Test d'égalité de langages rationnels

L'objet de ce problème est d'étudier comment on peut décider si deux expressions rationnelles $\mathcal{E}$ et $\mathcal{F}$ décrivent le même langage rationnel. Ce problème est intrinsèquement difficile : notamment si P $\neq$ NP alors il n'y a pas d'algorithme polynomial pour le résoudre. Dans ce sujet, après des préliminaires algorithmiques, nous allons considérer que les expressions rationnelles sont encodées comme des arbres, et étudier un algorithme pour calculer un automate non-déterministe qui reconnaît le même langage. Si on déterminise les automates qui correspondent à $\mathcal{E}$ et $\mathcal{F}$, on se ramène donc à tester si deux automates déterministes reconnaissent le même langage, ce qui fait l'objet de la dernière partie.

Attendus. Il est attendu des candidates et des candidats des réponses construites. Ils seront aussi évalués sur la précision, le soin et la clarté de la rédaction.

Partie I. Préliminaires algorithmiques

Dans tout le sujet, le mot *complexité* désigne la complexité temporelle.

1 Tableaux redimensionnables

Les *listes* Python telles qu'elles sont implantées en CPython (l'implantation de référence du langage Python) sont en réalité ce que l'on appelle des *tableaux redimensionnables*, une structure de données souple et efficace, qui permet de stocker un nombre non borné de données, et en plus d'accéder au i-ème élément en temps constant.

Pour notre usage dans ce problème, un *tableau redimensionnable* est une structure de données qui supporte les opérations suivantes (sauf mention contraire, on n'utilisera pas les autres) :

- **initialisation** : crée un tableau redimensionnable vide ne contenant aucune donnée (en Python, c'est l'instruction `t = []`) ;
- **longueur** : renvoie le nombre d'éléments stockés dans le tableau redimensionnable (en Python, c'est l'instruction `len(t)`) ;
- **accès** : permet d'accéder en lecture et en écriture à l'élément en i-ème position dans un tableau redimensionnable ; les indices commencent à 0, donc il faut que i soit compris entre 0 et la longueur moins 1 (en Python, c'est `t[i]`).
- **ajout à la fin** : ajoute une donnée après la dernière donnée existante du tableau redimensionnable (en Python, c'est l'instruction `t.append(x)`) ;

Pour notre analyse algorithmique, on considère les opérations suivantes sur les *tableaux* (pas les tableaux redimensionnables) et leurs complexités :

- allouer un tableau de taille n en temps $\mathcal{O}(n)$;
- accéder au i-ème élément d'un tableau en temps $\mathcal{O}(1)$;
- affecter une valeur au i-ème élément d'un tableau en temps $\mathcal{O}(1)$.

On souhaite implanter des tableaux redimensionnables en utilisant des *enregistrements* (les `struct` du langage C) qui contiennent trois champs : `tableau` qui est un tableau, `capacite` qui est la taille allouée au tableau, `longueur` qui est un entier indiquant combien d'éléments sont stockés dans la structure : une `capacite` de 8 indique que `tableau` est alloué pour contenir 8 éléments, et si la `longueur` vaut 5, cela signifie que seuls les 5 premiers éléments, d'indices 0 à 4 inclus, sont dans la liste qui est représentée, voir Fig. 1.

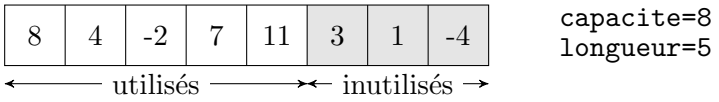


FIGURE 1 – Représentation de la liste abstraite `[8,4,-2,7,11]` au moyen d'un tableau redimensionnable de capacité 8.

Les algorithmes pour les tableaux redimensionnables sont les suivants :

- **initialisation** : crée un enregistrement avec `capacite=1`, `longueur=0` et alloue une case pour `tableau` ;
- **longueur** : renvoie `longueur` ;
- **accès** : renvoie `tableau[i]` ;
- **ajout à la fin** : il y a deux cas selon que l'on ait la place pour un nouvel élément ou non
  - si `longueur < capacite`, on place le nouvel élément dans la case d'indice `longueur` du champs `tableau` et on incrémente `longueur` ;
  - si `longueur == capacite`, on alloue un nouveau `tableau` de `capacite` doublée, on recopie tous les éléments de l'ancien tableau dedans, puis on place le nouvel élément dans la case d'indice `longueur` du champs `tableau` et on incrémente `longueur`.

Question 1. Représenter, à chaque étape, l'enregistrement encodant un tableau redimensionnable que l'on initialise, puis auquel on ajoute successivement 3, -2, 4, 1 et -2 (6 étapes en tout avec l'initialisation).

Question 2. Parmi les opérations considérées, et dans le modèle pour les tableaux décrit plus haut, quelle est la seule opération à ne pas s'effectuer en temps constant ? Dans quels cas précisément, en fonction de la valeur de `longueur`, l'opération ne se fait pas en temps constant ? Quelle est sa complexité dans ces cas ?

Question 3. On initialise puis effectue $n \geq 1$ insertions consécutives dans un tableau redimensionnable. On note I_n le nombre total d'écritures dans `tableau` qui ont été effectuées dans le processus (instructions du type `tableau[i] = ...`, en comptant celles effectuées lors des recopies). Soit k l'unique entier tel que $2^{k-1} < n \leq 2^k$. Montrer que $I_1 = 1$ et que pour tout $n \geq 2$ on a

$$I_n = n + \sum_{i=0}^{k-1} 2^i.$$

Question 4. En déduire qu'initialiser puis effectuer $n \geq 1$ insertions consécutives dans un tableau redimensionnable se fait en temps $\mathcal{O}(n)$.

2 Tri lexicographique

Dans cette section on s'intéresse au problème de trier pour l'ordre lexicographique n listes contenant chacune k entiers de $\{0, \ldots, c-1\}$, où n, k, c sont des entiers strictement positifs. Quand on est dans ce cadre, on peut trier plus efficacement qu'en utilisant un tri comme le tri fusion. Cet algorithme sera utilisé dans la section 6.

On considère que les listes sont implantées au moyen de tableaux redimensionnables, et que l'on peut ainsi ré-utiliser les complexités de la section précédente.

Par exemple, l'entrée de l'algorithme pour $n = 5$, $k = 3$ et $c = 7$ pourrait être, en notation PYTHON : `L = [[2,5,1], [6,0,0], [1,2,3], [1,0,3], [4,2,1]]`.

Question 5. Quel est le résultat attendu après avoir trié `L` ci-dessus selon l'ordre lexicographique ?

