } i < k \leq n \\
\overline{q_{k,k}} & \to & \overline{q_1} \wedge \ldots \wedge \overline{q_{k-1}} \wedge \overline{q_k} & \text{Pour } k < n
\end{array}$$

   On montre, par récurrence sur $1 \leq i \leq 2^{n-1}$ qu'il existe un calcul de l'automate sur 1^{2i-1} tel que l'ensemble des états étiquetant des noeuds de profondeur $2i - 1$ est

$$\{q_j \mid \text{le jème bit de } i-1 \text{ est } 1\} \cup \{\overline{q_j} \mid \text{le jème bit de } i-1 \text{ est } 0\}$$

   Si $i = 1$, $2i - 1 = 1$ et on a un calcul de l'automate sur le mot 1 dont les noeuds de profondeur 1 sont étiquetés $\overline{q_1}, \ldots, \overline{q_n}$. Or tous les bits de $i - 1$ sont nuls.

   Si la propriété est vraie pour i. Soit j l'indice du bit nul de $i-1$ de poids le plus faible. Comme $i - 1 \leq 2^{n-1} - 1$, $j \leq n - 1$. Pour chaque noeud étiqueté q_k, $k < j$ on construit un successeur $\overline{q_{k,k}}$ qui a lui-même pour successeurs $\overline{q_1}, \ldots, \overline{q_k}$. Le noeud étiqueté $\overline{q_j}$ a pour successeur $q_{j,j}$ qui a lui-même pour successeurs $q_j, \overline{q_1}, \ldots, \overline{q_{j-1}}$. Enfin, pour $k > j$, les noeuds étiquetés q_k ont pour successeur $q_{j,k}$, qui ont eux-mêmes pour successeurs q_j et q_k et les noeuds étiquetés $\overline{q_k}$ ont pour successeur $\overline{q_{j,k}}$, qui a lui-même pour successeurs q_j et $\overline{q_k}$.

   Si on applique le résultat pour $i = 2^{n-1}$: le calcul de l'automate sur 1^{2^n-1} a pour feuilles les états $q_1, \ldots, q_{n-1}, \overline{q_n}$. En une étape, on obtient pour feuilles $\overline{q_{1,1}}, \ldots, \overline{q_{n-1,n-1}}, q_{n,n}$, d'où un calcul réussi sur 1^{2^n}.

Réciproquement, montrons par récurrence sur la profondeur d'un calcul réussi, que les mots acceptés à partir d'un état q_i $(i \geq 1)$ sont de la forme 1^{2m+1} où le $i$ème bit de m est 0 et ceux qui sont acceptés à partir d'un état $\overline{q_i}$, $i < n$ sont de la forme 1^{2m+1} où le $i$ème bit de m est 1. Enfin, ceux qui sont acceptés à partir de $\overline{q_n}$ sont de la forme 1^{2m+1} où $m < 2^{n-1}$.

Pour un calcul de profondeur 1, dont la racine est étiquetée q_i, il suffit que $i < n$ et en effet le $i$ème bit de $m = 0$ est 0. À partir de $\overline{q_i}$ le seul cas de calcul réussi correspond à $i = n$.

Si maintenant $k < n$ et ρ est un calcul réussi à partir de q_k, deux cas se présentent. Ou bien q_k a pour seul fils $q_{j,k}$, $j < k$ qui a lui-même deux fils q_j et q_k. Dans ce cas, par hypothèse de récurrence, le mot accepté est de la forme $1^2 \cdot 1^{2m+1}$ où le $j$ième bit de m est 0 et le $k$ième bit de m est 0 et $m < 2^{n-1}$. Dans ce cas, $m+1$ a encore 0 pour $k$ième bit et la récurrence est montrée. Ou bien q_k a pour fils $\overline{q_{k,k}}$ qui a lui même pour fils $\overline{q_1}, \ldots, \overline{q_k}$. Par hypothèse de récurrence, le mot accepté est de la forme $1^2 \cdot 1^{2m+1}$ tel que les k bits de poids faible de m valent 1. Dans ce cas, le $k$ième bit de $m+1$ vaut à nouveau 0.

