Exercice 1. Automate et palindrome
On fixe un alphabet Σ avec $|\Sigma| > 1$. Un mot $w \in \Sigma^*$ est un palindrome s'il s'écrit $w = a_1 \cdots a_n$ et qu'on a $a_i = a_{n-i+1}$ pour tout $1 \leq i \leq n$. On note $\Pi \subseteq \Sigma^*$ le langage des palindromes. Pour un automate fini A sur Σ, on note $L(A)$ le langage reconnu par A.

1. Soit $\Pi_n := \Pi \cap \Sigma^n$. Montrer que pour tout automate fini déterministe complet A, pour tout $n \in \mathbb{N}$, si $L(A) \cap \Sigma^{2n} = \Pi_{2n}$, alors A a au moins $|\Sigma|^n$ états.
2. En déduire que le langage Π n'est pas régulier.
3. Étant donné un automate fini A sur Σ, peut-on calculer un automate A_Π qui reconnaisse $L(A) \cap \Pi$?
4. Pour tout mot $u = b_1 \cdots b_m$ de Σ^*, on note $\overline{u} := b_m \cdots b_1$ son miroir. Étant donné A, peut-on calculer un automate A'_Π qui reconnaisse $\{u \in \Sigma^* \mid u\overline{u} \in L(A)\}$?
5. On appelle Π_{pair} l'ensemble des palindromes de longueur paire, i.e., $\Pi_{\text{pair}} = \bigcup_{n \in \mathbb{N}} \Pi_{2n}$. Proposer un algorithme qui, étant donné un automate fini A sur Σ, détermine si $L(A) \cap \Pi_{\text{pair}}$ est vide, fini, ou infini. Discuter de sa complexité en temps et en espace.
6. Modifier l'algorithme de la question 4 pour calculer la cardinalité de $L(A) \cap \Pi_{\text{pair}}$ quand cet ensemble est fini, en faisant l'hypothèse que l'automate d'entrée A est déterministe. Comment la complexité est-elle affectée?
7. Modifier l'algorithme des questions 4 et 5 pour qu'il s'applique à $L(A) \cap \Pi$.

1. Soit A un tel automate fini déterministe d'ensemble d'états Q, et fixons $n \in \mathbb{N}$. Considérons la fonction $f : \Sigma^n \to Q$ qui associe à $w \in \Sigma^n$ l'état q (unique) auquel A aboutit en lisant w. Montrons que f est injective. Procédons par l'absurde et supposons que $f(u) = f(v)$ pour $u \neq v$ de Σ^n. On sait que $u\overline{u}$ est un palindrome de longueur $2n$ donc il y a un chemin étiqueté par $\overline{u}$ de $f(u)$ à un état final de A. En combinant ce chemin avec le chemin de l'état initial à q étiqueté par v, on conclut que l'automate accepte $v\overline{u}$. Comme $u \neq v$, c'est pourtant un mot de longueur $2n$ qui n'est pas un palindrome, contradiction. Ainsi, f est injective, donc $|Q| \geq |\Sigma^n| \geq |\Sigma|^n$.

2. Procédons par l'absurde et supposons que Π soit régulier. Soit A un automate fini déterministe complet qui reconnaisse Π, et soit n son nombre d'états. Comme $L(A) = \Pi$ par hypothèse, on sait que $L(A) \cap \Sigma^{2n} = \Pi_{2n}$, ainsi d'après la question précédente A a au moins $|\Sigma|^n \geq 2^n$ états, mais il en a n et on a $n < 2^n$, donc contradiction. Ainsi Π n'est pas régulier.

3. C'est manifestement déraisonnable : pour A un automate reconnaissant le langage régulier Σ^*, la question nous demanderait de calculer un automate qui reconnaisse $\Sigma^* \cap \Pi = \Pi$, et on sait d'après la question précédente qu'il n'en existe pas.

4. De façon un peu contre-intuitive, c'est possible. Posons $A = (Q, I, F, \delta)$ où Q est l'ensemble d'états de A, où I est l'ensemble d'états initiaux, où F est l'ensemble d'états finaux, et où $\delta \subseteq Q \times \Sigma \times Q$ est la relation de transition. On construit d'abord en temps linéaire l'automate $\overline{A} := (Q, F, I, \overline{\delta})$ où $\overline{\delta} := \{(q', a, q) \mid (q, a, q') \in \delta\}$. Il est clair que $\overline{A}$ reconnaît $\overline{L(A)} := \{\overline{u} \mid u \in L(A)\}$, puisqu'il y a une correspondance bijective entre les chemins acceptants dans A et dans $\overline{A}$, et l'effet de cette bijection sur l'étiquette des chemins est l'opération miroir.