Question 6. Écrire une fonction PYTHON `plus_petit_k_uple` qui prend en argument deux listes d'entiers `l1` et `l2` supposées être de même taille et renvoie `True` si et seulement si `l1` arrive avant `l2` dans l'ordre lexicographique, et `False` sinon. Cette fonction doit posséder une complexité linéaire en la taille commune des listes fournies en argument (sans avoir à la justifier).

Question 7. Si on utilise la fonction `plus_petit_k_uple` pour comparer deux listes d'entiers de taille k, quelle serait la complexité en fonction de n et de k d'utiliser le Tri Fusion (MERGESORT) pour résoudre le problème de trier selon l'ordre lexicographique une liste de n listes d'entiers de taille k ?

La solution ci-dessus n'utilise pas le fait que les valeurs sont toutes comprises entre 0 et $c-1$. On va pouvoir proposer une solution algorithmiquement plus performante en exploitant cette spécificité.

On considère l'algorithme TRIPARCLE(L,j,c), où L est une liste de listes de notre problème et $j \in \{0, \ldots, k-1\}$. Cet algorithme renvoie une liste K contenant une permutation de L où les éléments sont triés selon leur indice j : pour tous i, i' tels que $0 \leq i < i' < n$, $K[i][j] \leq K[i'][j]$. On utilise pour cela le procédé suivant :

- Créer un tableau redimensionnable T contenant c tableaux redimensionnables initialement vides.
- Pour chaque élément (liste de longueur k) ℓ de L dans l'ordre, ajouter ℓ en fin du tableau redimensionnable $T[\ell[j]]$.
- Renvoyer la concaténation des tableaux redimensionnables $T[0]$, $T[1]$, …, $T[c-1]$.

Question 8. Que renvoie l'algorithme appliqué à `L = [[2,5,1], [6,0,0], [1,2,3], [1,0,3], [4,2,1]]` avec $j = 1$?

Question 9. Écrire l'algorithme TRIPARCLE(L,j,c) en PYTHON.

Tournez la page S.V.P.

Question 10. Montrer que l'algorithme TriParCle(L,j,c) a pour complexité $\mathcal{O}(c+n)$.

Question 11. Montrer que l'algorithme TriParCle(L,j,c) est stable : si on a deux indices i, i' avec $0 \leq i < i' < n$ tels que $L[i] \neq L[i']$ et que $L[i][j] = L[i'][j]$, alors dans le résultat K, l'élément $L[i]$ est avant $L[i']$.

L'algorithme TriLexicographique(L,c) consiste à trier successivement L avec TriParCle(L,j,c) en faisant décroître j de $k-1$ à 0.

Question 12. Appliquer l'algorithme TriLexicographique(L,7) à `L = [[2,5,1], [6,0,0], [1,2,3], [1,0,3], [4,2,1]]`, en indiquant la liste obtenue à chaque itération de la boucle sur j.

Question 13. Montrer que l'algorithme TriLexicographique(L) est une solution correcte au problème initial : il trie les éléments de L pour l'ordre lexicographique.

On a ainsi un algorithme de complexité $\mathcal{O}(k(c+n))$ pour résoudre le problème de trier pour l'ordre lexicographique n listes de k entiers compris entre 0 et $c-1$.

Important : Pour toute la suite, afin de simplifier les explications, on considérera que les listes de Python ont les opérations **initialisation**, **longueur**, **accès** et **ajout à la fin** en temps $\mathcal{O}(1)$, sans spécifier à chaque fois qu'il s'agit en fait de tableaux redimensionnables et que l'ajout à la fin se fait en réalité en temps $\mathcal{O}(1)$ amorti.

Partie II. Des expressions rationnelles aux automates

Dans cette partie, nous explorons l'algorithme de Glushkov dont le but est de trouver un automate (a priori non déterministe) qui reconnaît le même langage qu'une expression rationnelle donnée.

3 Représenter des expressions rationnelles non vides

Soit un alphabet fini Σ et ε le mot vide sur cet alphabet.

Les *expressions rationnelles non vides* sur Σ sont définies inductivement par :

— l'ensemble d'assertions $\{\varepsilon, a \mid a \in \Sigma\}$,

— l'ensemble de règles d'inférence {somme : $(E_1, E_2) \mapsto (E_1) + (E_2)$, étoile : $E \mapsto (E)^*$, produit : $(E_1, E_2) \mapsto (E_1) \cdot (E_2)$}.

On s'autorisera à ne pas mettre systématiquement des parenthèses, en prenant la convention de préséance suivante : l'étoile est prioritaire sur le produit qui est prioritaire sur la somme.

On s'autorisera également à ne pas écrire systématiquement le symbole $\cdot$ pour le produit. Ainsi, l'expression $(((a)\cdot(b))^*)+(((a)\cdot(a)+\varepsilon))$ pourra s'écrire $(ab)^*+aa+\varepsilon$.

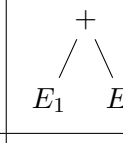
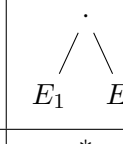
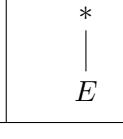
Par la suite on utilisera le terme *expression rationnelle* pour désigner une expression rationnelle non vide.

À chaque expression rationnelle E, on peut associer sa longueur $|E|$ et son langage $\mathcal{L}(E)$ sur Σ, définis de manière inductive :

expression	longueur	langage
ε	1	$\{\varepsilon\}$
$a \quad (a\in\Sigma)$	1	$\{a\}$
$(E_1)+(E_2)$	$1+\lvert E_1\rvert+\lvert E_2\rvert$	$\mathcal{L}(E_1)\cup\mathcal{L}(E_2)$
$(E_1)\cdot(E_2)$	$1+\lvert E_1\rvert+\lvert E_2\rvert$	$\mathcal{L}(E_1)\cdot\mathcal{L}(E_2)$
$(E)^*$	$1+\lvert E\rvert$	$\mathcal{L}(E)^*=\bigcup_{i=0}^{\infty}\mathcal{L}(E)^i$

avec par convention $L^0=\{\varepsilon\}$ et $L^i=L\cdot L^{i-1}$ pour tout langage L et tout entier $i\geq 1$. Sur les langages, on prend la règle de préséance suivante : l'étoile est prioritaire sur le produit qui est prioritaire sur l'union.

On peut représenter de telles expressions sous forme d'arbres syntaxiques (représentation naturelle à partir de la définition inductive) et construire à partir d'un tel arbre une liste de listes imbriquées pour les calculs en PYTHON :

expression	arbre	PYTHON
ε	ε	`[ '' ]`
$a \quad (a\in\Sigma)$	a	`[ 'a' ]`
$(E_1)+(E_2)$		`[ '+', E1, E2 ]`
$(E_1)\cdot(E_2)$		`[ '.', E1, E2 ]`
$(E)^*$		`[ '*', E ]`

Dans la suite, quand on parlera d'une expression rationnelle en PYTHON, elle sera nécessairement sous forme de liste de listes imbriquées, comme défini ci-dessus, sans qu'on ait besoin de le spécifier.

