Ordonnabilité des espaces métriques - X ENS C 2023

Pour toutes les remarques ou corrections, vous pouvez m'envoyer un mail à `galatee.hemery@gmail.com`

Partie I. Préliminaires

Question 1.1. Soit $\mathcal{M} = (X, \delta)$ un espace métrique tel que X fini.
On peut poser $d_{\min} = \min_{(x_1,x_2)\in X\times X} \delta(x_1, x_2)$ car l'ensemble est fini.
Ainsi, n'importe quelle suite finie $x_1, ..., x_{|X|}$ faisant apparaître exactement une fois chaque élément de X est un $d_{\min}$-ordre.
Donc $\mathcal{M}$ est $d_{\min}$-ordonnable donc ordonnable.

Question 1.2. Soit w_1, w_2 deux mots.
Si $w_1 = \varepsilon$, alors $\delta_{ppr}(w_1, w_2) = |w_2|$ (on ajoute les lettres une à une).
De même si $w_2 = \varepsilon$, alors $\delta_{ppr}(w_1, w_2) = |w_1|$ (on enlève les lettres une à une).
Sinon, on peut écrire $w_1 = \alpha_1 w_1'$ et $w_2 = \alpha_2 w_2'$ et $\delta_{ppr}(w_1, w_2) = \begin{cases} \delta_{ppr}(w_1', w_2') & \text{si } \alpha_1 = \alpha_2 \\ |w_1| + |w_2| & \text{sinon} \end{cases}$
En effet, dans le premier cas, la première lettre est conservée et il suffit de changer la fin du mot. Dans le second cas, on est obligé de "vider" w_1 pour changer la première lettre et obtenir w_2.

```
let rec delta_ppr l1 l2 = match (l1,l2) with
    |[],[] -> 0
    |[],t::q -> 1 + delta_ppr [] q
    |t::q,[] -> 1 + delta_ppr q []
    |a1::q1,a2::q2 when a1 = a2 -> delta_ppr q1 q2
    |a1::q1,a2::q2 -> 2 + delta_ppr q1 [] + delta_ppr [] q2 ;;
```

Question 1.3. On remarque que $(i+1)^2 = i^2 + 2i + 1$ et donc pour passer de w_i à w_{i+1}, on doit ajouter $2i + 1$ fois la lettre a.

```
int main(void)
{
    int i = 1 ;
    pushR('a') ; output() ;
    while (true) {
        i = i + 1 ;
        for (int j = 0 ; j< 2*i + 1 ; j++) {
            pushR('a') ;
        }
        output() ;
    }
}
```

Remarque : Un `return 0;` en fin de `main` serait plus naturel, mais je choisis de respecter le format proposé dans le sujet.

Question 1.4. Σ^* est bien un ensemble dénombrable. Il suffit de vérifier que δ_{pp} et δ_{ppr} sont des distances.

On note d'abord qu'elles sont bien définies de $\Sigma^* \times \Sigma^*$ dans $\mathbb{N}$.
Symétrie : Soit w_1, w_2.
Si on peut passer de w_1 à w_2 en k opérations, alors on peut passer de w_2 à w_1 en k opérations.
On considère $o_1, ..., o_k$ la suite d'opérations permettant de passer de w_1 à w_2 et $a_1, ..., a_k$ les lettres ajoutées ou enlevées à chaque étape.
Pour passer de w_2 à w_1, on applique la suite d'opérations $o'_k, ..., o'_1$ où o'_i est :

- `pushR(ai)` si o_i est `popR()` ;
- `pushL(ai)` si o_i est `popL()` ;
- `popR()` si o_i est `pushR(ai)` ;
- `popL()` si o_i est `pushL(ai)`.