Si $k < n$ et ρ est un calcul réussi à partir de $\overline{q_k}$, deux cas se présentent aussi. Ou bien $\overline{q_k}$ a pour seul fils $q_{k,k}$ qui, si $k < n$, a lui-même pour descendants $q_k, \overline{q_1}, \ldots, \overline{q_{k-1}}$. Dans ce cas, le mot accepté est de la forme 1^{2m+3} où les $k-1$ bits de poids faible de m valent 1 el le $k$ième bit vaut 0. Dans ce cas, le $k$ième bit de $m+1$ est bien 1.

Sinon, $\overline{q_k}$ a pour successeur $\overline{q_{j,k}}$, $j < k$, qui a lui-même pour successeurs $q_j, \overline{q_k}$. Par hypothèse de récurrence, le mot accepté est de la forme 1^{2m+3} et le $j$ème bit de m est 0, le $k$ième bit de m est 1. Il en résulte que le $k$ième bit de $m+1$ est 1. Ce qui achève la récurrence.

Finalement, si ρ est un calcul réussi à partir de $\overline{q_n}$, alors ou bien $\overline{q_n}$ a pour fils $q_{n,n}$ qui est une feuille du calcul et le mot accepté est 1. Ou bien $\overline{q_n}$ a pour fils $\overline{q_{j,n}}$ $(j < n)$ qui a lui-même pour fils $q_j, \overline{q_n}$. Dans ce cas, par hypothèse de récurrence, le mot accepté est de la forme 1^{2m+3} où $m < 2^{n-1}$ et le $j$ième bit de m est 0. Il en résulte que le $n$ième bit de $m+1$ est le même que celui de m : 0. Comme $m+1 \leq 2^{n-1}$, il en résulte que $m+1 < 2^{n-1}$.

D'après ce résultat, les mots acceptés à partir de tous les états $\overline{q_i}$, $i \leq n$ sont de la forme 1^{2m+1} où les $n-1$ bits de poids faible valent 1 et $m < 2^{n-1}$. Un seul mot satisfait cette spécification : 1^{2^n-1}. Il en résulte qu'un mot au plus est accepté à partir de q_0 : 1^{2^n}.

Ceci achève la démonstration que l'automate alternant construit accepte le langage constitué de l'unique mot $\{1^{2^n}\}$. Il est de taille $O(n^2)$ puisqu'il comporte $O(n^2)$ états et

$$\sum_{q\in Q} |\delta(q)| = \sum_{i=0}^{n-1} |\delta(q_i)| + \sum_{i=1}^{n} |\delta(\overline{q_i})| + \sum_{i=1}^{n-2} \sum_{k=i+1}^{n-1} |\delta(q_{i,k})| + \sum_{k=1}^{n} |\delta(q_{k,k})| + \sum_{i=1}^{n-1} \sum_{k=i+1}^{n} |\delta(\overline{q_{i,k}})| + \sum_{k=1}^{n-1} |\delta(\overline{q_{k,k}})|$$

Soit :

$$\sum_{q\in Q} |\delta(q)| = n + \sum_{i=1}^{n-1} i + \sum_{i=1}^{n} i + \sum_{i=1}^{n-2} 2(n-i+1) + \sum_{k=1}^{n} k + \sum_{i=1}^{n-1} 2(n-i) + \sum_{k=1}^{n-1} k$$

Soit une taille en $O(n^2)$.

Note : on peut en fait généraliser ce résultat. Si $w \in A^*$, on note $\widetilde{w}$ son image miroir : $\widetilde{\epsilon} = \epsilon$ et, pour tout $a \in A$, $\widetilde{a \cdot w} = \widetilde{w} \cdot a$. Si $L \subseteq A^*$, on note $\widetilde{L}$ l'ensemble des mots $\widetilde{w}$ tels que $w \in L$. L est reconnu par un automate non-déterministe à n états si et seulement si $\widetilde{L}$ est reconnu par un automate non-déterministe à n états. (facile à montrer). On a alors le résultat de compression logarithmique : si $\widetilde{L}$ est reconnu par un automate déterministe à n états, alors L est reconnu par un automate alternant à $O(\log^2 n)$ états.