Question 14. Donner l'arbre syntaxique et la représentation PYTHON de l'expression rationnelle $(a+b)^*a+(ab+\varepsilon)(c+\varepsilon)$.

Tournez la page S.V.P.

Question 15. Écrire une fonction PYTHON `expression` qui prend en argument une expression rationnelle et renvoie une représentation sous forme de chaîne de caractères de cette expression.

Exemple.
`expression(['+', [ '*', [ '.', [ 'a'] , [ 'b'] ] ], ['+', [ '.', ['a'], ['a']], ['']]])`
s'évalue en `'(((a).(b))*)+(((a).(a))+(_))'`, où le caractère `'_'` représente le mot vide.

Question 16. Écrire une fonction PYTHON `contient_mot_vide` qui prend en argument une expression rationnelle et renvoie `True` si le mot vide appartient au langage associé à l'expression rationnelle, `False` sinon. Votre fonction doit avoir une complexité linéaire en la longueur de l'expression rationnelle (sans avoir à le justifier).

4 Automate de Glushkov

L'automate de Glushkov d'une expression rationnelle E est un automate particulier qui reconnaît le langage $\mathcal{L}(E)$ associé à cette expression. Le but de cette section est de construire cet automate de manière efficace.

4.1 Rappels et propriétés

La première étape pour construire l'automate de Glushkov associé à une expression rationnelle E sur l'alphabet Σ est de *linéariser* cette expression, c'est-à-dire construire une nouvelle expression E_ℓ sur un nouvel alphabet Σ_ℓ, dont on déduit facilement E et dans laquelle chaque lettre possède au plus une occurrence. Une telle expression est qualifiée de *locale*.

On note $\#E$ le nombre de lettres apparaissant dans l'expression E et on construit une nouvelle expression E_ℓ sur l'alphabet $\{a_i \mid a \in \Sigma,\ 1 \leq i \leq \#E\}$, en adjoignant à chaque lettre qui apparaît dans l'expression E sa position : on numérote les lettres de E de gauche à droite en partant de 1, et on ajoute ce numéro en indice à chaque lettre. L'ensemble des lettres effectivement utilisées dans E_ℓ est appelé *alphabet* de E_ℓ et noté Σ_ℓ.

Ainsi, à partir de l'expression $E = (ab)^* + aa + \varepsilon$ sur l'alphabet $\Sigma = \{a, b\}$, on obtient l'expression linéarisée $E_\ell = (a_1b_2)^* + a_3a_4 + \varepsilon$ sur l'alphabet $\Sigma_\ell = \{a_1, b_2, a_3, a_4\}$.

Question 17. Écrire une fonction `linearisation` qui prend en argument une expression rationnelle et renvoie l'expression obtenue en la linéarisant. Cette fonction doit avoir une complexité linéaire en la longueur de l'expression de départ (sans avoir à le justifier).

Exemple.
`linearisation(['+', ['*', ['.', ['a'] , ['b'] ] ], ['+', ['.', ['a'], ['a']], ['']]])`
s'évalue en `['+', ['*', ['.', ['a1'], ['b2']]], ['+', ['.', ['a3'], ['a4']], ['']]]`.

Pour construire l'automate de Glushkov de E à partir de E_ℓ, on définit les sous-ensembles et valeurs suivants :

— First(E) est l'ensemble des lettres de Σ_ℓ qui apparaissent au début d'un mot de $\mathcal{L}(E_\ell)$:

$$\text{First}(E) = \{x \in \Sigma_\ell \mid \exists u \in {\Sigma_\ell}^*,\, xu \in \mathcal{L}(E_\ell)\}\,,$$

— Last(E) est l'ensemble des lettres de Σ_ℓ qui apparaissent à la fin d'un mot de $\mathcal{L}(E_\ell)$:

$$\text{Last}(E) = \{x \in \Sigma_\ell \mid \exists u \in {\Sigma_\ell}^*,\, ux \in \mathcal{L}(E_\ell)\}\,,$$

— Null(E) = vrai si $\mathcal{L}(E_\ell)$ contient le mot vide, et faux sinon ;

— Follow(E) est l'ensemble des facteurs de longueur 2 des mots de $\mathcal{L}(E_\ell)$:

$$\text{Follow}(E) = \{xy \in {\Sigma_\ell}^2 \mid \exists u, v \in {\Sigma_\ell}^*,\, uxyv \in \mathcal{L}(E_\ell)\}\,.$$

Question 18. Donner les valeurs de First(E), Last(E), Null(E) et Follow(E) pour l'expression $E = (a+b)^*a + (ab+\varepsilon)(c+\varepsilon)$.

On associe à l'expression E l'automate $\mathcal{A}_\ell = (\Sigma_\ell, \Sigma_\ell \cup \{i\}, \delta_\ell, i, F)$ sur l'alphabet Σ_ℓ, ayant pour ensemble d'états $\Sigma_\ell \cup \{i\}$ (où $i \notin \Sigma_\ell$) et pour état initial i défini par :

— pour toute lettre $x \in \text{First}(E)$, on a la transition $\delta_\ell(i, x) = \{x\}$ de l'état i par la lettre x,

— pour toute lettre $x \in \Sigma_\ell$ et toute lettre $y \in \Sigma_\ell$ telles que $xy \in \text{Follow}(E)$, on a la transition $\delta_\ell(x, y) = \{y\}$ de l'état x par la lettre y,

— l'ensemble des états finaux $F = \text{Last}(E) \cup \{i\}$ si Null(E) est vrai, et $F = \text{Last}(E)$ sinon.

Question 19. Montrer que l'automate $\mathcal{A}_\ell$ reconnaît le langage décrit par l'expression E_ℓ.

On déduit de $\mathcal{A}_\ell$ un automate $\mathcal{A}$ en supprimant les indices sur les transitions. Ainsi, la transition $a_j \xrightarrow{a_k} a_k$ ($a \in \Sigma, j, k \in \{1, \ldots, \#E\}$) de $\mathcal{A}_\ell$ devient la transition $a_j \xrightarrow{a} a_k$ dans $\mathcal{A}$.

Cet automate est appelé *automate de Glushkov* associé à E.

Question 20. Justifier que dans l'automate $\mathcal{A}$, toutes les transitions qui arrivent dans un état portent la même étiquette.

Question 21. Montrer que l'automate $\mathcal{A}$ reconnaît le langage décrit par l'expression E.