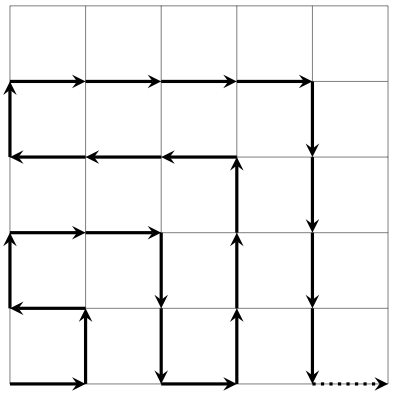
Ainsi, on a bien $\delta_{pp}(w_1, w_2) = \delta_{pp}(w_2, w_1)$ et comme la construction respecte le caractère push-pop-droite $\delta_{ppr}(w_1, w_2) = \delta_{ppr}(w_2, w_1)$.
Séparation : Soit w_1, w_2.
Si $\delta_{pp}(w_1, w_2) = 0$ ou $\delta_{ppr}(w_1, w_2) = 0$, cela signifie qu'aucune opération n'est nécessaire pour passer de w_1 à w_2 donc $w_1 = w_2$.
Inégalité triangulaire : Soit $(w_1, w_2, w_3) \in (\Sigma^*)^3$.
En effectuant les opérations permettant de passer de w_1 à w_2 puis de w_2 à w_3, on passe de w_1 à w_3.
Ainsi $\delta_{pp}(w_1, w_3) \leq \delta_{pp}(w_1, w_2) + \delta_{pp}(w_2, w_3)$ et $\delta_{ppr}(w_1, w_3) \leq \delta_{ppr}(w_1, w_2) + \delta_{ppr}(w_2, w_3)$.

Question 1.5. L'idée est de parcourir les éléments par longueur croissante et pour une même longueur parcourir les $a^i b^j$ par j croissant puis les $b^i a^j$ par j croissant.
$\varepsilon, a, b, aa, ab, bb, ba, aaa, aab, abb, bbb, ..., a^n, a^{n-1}b, ..., ab^{n-1}, b^n, b^{n-1}a, ..., ba^{n-1}, a^{n+1}, ...$

```
int main(void)
{
    int n = 0 ; // Longueur nulle
    output() ;
    while (true) {
        n = n + 1;
        // Longueur n, on part de a^(n-1)
        pushR('a') ; output () ;
        for (int i = 0 ; i < n ; i++) {
            popL() ; // élimination d'un a au début
            pushR('b') ; // ajout d'un b à la fin
            output() ;
        }
        for (int i = 0 ; i < n - 1; i++) {
            popL() ; // élimination d'un b au début
            pushR('a') ; // ajout d'un a à la fin
            output() ;
        }
        //dernières opérations pour revenir à a^(n-1), qu'on a déjà vu
        popL() ;
        pushR('a') ;
    }
}
```

On a ainsi $d = 3$, la plus grande suite d'opérations concerne le passage de ba^{n-1} à a^{n+1} où il faut enlever un b et ajouter deux a.

Question 1.6. On réalise un parcours correspondant au dessin suivant :



On part de a^{2p} avec p un entier naturel.
On ajoute un a au début, on a donc a^{2p+1}, puis on ajoute des b à la fin pour parcourir les $a^{2p+1}b^i$ pour $0 \leq i \leq 2p+1$.
Ensuite, on élimine les a un par un pour parcourir $a^i b^{2p+1}$ pour $0 \leq i \leq 2p+1$.
On ajoute ensuite un b à la fin, on a donc b^{2p+2}, puis on ajoute des a au début pour parcourir les $a^i b^{2p+2}$ pour $0 \leq i \leq 2p+2$.
Ensuite, on élimine les b un par un pour parcourir $a^{2p+2}b^i$ pour $0 \leq i \leq 2p+2$.
On aboutit à a^{2p+2} et on peut réitérer avec $p+1$ à la place de p.

```
int main(void)
{
    int p = 0 ;
    output() ; // mot vide a^(2*0)
    while (true) {
        pushL('a') ; output () ;
        for (int i = 0 ; i < 2*p +1 ; i++) {
            pushR('b') ; output () ;
        }
        for (int i = 0 ; i < 2*p +1 ; i++) {
            popL() ; output () ;
        }
        pushR('b') ; output () ;
        for (int i = 0 ; i < 2*p +2 ; i++) {
            pushL('a') ; output () ;
        }
        for (int i = 0 ; i < 2*p +2 ; i++) {
            popR() ; output () ;
        }
    }
}
```

Question 1.7.