   On construit ensuite (en temps quadratique en A) l'automate produit $A \times \overline{A}$ défini comme suit : $A \times \overline{A} := \left(Q \times Q, I \times F, F', \delta \times \overline{\delta}\right)$ où la relation de transition est définie comme $\overline{\delta} := \left\{\left((q, \overline{q'}), a, (q', \overline{q}) \mid \ (q, a, q') \in \delta \wedge \left(\overline{q'}, a, \overline{q}\right) \in \overline{\delta}\right.\right.$, et où les états finaux sont $\{(q, q) \mid q \in Q\}$. Il est clair que l'automate produit peut atteindre l'état $(q, \overline{q})$ en lisant un mot $w \in \Sigma^*$ si et seulement si l'automate A peut atteindre l'état q en lisant w (dans la première composante) et l'automate $\overline{A}$ peut atteindre l'état $\overline{q}$ en lisant w (dans la seconde composante), ce qui est le cas, par définition de $\overline{A}$, si et seulement s'il y a un chemin dans A étiqueté par $\overline{u}$ de $\overline{q}$ à un état final de F. En particulier, l'automate produit peut atteindre l'état (q, q) en lisant un mot $w \in \Sigma^*$ si et seulement si l'automate A peut atteindre q en lisant w et l'automate A a un chemin étiqueté par $\overline{u}$ de q à un état final. Ainsi, la définition de F' assure que l'automate produit accepte un mot $w \in \Sigma^*$ si et seulement s'il existe un état q tel que A a un chemin acceptant pour $w\overline{w}$ qui passe par un certain état q après la lecture de w, c'est-à-dire si et seulement si A accepte $w\overline{w}$. Ceci établit que la construction est correcte.

   [Cette construction est donnée dans [ALR ${ }^{+}$09], Lemma 2.]

5. La construction de l'automate produit A'_Π de la question précédente s'effectue en temps quadratique en A. Il est clair que $L(A) \cap \Pi_{\text{pair}}$ a la même cardinalité que $L\left(A'_\Pi\right)$, puisque la fonction $f : \Sigma^* \to \Pi_{\text{pair}}$ définie par $f(u) := u\overline{u}$ pour tout $u \in \Sigma^*$ définit une bijection entre $L\left(A'_\Pi\right)$ et $L(A) \cap \Pi_{\text{pair}}$. Ainsi, il suffit de déterminer si $L\left(A'_\Pi\right)$ est vide, fini, ou infini.

   On détermine d'abord si $L\left(A'_\Pi\right)$ est non-vide en vérifiant qu'il existe un chemin d'un état initial de A'_Π à un état final de A'_Π, en temps linéaire, par exemple avec un DFS.

   On détermine ensuite si A'_Π contient un cycle d'états accessibles et co-accessibles. Pour ce faire, on détermine les états accessibles et co-accessibles par un parcours de graphe, et ensuite on utilise un DFS pour déterminer si on rencontre un cycle dans ces sommets. Ceci est toujours en temps linéaire, et conclut.

6. Dans le cas où l'automate produit est déterministe, on peut compter le nombre de mots qu'il accepte par de la programmation dynamique. Spécifiquement, le nombre de mots acceptés à partir d'un état q est 1 ou 0 selon que q est final ou non, plus le nombre de mots acceptés à partir des états vers lesquels q a des transitions sortantes. Noter que, comme l'automate est déterministe, les étiquettes de ces transitions sont différentes, ainsi les ensembles de mots concernés sont disjoints, et c'est effectivement correct de faire la somme comme expliqué.

   On n'a en fait pas vraiment besoin que l'automate produit soit déterministe : il suffit qu'il soit inambigu, c'est-à-dire que tout mot accepté a un unique chemin acceptant. Dans ce cas, la construction présentée reste correcte, parce que si un état q a des transitions étiquetées par la même lettre vers deux états distincts q_1 et q_2, alors la définition de l'inambiguïté impose que l'ensemble des mots acceptés à partir de q_1 et celui des mots acceptés à partir de q_2 sont disjoints.