Question 22. Donner en la justifiant une comparaison entre le nombre d'états de l'automate $\mathcal{A}$ ainsi obtenu et la taille de l'expression rationnelle E.

4.2 Construction de l'automate

Dans cette section, on s'intéresse à l'implantation de l'automate de Glushkov d'une expression rationnelle.

Pour toute expression rationnelle E, les ensembles First(E), Last(E) et Follow(E) sont finis par construction. On suppose qu'on dispose d'une classe `Set` qui nous servira à représenter et manipuler des ensembles finis, dont voici un extrait de la documentation et qu'on pourra supposer importée (on ne vous demande pas d'écrire le code de cette classe) :

```
class Set(builtins.object)
 |  représentation d'un ensemble fini
 |
 |  Methods defined here:
 |
 |  disjoint_extend(self, other)
 |      modifie l'objet courant en l'étendant avec le contenu de
 |      l'objet passé en paramètre et renvoie l'objet courant ainsi modifié,
 |      les deux objets doivent représenter des ensembles disjoints,
 |      le résultat représente l'union des deux ensembles;
 |      cette méthode s'exécute en temps linéaire en la taille de
 |      l'ensemble fourni en argument.
 |
 |  empty()
 |      renvoie un ensemble vide;
 |      cette méthode s'exécute en temps constant.
 |
 |  extend(self, other)
 |      modifie l'objet courant en l'étendant avec le contenu de
 |      l'objet passé en paramètre et renvoie l'objet courant ainsi modifié,
 |      le résultat représente l'union des deux ensembles (sans doublons);
 |      cette méthode s'exécute en temps proportionnel au produit
 |      de la taille de l'ensemble sur lequel la méthode est
 |      invoquée et de la taille de l'ensemble fourni en argument.
 |
 |  product(self, other)
 |      renvoie le produit cartésien de deux langages,
 |      le résultat est un ensemble de couples;
 |      cette méthode s'exécute en temps proportionnel au produit
 |      de la taille de l'ensemble sur lequel la méthode est
 |      invoquée et de la taille de l'ensemble fourni en argument.
 |
 |  singleton(e)
 |      renvoie un singleton contenant l'élément fourni en argument;
 |      cette méthode s'exécute en temps constant.
```

Une façon d'implanter une telle classe `Set` est d'utiliser un tableau redimensionnable. On obtient alors la complexité linéaire de la méthode `disjoint_extend` en suivant le même principe de redimensionnement que celui expliqué en section 1.

Remarque : On n'utilise pas le type **set** de PYTHON car on souhaite pouvoir exploiter la différence de complexité entre une union disjointe et une union qui a priori ne l'est pas, et on proposera une optimisation de la classe ci-dessus plus tard dans le sujet.

On donne le code suivant pour calculer First, Last, Null et Follow d'une expression rationnelle locale non vide :

```
def glushkov(exp):
 1  """renvoie first, last, null et follow de l'expression exp
 2  exp doit être locale et non vide
 3  first, last et follow sont des instances de Set, null est un booléen"""
 4  assert(len(exp) > 0 and len(exp) <= 3)
 5  if len(exp) == 1:
 6      if exp[0] == '':
 7          first, last, null, follow = Set.empty(), Set.empty(), True, Set.empty()
 8      else:
 9          first, last = Set.singleton(exp[0]), Set.singleton(exp[0])
10          null, follow = False, Set.empty()
11  elif len(exp) == 3:
12      assert(exp[0] in ['+', '.'])
13      first1, last1, null1, follow1 = glushkov(exp[1])
14      first2, last2, null2, follow2 = glushkov(exp[2])
15      if exp[0] == '+':
16          first = first1.disjoint_extend(first2)
17          last = last1.disjoint_extend(last2)
18          null = null1 or null2
19          follow = follow1.disjoint_extend(follow2)
20      else:
21          first = first1 if not null1 else first1.disjoint_extend(first2)
22          last = last2 if not null2 else last2.disjoint_extend(last1)
23          null = null1 and null2
24          follow = follow1.disjoint_extend(follow2).disjoint_extend(last1.product(first2))
25  else:
26      assert(exp[0] == '*')
27      first1, last1, null1, follow1 = glushkov(exp[1])
28      first, last, null = first1, last1, True
29      follow = follow1.extend(last1.product(first1))
30  return first, last, null, follow
```

Question 23. Dérouler la fonction `glushkov` sur l'expression `['+', ['*', ['.', ['a1'] , ['b2'] ] ], ['.', ['a3'], ['a4']]]`.

Question 24. La fonction `glushkov` réalise un parcours de l'arbre syntaxique de l'expression. Comment est appelé ce parcours ?

Question 25. Justifier que les utilisations de `disjoint_extend` sont adéquates.

Question 26. Donner un exemple pour expliquer pourquoi la dernière extension du code (ligne 29) n'est pas une extension disjointe.

Question 27. Montrer la correction totale de la fonction `glushkov`.

On s'intéresse maintenant à la complexité temporelle de la fonction `glushkov`, en séparant l'analyse en fonction de ce qu'on calcule parmi First, Last, Null et Follow. On note n la longueur de l'expression fournie en argument.

Tournez la page S.V.P.

Question 28. Montrer que la complexité temporelle du calcul de Null en suivant l'algorithme de la fonction `glushkov` est en $\Theta(n)$.

On admet que la complexité des calculs de First et Last en suivant l'algorithme de la fonction `glushkov` est linéaire en la longueur de l'expression de départ.

Question 29. Montrer que la complexité dans le pire des cas du calcul de Follow en suivant l'algorithme de la fonction `glushkov` est en $\Omega(n^5)$ où n est la longueur de l'expression fournie en argument. On pourra par exemple considérer la famille d'expressions locales définie récursivement par $F_1 = a_1^*$ et $F_n = (F_{n-1} + a_n)^*$. On rappelle que les complexités des méthodes de la classe `Set` sont données dans la documentation de cette même classe.

4.3 Amélioration de la complexité

L'extension non disjointe dans la fonction `glushkov` joue un rôle non négligeable dans sa complexité. Dans cette partie, nous allons voir comment nous ramener à une extension disjointe.

Soit une expression locale E de la forme F^*. Le calcul de `follow` repose sur la relation

$$\mathrm{Follow}(E) = \mathrm{Follow}(F^*) = \mathrm{Follow}(F) \cup (\mathrm{Last}(F) \times \mathrm{First}(F)) \ ,$$

où l'union n'est pas nécessairement disjointe, comme on l'a vu à la question 26.