a. On réalise la suite suivante $\varepsilon, a, ab, b, bb, abb, ..., b^{2p}, ab^{2p}, ab^{2p+1}, b^{2p+1}, b^{2p+2}, ...$

```
int main(void)
{
    output() ;
    while (true) {
        pushL('a') ; output () ;
        pushR('b') ; output () ;
        popL() ; output ()
        pushR('b') ; output () ;
    }
}
```

b. On remarque que $\delta_{ppr}(ab^i, b^j) = i + j + 1$ pour tout entiers naturels i, j.
Supposons par l'absurde que l'on a un d-ordre s pour un certain entier naturel d pour L.
On note $s = w_1, w_2, ...$
Il existe un indice j_0 à partir duquel les ab^i pour $i \leq d$ n'apparaissent plus : $j_0 = \max_{0 \leq i \leq d}(j | w_j = ab^i) + 1$.
Si w_{j_0} est de la forme ab^{i_0} alors $i_0 > d$ et les éléments de L à distance au plus d de w_{j_0} sont tous de la forme $ab^{i'}$ en vertu de la remarque faite en début de question.
Ainsi, w_{j_0+1} est de la forme ab^{i_1} avec $i_1 > d$ nécessairement.
En itérant le raisonnement, on obtient que la suite ne contient pas d'éléments de la forme b^i à partir du rang j_0.
Or il existe une infinité de b^i dans L et seulement un nombre fini peut apparaître avant le rang j_0. CONTRADICTION
Le même raisonnement s'applique si w_{j_0} est de la forme b^{i_0}.
Ainsi on peut conclure que $\mathcal{M}_{ppr}[L]$ n'est pas ordonnable.

Question 1.8. Soit t un entier naturel non nul. On cherche à montrer que $\mathcal{P}_t$ est NP-complet, c'est-à-dire NP-difficile et de classe NP.

Le problème est bien de classe NP. La donnée d'une suite finie s est un certificat. On vérifie bien en temps polynomiale que la suite s est un t-ordre, c'est-à-dire que chaque mot de L apparaît exactement une fois (pour cela on peut utiliser un dictionnaire), et que la distance entre deux éléments consécutifs est bien inférieure ou égale à t.

Pour montrer que le problème est NP-difficile, on utilise une réduction de **HamGrille** vers $\mathcal{P}_t$.
Soit $G = (V, E_V)$ un graphe grille.
On pose $L_G = \{a^{it}b^{jt}, (i, j) \in V\}$ fini car V fini et $|L_G| = |V|$.
On a $\delta_{pp}(a^{it}b^{jt}, a^{i't}b^{j't}) = (|i - i'| + |j - j'|) \times t$ pour tout $(i, j), (i', j')$ dans V.
Ainsi pour $(i, j), (i', j')$ dans V on a :
$\delta_{pp}(a^{it}b^{jt}, a^{i't}b^{j't}) \leq t$ si et seulement si $\{(i, j), (i', j')\} \in E_V$.
Supposons que $\mathcal{M}_{pp}[L_G]$ est t-ordonnable, alors il existe un t-ordre $s = w_1, ..., w_{|V|}$ qui correspond à une suite finie de sommets $(i_1, j_1), ..., (i_{|V|}, j_{|V|})$ deux à deux distincts et tel que deux sommets consécutifs sont reliés.
Ainsi, il s'agit d'un chemin hamiltonien dans G.
S'il existe un chemin hamiltonien, on peut lui faire correspondre un t-ordre de la même façon.
Ainsi $\mathcal{M}_{pp}[L_G]$ est t-ordonnable si et seulement si G admet un chemin hamiltonien.
La construction de L_G se fait en temps linéaire en la taille de V donc en temps polynomiale

en la taille de G.
Ainsi **HamGrille** $\leq_P \mathcal{P}_t$ ("se réduit polynomialement à").
HamGrille est NP-complet donc NP-difficile donc $\mathcal{P}_t$ est NP-difficile.

Partie II. Implémentation en C des programmes push-pop

Question 2.1. `liste lg = {.premier = NULL, .dernier = NULL};`

Question 2.2.

```
bool est_vide(void) {
    return (lg.premier == NULL) && (lg.dernier == NULL) ;
}
```

Question 2.3. Ces deux fonctions sont de complexité en $O(1)$ en temps. Le nombre d'opérations est borné et on ne fait aucun appel récursif, aucune boucle.

```
void pushL(int alpha) {
    chainon *nouveau = malloc(sizeof(chainon));
    assert(nouveau != NULL);
    nouveau->val = alpha;
    nouveau->prec = NULL;
    nouveau->suiv = lg.premier;
    if (est_vide()) {
        lg.premier = nouveau ;
        lg.dernier = nouveau ;
    }
    else {
        lg.premier->prec = nouveau ;
        lg.premier = nouveau ;
    }
}
void pushR(int alpha) {
    chainon *nouveau = malloc(sizeof(chainon));
    assert(nouveau != NULL);
    nouveau->val = alpha;
    nouveau->suiv = NULL;
    nouveau->prec = lg.dernier;
    if (est_vide()) {
        lg.premier = nouveau ;
        lg.dernier = nouveau ;
    }
    else {
        lg.dernier->suiv = nouveau ;
        lg.dernier = nouveau ;
    }
}
```

Question 2.4. Ces deux fonctions sont de complexité en $O(1)$ en temps. Le nombre d'opérations est borné et on ne fait aucun appel récursif, aucune boucle.

```
int popL(void) {
    assert( !(est_vide())) ;
    int valeur = lg.premier->val ;
    chainon *a_liberer = lg.premier ;
    lg.premier = lg.premier->suiv;
    free(a_liberer);
    if (lg.premier == NULL) {
        lg.dernier = NULL ;
    }
    else {
        lg.premier->prec = NULL;
    }
    return valeur;
}

int popR(void) {
    assert( !(est_vide())) ;
    int valeur = lg.dernier->val ;
    chainon *a_liberer = lg.dernier ;
    lg.dernier = lg.dernier->prec;
    free(a_liberer);
    if (lg.dernier == NULL) {
        lg.premier = NULL ;
    }
    else {
        lg.dernier->suiv = NULL;
    }
    return valeur;
}
```

Question 2.5. La complexité en temps des fonctions est linéaire en la taille de la liste : on parcours l'ensemble des maillons.

```
void output() {
    chainon *curseur = lg.premier ;
    while (curseur != NULL) {
        printf("%i ",curseur->val);
        curseur = curseur->suiv ;
    }
    printf("\n");
}
void vide_liste(void) {
    while (!(est_vide())) {
        popL();
    }
}
```

Partie III. Ordonnabilité des graphes

Question 3.1. V est un ensemble fini, il suffit de montrer que δ_G est une distance.
On note que δ_G est bien définie sur $V \times V$ et à valeurs dans $\mathbb{N}$ car le graphe est connexe et un chemin existe toujours entre deux sommets.
Symétrie. Le graphe étant non orienté, on a bien $\delta_G(u, v) = \delta_G(v, u)$ pour tout u, v dans V car tout chemin peut être empreinté dans les deux sens.
Séparation. Soit u, v dans V tel que $\delta_G(u, v) = 0$ alors il existe un chemin de longueur nulle reliant u et v donc $u = v$.
Inégalité triangulaire. Soit u, v, w dans V.
En concaténant un plus court chemin de u à v et un plus court chemin de v à w on obtient un chemin de u à w de longueur $\delta_G(u, v) + \delta_G(v, w)$ qui n'est pas nécessairement le plus court.
Ainsi, $\delta_G(u, w) \leq \delta_G(u, v) + \delta_G(v, w)$.

Question 3.2.

a. $6, 1, 0, 2, 7, 3, 8, 4, 9, 5, 10$

b. $5, 1, 2, 6, 0, 7, 3, 4, 8$

Question 3.3. Il suffit de réaliser un parcours du graphe, en profondeur par exemple en utilisant une structure de pile en partant du sommet 0 et en créant une arête vers un sommet la première fois qu'on l'ajoute à la pile issue du sommet qui a permis de le rencontrer.
On construit bien un arbre car on ajoute exactement une arête vers chaque sommet et comme 0 est visité en premier, on ajoute aucune arête vers 0 et chaque sommet est découvert depuis un sommet déjà relié à un arbre donc l'ajout d'une arête conserve la structure d'arbre.
A chaque étape, l'abre construit et un arbre couvrant de l'ensemble des sommets rencontrés (invariant de boucle).
Le graphe étant connexe, on rencontre tous les sommets et le graphe construit est bien un arbre couvrant.
On utilise la variable globale `lg` comme pile.

```
void calcule_arbre_couvrant (bool graphe[N][N], bool arbre[N][N]) {
    vide_liste();
    pushR(0);
    bool vu[N];
    for (int i = 0; i<N; i++) {
        vu[i] = false;
        for (int j =0;j<N;j++) {
            arbre[i][j] = false;
        }
    }
    while (!(est_vide())) {
        int u = popR() ;
        for (int v = 0; v<N; v++) {
            if ((!vu[v])&&(graphe[u][v])) {
                arbre[u][v] = true ;
                vu[v] = true ;
                pushR(v) ;
            }
```

```
        }
    }
}
```