Or, on peut exprimer cette relation avec l'union disjointe suivante (en notant $\sqcup$ l'opérateur d'union disjointe) :

$$\mathrm{Follow}(E) = \mathrm{Follow}(F^*) = \mathrm{Follow}(F) \sqcup \big((\mathrm{Last}(F) \times \mathrm{First}(F)) \setminus \mathrm{Follow}(F)\big) \ .$$

Notons

$$\mathrm{RFoll}(F) = \big(\mathrm{Last}(F) \times \mathrm{First}(F)\big) \setminus \mathrm{Follow}(F) \ .$$

Le but des questions suivantes est de montrer que si F est une expression rationnelle locale, cet ensemble peut se calculer inductivement.

Question 30. Donner les valeurs de $\mathrm{RFoll}(\varepsilon)$, $\mathrm{RFoll}(a)$ pour une lettre $a \in \Sigma$ et $\mathrm{RFoll}(E^*)$ pour une expression rationnelle locale E.

Question 31. Montrer que si E et F sont des expressions rationnelles locales, on a :

$$\begin{aligned} \mathrm{RFoll}(E + F) &= \mathrm{RFoll}(E) \sqcup \mathrm{RFoll}(F) \sqcup \mathrm{Last}(E) \times \mathrm{First}(F) \sqcup \mathrm{Last}(F) \times \mathrm{First}(E) \\ \mathrm{RFoll}(E \cdot F) &= \mathrm{Last}(F) \times \mathrm{First}(E) \sqcup \mathrm{Null}(F) \cdot \mathrm{RFoll}(E) \sqcup \mathrm{Null}(E) \cdot \mathrm{RFoll}(F) \ , \end{aligned}$$

où le produit $b \cdot X$ d'un booléen b par un ensemble X est l'ensemble vide si b vaut faux, et l'ensemble X si b vaut vrai.

Question 32. Expliquer brièvement (sans écrire de code) comment on peut utiliser cette formule pour améliorer la complexité du calcul de Follow de manière efficace.

Question 33. Expliquer (sans écrire de code) comment on peut implanter des ensembles avec des listes chaînées de sorte que la complexité de l'extension disjointe soit constante et les complexités des autres opérations présentes dans la classe `Set` restent inchangées. Quelle serait la conséquence d'un tel choix sur la complexité de la fonction `glushkov` (en supposant qu'on a utilisé l'implantation suggérée à la question 32) ?

Partie III. Test d'égalité de langages reconnus par automates

Pour tester si deux expressions $\mathcal{E}$ et $\mathcal{F}$ décrivent le même langage, on se propose de calculer leurs automates de Glushkov, qui sont ensuite déterminisés grâce à la construction classique par sous-ensembles. Il faut alors décider si deux automates déterministes reconnaissent le même langage. Nous verrons deux façons de procéder.

On travaillera uniquement sur des automates déterministes et complets. Soit $\mathcal{A} = (\Sigma, Q, \delta, i_0, F)$ un tel automate où :

- Σ est un alphabet fini non vide contenant les *lettres* ;
- Q est un ensemble fini non vide contenant les *états* ;
- $\delta : Q \times \Sigma \to Q$ est l'application des *transitions* ;
- $i_0 \in Q$ est l'état initial ;
- $F \subseteq Q$ est l'ensemble des *états terminaux*.

Quand $\delta(p, a) = q$ on dira qu'il y a une transition de p à q étiquetée par a, et on le notera également $p \xrightarrow{a} q$. La *taille* d'un automate est son nombre d'états, elle est notée $\|\mathcal{A}\|$. On a ainsi $\|\mathcal{A}\| = |Q|$. On notera $\mathcal{L}(\mathcal{A})$ le langage reconnu par $\mathcal{A}$.

5 Partitions des états

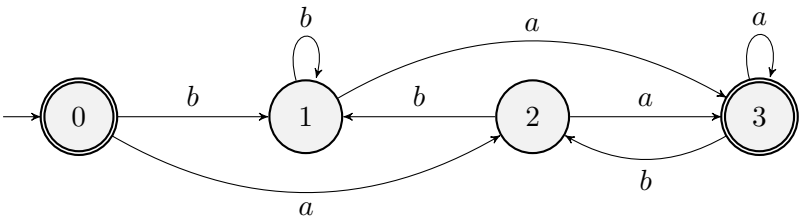
On rappelle que si E est ensemble fini non vide, une *partition* $\mathcal{P}$ de E est un ensemble $\{E_0, \ldots, E_{\ell-1}\}$ de sous-ensembles non vides de E dont l'union est E tout entier ($\cup_{i=0}^{\ell-1} E_i = E$) et qui sont deux à deux disjoints : si i et j sont deux indices différents entre 0 et $\ell-1$, alors $E_i \cap E_j = \emptyset$. Chaque E_i est appelé une *part* de la partition. Par exemple, $\mathcal{P} = \{\{0, 1, 4\}, \{2, 5\}, \{3\}\}$ est une partition de $\{0, 1, 2, 3, 4, 5\}$ en trois parts.

Dans toute la suite on ne considérera que des partitions d'ensembles d'états d'automates déterministes et complets. Si $\mathcal{A} = (\Sigma, Q, \delta, i_0, F)$ est un tel automate, et $\mathcal{P}$ est une partition de Q, alors pour tout $(p, q) \in Q^2$ on note $p \sim_{\mathcal{P}} q$ le fait que p et q sont dans la même part de la partition $\mathcal{P}$. On définit le Σ-*raffinement* de $\mathcal{P}$ comme étant l'unique partition $\mathcal{P}'$ de Q telle que

$$\forall (p, q) \in Q^2, \quad p \sim_{\mathcal{P}'} q \quad \Longleftrightarrow \quad \begin{cases} p \sim_{\mathcal{P}} q, \\ \text{et} \\ \forall a \in \Sigma,\ \delta(p, a) \sim_{\mathcal{P}} \delta(q, a). \end{cases} \tag{1}$$

Tournez la page S.V.P.

Question 34. On considère l'automate ci-dessous et la partition $\mathcal{P} = \{\{0,1\},\{2,3\}\}$ de son ensemble d'états. Calculer le Σ-raffinement de $\mathcal{P}$.



On souhaite à présent calculer efficacement le Σ-raffinement d'une partition de l'ensemble d'états d'un automate. Pour simplifier les représentations en machine, on considérera que l'alphabet Σ est $\{0,\dots,m-1\}$ et que si l'automate possède n états, son ensemble d'états est $\{0,\dots,n-1\}$. Un tel automate déterministe et complet sera encodé en PYTHON par un tuple `(m, n, delta, q0, F)`, où `m` est le nombre de lettres de l'alphabet, `n` est le nombre d'états de l'automate, `q0` est l'état initial (un entier entre 0 et $n-1$) et

— `delta` est une liste contenant n listes de longueur m, appelé la *table des transitions* ; pour tous états $p, q \in \{0,\dots,n-1\}$ et toute lettre $a \in \{0,\dots,m-1\}$ on a `delta[p][a]=q` si et seulement si $p \xrightarrow{a} q$ (on rappelle que les automates de cette partie sont déterministes et complets, ce qui permet un tel encodage) ;

— `F` est une liste de n booléens avec, pour tout état $p \in \{0,\dots,n-1\}$, `F[p] == True` si et seulement si p est terminal.