Question 3.4. *a.* Lors d'un appel récursif, `mystere` est égal à `true` si le noeud n est à une profondeur impaire et `false` sinon.
En effet, on change le booléen pour chaque appel à un fils donc pour chaque changement de parité dans la profondeur.

b. $0, 5, 1, 6, 2, 7, 3, 8, 4$

Question 3.5. La fonction `visite` réalise un parcours de l'arbre donc chaque noeud est bien ajouté à `lg` exactement une fois.
Il suffit de montrer que c'est bien une 3-suite qui est construite et donc que le noeuds inséré après un noeud n est toujours à distance 3 au plus.
0 est toujours inséré en premier.
Après 0, on appelle `visite` sur un premier fils avec `mystere = true` qui va réaliser des appels à ses fils avec `mystere = false`, qui ajouteront en premier le noeud s'il en a et s'ajoutera lui-même s'il n'en a pas. Donc le second élément est un élément de profondeur 1 ou 2, inférieur à 3.
Premier cas : Soit n un noeud de profondeur paire, alors l'appel est fait avec `mystere = false` et n est ajouté en début de fonction, ensuite, des appels sont réalisés à `visite` avec ses fils et `mystere = true`, le premier sera noté f_1 s'il existe.
Donc de même que pour 0. des appels aux fils de f_1 avec `mystere = false` seront faits, qui ajouteront en premier le noeud si f_1 a des fils et s'ajoutera lui-même s'il n'en a pas. Donc l'élément suivant est un élément plus profond de 1 ou de 2 donc à distance inférieur à 3.
Si n n'a pas de fils, alors soit c'est le dernier élément, soit le prochain appel fait sera sur un de ses frères directs à distance 2, soit son père de profondeur impaire aura fini ses appels à ses fils et s'ajoutera à `lg`.
Deuxième cas : Soit n un noeud de profondeur impaire, alors l'appel concernant n est fait avec `mystere = true` et n est ajouté à `lg` lorsque tout l'arbre issue de n est ajouté.
L'appel concernant le père p de n a réalisé l'appel concernant n.
Soit n n'est pas le dernier fils, alors après avoir terminé l'appel concernant n, on réalise un appel concernant un frère f de n avec `mystere = true`. Si f n'a pas de fils, il s'ajoute et il est bien a distance 2, sinon, on ajoutera en premier un de ses fils de profondeur paire, qui est à distance 3.
Soit n est le dernier fils de p, dans ce cas, on termine l'appel concernant p. Considérons l'appel concernant g le père de p, de profondeur impaire. Soit p est le dernier fils de g, alors g est ajouté à `lg` et est à distance 2 de n. Sinon, un appel à un oncle o de n de profondeur paire est fait et on ajoute o à `lg` à distance 3 de n.

Cette disjonction de cas un peu longue permet de conclure...

Question 3.6. Pour chaque noeud, on teste si tous les sommets sont des fils, donc la complexité est en $O(N^2)$ car on a N appels à `visite` réalisant ces N tests.
En utilisant une structure récursive permettant d'avoir directement accès aux fils sans test, on pourrait avoir une complexité en $O(N)$.

Question 3.7. *a.* Si l'arbre est grand, on pourrait avoir un dépassement de la taille de la pile d'appel.

b. Pour remédier à cela, on peut utiliser une pile et une boucle `while`.
La pile contient des couples $(n, etat)$ où $etat$ est un entier qui vaut $0, 1$ ou 2 et n un noeud.
On utilise 0 pour les noeuds de profondeur paire et 1 et 2 pour les sommets de profondeur impaire.
A chaque étape, on dépile le terme t un haut de la pile et ce tant que la pile n'est pas vide :

— si $t = (n, 0)$, alors on ajoute n à `lg` et on ajoute $(f, 1)$ pour tout fils f de n dans l'ordre décroissant ;

— si $t = (n, 1)$, alors on ajoute $(n, 2)$ à la pile puis $(f, 0)$ pour tout fils f de n dans l'ordre décroissant ;

— si $t = (n, 2)$, on a finit de traiter les fils de n de profondeur impaire donc on ajoute n à `lg`.