Les partitions de l'ensemble des états d'un automate sont représentées de la façon suivante. Si $\mathcal{P}$ est une partition de $\{0,\dots,n-1\}$ qui contient c parts, on encodera $\mathcal{P}$ par une liste `part` de longueur n, contenant des entiers de $\{0,\dots,c-1\}$, et tel que pour tous états $p, q \in \{0,\dots,n-1\}$, p et q sont dans la même part de $\mathcal{P}$ si et seulement si `part[p] == part[q]`. Autrement dit, on numérote les parts avec des nombres entre 0 et $c-1$, et la liste `part` associe à chaque état le numéro de sa part. On remarque qu'un tel encodage n'est pas unique car on peut échanger les numéros des parts.

Exemple. Si $n = 7$ et que la partition est $\{\{0,3,4\},\{1\},\{2,5,6\}\}$, on peut la représenter par `[0,1,2,0,0,2,2]` ou encore `[1,2,0,1,1,0,0]`.

Question 35. Écrire la fonction PYTHON `nombre_parts(part)` qui calcule le nombre de parts de la partition encodée par `part`, en temps $\mathcal{O}(n)$, où n est la taille de `part`. On ne demande pas de justifier la complexité.

On souhaite écrire la fonction PYTHON `raffine(A, part)` qui calcule et renvoie le Σ-raffinement de la partition encodée par une liste `part` de l'ensemble d'états d'un automate `A` donné également en argument en utilisant l'équation (1). Pour cela, on va associer à chaque état p la liste `s[p]` de longueur $m+2$ suivante :

```
s[p] = [part[p], part[ delta[p][0] ], ..., part[ delta[p][m-1] ], p].
```

On encode donc dans `s[p]` les informations utiles pour l'équation (1) ; on y a ajouté `p` à la fin pour pouvoir directement retrouver l'état p à partir de `s[p]`.

Question 36. En utilisant le tri lexicographique de la section 2, écrire une implantation en PYTHON de la fonction `raffine(A, part)` en temps $\mathcal{O}(nm)$: elle doit renvoyer un encodage du Σ-raffinement de la partition encodée par `part`, où `A` est un encodage de l'automate déterministe et complet avec n états sur un alphabet de taille m. On ne demande pas de justifier la complexité.

6 Partition de Nerode

Soit $\mathcal{A} = (\Sigma, Q, \delta, i_0, F)$ un automate déterministe et complet. Soit $\overline{F} = Q \setminus F$ le complémentaire de F dans Q. La partition $\mathcal{P}_0$ de Q est la partition telle que pour tout $(p, q) \in Q^2$, $p \sim_{\mathcal{P}_0} q$ si et seulement si $(p, q) \in F^2$ ou $(p, q) \in \overline{F}^2$: autrement dit p et q sont dans la même part de $\mathcal{P}_0$ quand ils sont soit tous les deux terminaux, soit tous les deux non-terminaux.

Question 37. Écrire une fonction PYTHON `calcule_partition0(A)` qui calcule et renvoie la partition $\mathcal{P}_0$ des états de l'automate déterministe et complet encodé par `A`, en temps $\mathcal{O}(n)$. On ne demande pas de justifier la complexité.

Soit $\mathcal{A}$ un automate déterministe et complet sur l'alphabet Σ. Pour tout entier $i \geq 1$, on définit récursivement la partition $\mathcal{P}_i$ comme étant le Σ-raffinement de $\mathcal{P}_{i-1}$. Il s'agit donc d'itérer i fois le Σ-raffinement de $\mathcal{P}_0$. Pour simplifier les notations, pour tout $i \in \mathbb{N}$ on notera $\sim_i$ au lieu de $\sim_{\mathcal{P}_i}$ dans la suite.

Question 38. Calculer $\mathcal{P}_0$, $\mathcal{P}_1$ et $\mathcal{P}_2$ pour l'automate de la question 34.

Pour un alphabet Σ et un entier $i \geq 0$, on note $\Sigma^{\leq i}$ l'ensemble des mots sur Σ dont la longueur est au plus i : $\Sigma^{\leq i} = \{u \in \Sigma^* \mid |u| \leq i\}$. Si $\mathcal{A} = (\Sigma, Q, \delta, i_0, F)$ est un automate déterministe et complet, pour tout $q \in Q$ on note $\mathcal{A}_q = (\Sigma, Q, \delta, q, F)$, l'automate obtenu en déplaçant l'état initial en q. On note $\mathcal{L}_q$ le langage reconnu par $\mathcal{A}_q$. Ainsi, par exemple, $\mathcal{L}_{i_0}$ est le langage $\mathcal{L}(\mathcal{A})$ reconnu par $\mathcal{A}$ car $\mathcal{A} = \mathcal{A}_{i_0}$.

Question 39. Soit $\mathcal{A} = (\Sigma, Q, \delta, i_0, F)$ un automate déterministe et complet. Montrer que pour tout $i \in \mathbb{N}$ et pour tout $(p, q) \in Q^2$, $p \sim_i q$ si et seulement si $\mathcal{L}_p \cap \Sigma^{\leq i} = \mathcal{L}_q \cap \Sigma^{\leq i}$.

Soit $\mathcal{A} = (\Sigma, Q, \delta, i_0, F)$ un automate déterministe et complet. On définit la *partition de Nerode* $\mathcal{N}_\mathcal{A}$ associée à $\mathcal{A}$ comme étant l'unique partition de Q telle que pour tout $(p, q) \in Q^2$, p et q sont dans la même part de $\mathcal{N}_\mathcal{A}$ si et seulement si $\mathcal{L}_p = \mathcal{L}_q$. Pour simplifier les notations, on écrira $p \sim q$ au lieu de $p \sim_{\mathcal{N}_\mathcal{A}} q$ pour indiquer que p et q sont dans la même part de $\mathcal{N}_\mathcal{A}$. L'algorithme de Moore ci-dessous permet de calculer la partition de Nerode d'un automate encodé par `A` (on considère qu'appliquer la fonction `nombre_parts` à `None` renvoie 0).

```
def moore(A):
 1  """A est un automate déterministe et complet,
 2  qui respecte l'encodage du sujet. la fonction renvoie
 3  la partition de Nerode de A."""
 4  ancienne_part = None
 5  part = calcule_partition0(A)
 6  while nombre_parts(ancienne_part) != nombre_parts(part):
 7      ancienne_part = part
 8      part = raffine(A, part)
 9  return part
```

Question 40. Montrer que si l'automate possède $n \geq 2$ états, il y a au plus $n-1$ itérations de la boucle `while`.

Question 41. Montrer la correction totale de l'algorithme de Moore.

Question 42. Montrer que l'algorithme de Moore a une complexité en $\mathcal{O}(m\,n^2)$, où n est le nombre d'états de l'automate et m le nombre de lettres de l'alphabet.

Question 43. Montrer que l'algorithme de Moore a une complexité en $\Theta(m\,n^2)$, où n est le nombre d'états de l'automate et m le nombre de lettres de l'alphabet.

7 Test d'égalité utilisant la partition de Nerode

Pour les deux questions suivantes, vous pouvez utiliser la définition de la partition de Nerode, ou bien le calcul qui en est fait par l'algorithme de Moore, qui est correct d'après la question 41.

Question 44. Soient p et q deux états de l'automate. Montrer que si $p \sim q$ alors $(p, q) \in F^2$ ou $(p, q) \in \overline{F}^2$.

Question 45. Soient p et q deux états de l'automate. Montrer que si $p \sim q$ alors pour tout lettre $a \in \Sigma$, on a $\delta(p, a) \sim \delta(q, a)$.

Les deux propriétés des questions 44 et 45 permettent de définir l'*automate quotient* de $\mathcal{A}$ par sa partition de Nerode qui est l'automate déterministe et complet $\mathcal{A}/\!\sim = (\Sigma, \mathcal{N}_\mathcal{A}, \delta_\sim, Q_0, F_\sim)$ dont l'ensemble d'états est l'ensemble des parts Q_0, Q_1, … $Q_{\ell-1}$ de $\mathcal{N}_\mathcal{A}$, où Q_0 est la part de i_0 et :

— pour tout $Q_i \in \mathcal{N}_\mathcal{A}$ et pour tout $a \in \Sigma$, on définit $\delta_\sim(Q_i, a)$ comme étant l'unique part Q_j de $\mathcal{N}_\mathcal{A}$ telle que pour tout $p \in Q_i$, $\delta_\sim(p, a) \in Q_j$ (cela est garanti par la question 45) ;
— $F_\sim$ est l'ensemble des $Q_i \in \mathcal{N}_\mathcal{A}$ composés uniquement d'états terminaux (la question 44 assure que les états d'une même part sont tous terminaux ou tous non-terminaux).

Question 46. Calculer l'automate quotient de $\mathcal{A}$ par sa partition de Nerode, où $\mathcal{A}$ est l'automate de la question 34.

Pour finaliser le test d'équivalence, on utilise un résultat (admis) de théorie des automates : soient $\mathcal{A} = (\Sigma, Q_\mathcal{A}, \delta_\mathcal{A}, i_\mathcal{A}, F_\mathcal{A})$ et $\mathcal{B} = (\Sigma, Q_\mathcal{B}, \delta_\mathcal{B}, i_\mathcal{B}, F_\mathcal{B})$ deux automates déterministes et complets dont tous les états sont accessibles depuis leurs états initiaux. Alors $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$ si et seulement si $\mathcal{A}/\!\sim$ et $\mathcal{B}/\!\sim$ sont *isomorphes*, c'est-à-dire qu'il existe une bijection ϕ de $Q_\mathcal{A}$ dans $Q_\mathcal{B}$ telle que

— $\phi(i_\mathcal{A}) = i_\mathcal{B}$,

— pour tout $p \in Q_\mathcal{A}$ et tout $a \in \Sigma$, $\phi(\delta_\mathcal{A}(p, a)) = \delta_\mathcal{B}(\phi(p), a)$,

— $\phi(F_\mathcal{A}) = F_\mathcal{B}$.

En particulier, ce résultat utilise le fait que $\mathcal{A}$ et $\mathcal{A}/\!\sim$ reconnaissent le même langage, ce qu'on ne vous demande pas de montrer.

Question 47. Sans écrire le programme, expliquer comment on peut en temps $\mathcal{O}(|\Sigma|(\|\mathcal{A}\| + \|\mathcal{B}\|))$ tester si deux automates déterministes et complets $\mathcal{A}$ et $\mathcal{B}$, dont tous les états sont accessibles depuis leurs états initiaux, sont isomorphes. On rappelle que $\|\mathcal{A}\|$ est le nombre d'états de l'automate $\mathcal{A}$ et $|\Sigma|$ est le cardinal de l'alphabet.

En conclusion de cette section, on remarque qu'on peut en déduire un algorithme pour tester en temps $\mathcal{O}(m\, n^2)$ si deux automates déterministes et complets $\mathcal{A}$ et $\mathcal{B}$ reconnaissent le même langage, où $n = \max\{\|\mathcal{A}\|, \|\mathcal{B}\|\}$ et $m = |\Sigma|$:

— on enlève les états qui ne sont pas accessibles depuis les états initiaux dans $\mathcal{A}$ et $\mathcal{B}$ en les identifiants avec un parcours en profondeur ;

— on applique l'algorithme de Moore aux deux automates, et on calcule leur automates quotients par leurs partitions de Nerode respectives ;

— on teste si les deux automates quotients sont isomorphes.

8 Test d'égalité avec la structure "unir & trouver"

L'objet de cet section est de proposer un algorithme plus efficace pour déterminer si deux automates déterministes et complets $\mathcal{A} = (\Sigma, Q_\mathcal{A}, \delta_\mathcal{A}, i_\mathcal{A}, F_\mathcal{A})$ et $\mathcal{B} = (\Sigma, Q_\mathcal{B}, \delta_\mathcal{B}, i_\mathcal{B}, F_\mathcal{B})$, définis sur le même alphabet Σ, reconnaissent le même langage.

L'algorithme étudié utilise également des partitions mais diffère fondamentalement des parties précédentes.

On travaille sur les deux automates simultanément, l'ensemble dont on considère les partitions est $Q = Q_\mathcal{A} \cup Q_\mathcal{B}$, l'union des ensembles d'états de $\mathcal{A}$ et de $\mathcal{B}$, que l'on suppose disjoints. Si $p \in Q$, on note $\mathcal{L}_p$ le langage reconnu en changeant l'état initial de l'automate qui contient p pour le placer en p.