Partie IV. Ordonnabilités des langages réguliers pour la distance push-pop-droite

Question 4.1. On réalise un parcours de l'automate et on compte le nombre de retour à un sommet déjà visité.
Pour cela, on peut raisonner récursivement et conserver dans un tableau les sommets rencontrés ou non.

```
nmb_retour(vu,A,etat) :

    ------------------------------
    Entrée : vu un tableau de booleen indiquant quels états ont été visités
             A un automate
             etat un état de A
    Sortie : nombre de retour à un état déjà visité

    ------------------------------
    vu[etat] = true
    resultat = 0
    POUR TOUT a dans Sigma tel que trans(etat,a) bien défini :
        si vu[trans(etat,a)] :
            resultat += 1
        sinon :
            resultat += nmb_retour(vu,A,trans(etat,a))
    vu[etat] = false
    RENVOYER resultat
```

Ensuite, il s'agit simplement de vérifier que le résultat et inférieur ou égale à 1.

Question 4.2. Soit A un automate poêle. On note N le nombre d'état de l'automate. Ainsi, A a un unique chemin strict vers un cycle depuis q_{init}, on note u son étiquette et un unique cycle, on note v l'étiquette du cycle considéré à partir de l'état d'entrée.
Soit $w \in L(A)$ tel que $|w| \geq N$, alors le chemin depuis q_{init} d'étiquette w aboutit à un état final et existe. Ce chemin visite $N + 1$ états au moins et par le principe des tiroirs, il visite deux fois le même état.
Ainsi, on met en évidence l'utilisation du cycle.
Ainsi, le début du chemin est nécessairement le chemin strict vers un cycle, suivi d'un

certain nombre d'utilisation de l'unique cycle, puis une fin à partir de l'état d'entrée utilisant au plus une fois chaque état.
On a donc un chemin de la forme $q_{init} \xrightarrow{u} q_{entree} \xrightarrow{v} q_{entree} \dots \xrightarrow{v} q_{entree} \xrightarrow{w_f} q_f$ avec $|w_f| \leq N-1$.
Ainsi, on obtient $w = uv^k w_f$ avec $k \in \mathbb{N}$.
On note F' l'ensemble des mots w_f accepté par A depuis q_{entree} sans utiliser le cycle.
On note F l'ensemble des mots accepté par A depuis q_{init} sans utiliser le cycle.
Ces deux ensembles sont bien fini car la longueur des mots est majorée par $N-1$.
Ainsi, on a bien $L(A) = F \cup (\{u\} \cdot \{v\}^* \cdot F')$, qui correspond à ce qui est attendu.

Question 4.3. On calcul un d ordre avec $d \leq 3N$ en considérant le N de la question précédente.

```
let rec ajoute (l : lettre list) =
    (* ajoute un mot à la fin de lg *)
    match l with
        |[] -> ()
        |t::q -> pushR t ; ajoute q ;;

let rec elimine (l : lettre list) =
    (* elimine un mot à la fin de lg *)
    match l with
        |[] -> ()
        |t::q -> popR () ; elimine q ;;

let rec parcours (f : (lettre list) list) =
    (* parcours une liste de mot, les ajoute à lg,
    affiche le résultat puis les supprime *)
    match f with
        |[] -> ()
        |t::q -> ajoute t ; output () ;
                 elimine t ; parcours q ;;

let programme_ppr (u:lettre list) (v:lettre list)
                  (f: (lettre list) list) (f' : (lettre list) list)) =
    parcours f ;
    ajoute u ;
    while (true) do
        parcours f' ; ajoute v ;
    done ;;
```

Question 4.4. Soit L un langage infini. On peut construire une suite $w_1, w_2, ...$ en raisonnant par dichotomie pour qu'elle corresponde à un branche lourde de L.
$T_\varepsilon = T_\Sigma$ contient tous les mots de L et donc une infinité de mots de L.
On peut ainsi poser $w_1 = \varepsilon$.
Supposons que l'on a construit la suite jusqu'à un rang i. Alors T_{w_i} contient une infinité de mots de L.
On considère les deux sous-arbres fils de T_{w_i}, correspondant à $T_{w_i a}$ et $T_{w_i b}$. Nécessairement, l'un des deux contient une infinité de mots de L, ainsi, on peut poser $w_{i+1} = w_i a$ ou $w_i b$ selon.