Informellement, l'algorithme fonctionne de la façon suivante. Au début, chaque état est seul dans sa part. Ensuite, on fait l'union, autant que possible, des parts des états p et q qui

Tournez la page S.V.P.

```
class Partition(builtins.object)
 |
 |  Methods defined here:
 |
 |  __init__(self, n)
 |      initialise une partition de {0,...,n-1}
 |      où chaque élément est seul dans sa part
 |
 |  trouver(self, x)
 |      renvoie un identifiant unique de la part de x :
 |      x et y sont dans la même part si et seulement si
 |      trouver(x) == trouver(y)
 |
 |  unir(self, x, y)
 |      modifie la partition en réalisant l'union des parts de x et de y
```

FIGURE 2 – Méthodes de la classe `Partition` implantant la structure Unir et Trouver.

doivent nécessairement vérifier $\mathcal{L}_p = \mathcal{L}_q$ si $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$: par exemple, la première union consiste à créer la part $\{i_\mathcal{A}, i_\mathcal{B}\}$, puisque si $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$ alors $\mathcal{L}_{i_\mathcal{A}} = \mathcal{L}_{i_\mathcal{B}}$. Toujours sous l'hypothèse que $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$, pour chaque $a \in \Sigma$, on doit avoir $\mathcal{L}_p = \mathcal{L}_q$ quand $p = \delta_\mathcal{A}(i_\mathcal{A}, a)$ et $q = \delta_\mathcal{B}(i_\mathcal{B}, a)$, il faut donc réaliser l'union des parts contenant $\delta_\mathcal{A}(i_\mathcal{A}, a)$ et $\delta_\mathcal{B}(i_\mathcal{B}, a)$ pour toute lettre a. L'algorithme utilise une pile, implantée par une liste PYTHON, pour garder la trace des paires d'états à considérer à chaque étape. L'algorithme a deux façons de s'arrêter :

— soit il doit unir les parts de p et q alors que seulement l'un de ces deux états est final, auquel cas il renvoie faux ;

— soit il n'y a plus de paire d'états à considérer, auquel cas il renvoie vrai.

Une implantation de l'algorithme est proposée à la Fig. 3 page 17 elle utilise la méthode `pop()` qui permet d'enlever le dernier élément d'une liste PYTHON et de le renvoyer, en temps constant. Elle utilise également une implantation de la structure de données “unir & trouver” (“union & find” en anglais), qui permet de travailler sur des partitions d'un ensemble fini non vide $\{0, \dots, n-1\}$ comme décrit Fig. 2. L'implantation est réalisée de sorte que si on initialise une partition de taille n puis que l'on effectue t appels à la fonction `trouver` ou `unir`, la complexité totale en temps est en $\mathcal{O}(t\alpha(n))$, où α est l'inverse de la fonction d'Ackermann. La fonction α tend vers l'infini extrêmement lentement (on considère en général que $\alpha(n) \leq 5$ pour les valeurs de n que l'on peut utiliser en pratique).

L'entrée de `test_egalite_langages` est constituée des deux automates déterministes et complets $\mathcal{A}$ et $\mathcal{B}$, définis sur le même alphabet, encodés par `A=(mA,nA,deltaA,iA,FA)` et `B=(mB,nB,deltaB,iB,FB)`. Comme cette fonction doit travailler avec une partition de l'union de $Q_\mathcal{A}$ et de $Q_\mathcal{B}$ et que dans notre choix d'encodage $Q_\mathcal{A} = \{0, \dots, n_\mathcal{A} - 1\}$ et $Q_\mathcal{B} = \{0, \dots, n_\mathcal{B} - 1\}$ ne sont pas disjoints, on différencie les états de $Q_\mathcal{B}$ en leur ajoutant $n_\mathcal{A}$: l'ensemble de la partition est $\{0, \dots, n_\mathcal{A} + n_\mathcal{B} - 1\}$, et les entiers de $\{0, \dots, n_\mathcal{A} - 1\}$ représentent les états de $\mathcal{A}$, alors que les entiers de $\{n_\mathcal{A}, \dots, n_\mathcal{A} + n_\mathcal{B} - 1\}$ représentent les états de $\mathcal{B}$.

Question 48. Montrer qu'à tout moment lors de l'exécution de `test_egalite_langages`, pour tout couple d'états (p, q) de la liste `a_faire`, il existe un mot $u \in \Sigma^*$ tel que $\delta_\mathcal{A}(i_\mathcal{A}, u) = p$ et $\delta_\mathcal{B}(i_\mathcal{B}, u) = q$. En déduire que si le programme renvoie faux, les langages reconnus par $\mathcal{A}$ et par $\mathcal{B}$ sont différents.

```
def test_egalite_langages(A, B):
    """A et B sont des encodages d'automates déterministes et complets
    sur le même alphabet"""
    mA, nA, deltaA, iA, FA = A
    mB, nB, deltaB, iB, FB = B
    assert(mA == mB)
    partition = Partition(nA + nB)
    a_faire = [(iA, iB)]
    while len(a_faire)>0:
        p, q = a_faire.pop()
        if partition.trouver(p) != partition.trouver(q + nA):
            if FA[p] != FB[q]:
                return False
            else:
                partition.unir(p, q + nA)
                for a in range(mA):
                    a_faire.append( (deltaA[p][a], deltaB[q][a]) )
    return True
```

FIGURE 3 – Programme PYTHON pour tester l'égalité des deux langages reconnus par les automates déterministes et complets $\mathcal{A}$ et $\mathcal{B}$.

Dans la suite, pour tout état $p \in Q_{\mathcal{A}} \cup Q_{\mathcal{B}}$ et toute lettre $a \in \Sigma$, on note $\delta(p,a) = \delta_{\mathcal{A}}(p,a)$ si $p \in Q_{\mathcal{A}}$ et $\delta(p,a) = \delta_{\mathcal{B}}(p,a)$ si $p \in Q_{\mathcal{B}}$.

Question 49. On suppose que le programme `test_egalite_langages` renvoie vrai. On note $\mathcal{P}_{\text{fin}}$ la partition obtenue à la fin de l'exécution du programme. Montrer la propriété suivante : à tout moment lors de l'exécution du programme, si deux états p et q sont dans la même part P de la partition courante `partition`, alors, à la fin de l'exécution, $\delta(p,a)$ et $\delta(q,a)$ sont dans la même part de $\mathcal{P}_{\text{fin}}$, pour toute lettre $a \in \Sigma$. On pourra utiliser une récurrence sur le cardinal de la part P.

Question 50. En déduire que si le programme renvoie vrai, les deux automates reconnaissent le même langage.

Question 51. Montrer que `test_egalite_langages` s'exécute en temps $\mathcal{O}(m\, n\alpha(n))$, où $n = \|\mathcal{A}\| + \|\mathcal{B}\|$ et $m = \|\Sigma|$.

* *
*