Question 4.5. Soit L un langage. Supposons par l'absurde que L a au moins deux branches lourdes $w_1, w_2, ...$ et $w'_1, w'_2, ...$ et que $\mathcal{M}_{ppr}[L]$ est d-ordonnable pour un d entier naturel non nul et on considère $s = m_1, m_2, ...$ un d-ordre.
On note i_0 l'indice à partir duquel $w_i \neq w'_i$ qui existe sinon les branches sont identiques.
On sait que $T_{w_{i_0+d}}$ et $T_{w'_{i_0+d}}$ contiennent chacun une infinité disjointe d'éléments de L et qu'il est nécessaire de repasser par w_{i_0-1} pour passer d'un éléments d'une branche à un élément de l'autre et ainsi de faire au moins $2d > d$ opérations push-pop-droite.
Le même raisonnement mené à la question 1.7.b s'adapte ici.
On a un nombre fini de mots dans L de longueur au plus $i_0 + d$, on note j_0 l'indice à partir duquel tous ces mots sont traités dans s le d ordre.
On note ensuite le premier indice j_1 d'un mot de $T_{w_{i_0+d}}$ qui existe bien. A partir de cet indice, le préfixe w_{i_0} ne peut plus être modifier car à chaque étape, on peut fait au plus d opérations et les mots sont de longueur au moins $i_0 + d + 1$.
En particulier, on ne peut plus atteindre l'infinité de mots restants de $T_{w'_{i_0+d}}$ de préfixe $w'_{i_0} \neq w_{i_0}$. CONTRADICTION.

Question 4.6. *a.* Supposons par l'absurde que u est un préfixe de u'.
L'automate étant déterministe, le chemin d'étiquette u est exactement le début du chemin d'étiquette u'.
On a donc $q_{init}, u_1, q_2, ..., u_{|u|}, q_{|u|+1}, ..., u'_{|u'|}, q_{|u'|+1}$ le chemin considéré.
Or ce sont des chemins stricts vers des cycles donc $q_{|u|+1}$ appartient à un cycle comme fin du chemin d'étiquette u et n'appartient par à un cycle comme état interne du chemin d'étiquette u'. CONTRADICTION
On obtient le second résultat en échangeant les rôles de u et u'.

b. T_u et $T_{u'}$ contiennent une infinité de mots e $L(A)$.
En effet, il existe au moins un mot w_0 reconnu depuis l'état q de la fin du chemin strict vers un cycle d'étiquette u car l'automate est émondé donc q est co-accessible.
On note v l'étiquette du cycle en partant de q, on a alors $uv^k w_0 \in L(A)$ pour tout k entier naturel.
En suivant les lettres du cycle, on peut repérer une branche lourde $w_1, w_2, ..., u, ...$ pour $L(A)$.
Le raisonnement est le même concernant u' et on repère une branche lourde $w'_1, w'_2, ..., u', ...$ pour $L(A)$ distinct d'après la question a.
$L(A)$ a donc effectivement au moins deux branches lourdes.

Question 4.7. Soit A un automate émondé qui a un seul chemin strict vers un cycle. On suppose qu'il n'est pas pseudo-acyclique. On a alors l'existence de deux cycles d'étiquettes v et v' en partant de l'unique état d'entrée.
On peut supposer que les cycle n'ont aucune répétitions du premier état avant le dernier quitte à ne considérer que le début ou que la fin.
On note u l'étiquette du chemin strict vers un cycle.
On remarque que les deux cycles partagent au moins l'état d'entrée sinon l'un d'en eux ne serait pas accessible et l'automate ne serait pas émondé.
Supposons par l'absurde que v est un préfixe de v', alors il y a répétition que du premier état dans le cycle de v' avant la fin : c'est impossible. Le même raisonnement s'applique en échangeant les rôles.
On a donc T_{uv} et $T_{uv'}$ qui contiennent une infinité de termes de L chacun. Avec le même raisonnement que dans la question 4.6.b, on obtient l'existence de deux branches lourdes.

On a montré que si L est reconnu par un automate poêle, alors $\mathcal{M}_{ppr}[L]$ est ordonnable dans la question 4.3.
Si L est reconnu par un automate émondé A qui n'est pas poêle, alors :

— soit l'automate a au moins deux chemins stricts distinct vers des cycles et donc deux branches lourdes d'après 4.6. Ainsi $\mathcal{M}_{ppr}[L]$ n'est pas ordonnable d'après 4.5.

— soit l'automate n'est pas pseudo-acyclique, alors L a au moins deux branches lourdes d'après 4.6. Ainsi $\mathcal{M}_{ppr}[L]$ n'est pas ordonnable d'après 4.5.

Ainsi, on peut en déduire l'équivalence attendue.