   Il suffit alors d'observer que, si A est déterministe (ou, en fait, s'il est inambigu), alors $\overline{A}$ est toujours inambigu (même si pas forcément déterministe). Maintenant, le produit $A \times \overline{A}$ avec l'ensemble d'états finaux indiqué est également inambigu : si un mot $u \in \Sigma^*$ avait un chemin vers (q, q) et vers (q', q') dans l'automate produit avec $q \neq q'$, alors on saurait que $u\overline{u}$ a deux chemins acceptants distincts dans A (un où l'état intermédiaire est q, un autre où c'est q'), ce qui contredirait le fait que A soit inambigu. Ainsi, si A est inambigu on peut construire le produit avec la même complexité qu'avant, il est inambigu, et on compte la taille du langage qu'il accepte comme expliqué. Comme cet ensemble est en bijection avec $L(A) \cap \Pi_{\text{pair}}$ (comme prouvé à la question 4), on a établi le résultat.

   [L'hypothèse que l'automate d'entrée est déterministe ou inambigu est probablement indispensable, parce que compter le nombre de mots acceptés par un automate quelconque est #P-difficile : voir [KSM95] Theorem 2.1.]

7. L'automate produit A' de la question 4 ne gère intuitivement que les u qui se complètent en un palindrome de longueur paire. Ceci dit, pour chaque $a \in \Sigma$, on peut adapter l'ensemble d'états finaux de l'automate produit pour construire un automate A'_a qui reconnaisse exactement $\{u \in \Sigma^* \mid ua\overline{u} \in L(A)\}$, c'est-à-dire le langage des mots u qui se complètent en un palindrome de longueur impaire accepté par A en ajoutant a comme lettre centrale, puis le miroir de u. On définit A'_a pour chaque $a \in \Sigma$ exactement comme A' mais avec l'ensemble d'états finaux $F'_a := \{(q, q') \mid (q, a, q') \in \delta\}$, ce qui est clairement correct : un mot $u \in \Sigma^*$ a un chemin acceptant dans A'_a finissant en (q, q') si et seulement s'il y a un chemin d'un état initial de A à q étiqueté par u, une transition dans A étiquetée par a de q à q', et un chemin de q' à un état final de A étiqueté par $\overline{u}$ dans A. Du reste, si A est inambigu alors tous ces automates produits le sont, par le même raisonnement qu'à la question précédente. On peut ainsi appliquer la construction de la question précédente à A' et à A'_a pour chaque $a \in \Sigma$, ce qui ne fait que rajouter un facteur $|\Sigma|$ à la complexité : on peut obtenir la cardinalité du nombre de palindromes reconnus en sommant les quantités pour A' et pour chaque A'_a (en traitant ∞ de la manière attendue). Ceci est correct parce que les palindromes de $\Pi \cap L(A)$ se partitionnent entre ceux de longueur paire et ceux de longueur impaire dont la lettre centrale est $a \in \Sigma$ pour chaque a. (En revanche, noter que le langage de A' et celui des A'_a ne sont pas forcément disjoints, vu qu'il est tout à fait possible, par exemple, que A accepte $u\overline{u}$ et $ua\overline{u}$ pour diverses valeurs de $a \in \Sigma$. Autrement dit, il faut bien sommer la cardinalité de ces langages même si leur union n'est pas disjointe, pour la raison qu'on vient d'expliquer.)

Références

[ALR +09] Terry Anderson, John Loftus, Narad Rampersad, Nicolae Santean, and Jeffrey Shallit. Detecting palindromes, patterns and borders in regular languages. Information and Computation, 207(11), 2009. https://www.sciencedirect.com/-science/article/pii/ S0890540109000650. [KSM95] Sampath Kannan, Z. Sweedyk, and Steve Mahaney. Counting and random generation of strings in regular languages. In Proc. SODA, 1995.

Exercice 2. 3-COLOR

Soit $G = (V, E)$ un graphe non orienté. On appelle **k-coloration** de G une fonction $c : V \longrightarrow \{1, 2, \ldots, k\}$ telle que pour tout arc $(u, v) \in E$, on a $c(u) \neq c(v)$.
On considère le problème suivant :

3-COLOR

Entrée : un graphe G non orienté
Sortie : un booléen indiquant si G est 3-colorable

1. Montrer que 3-COLOR appartient à la classe NP.
2. Écrire une fonction OCaml `check_3color` qui prend en entrée un graphe G (représenté par liste d'adjacence) et une coloration c (un tableau) et qui renvoie un booléen indiquant si c est une coloration de G.
3. Donner une réduction de 3-COLOR à 3-SAT.

Dans la suite, on veut trouver une réduction de 3-SAT à 3-COLOR.
On considère une formule φ de 3-SAT de variables $x_1, ..., x_n$. On veut construire un graphe G qui soit 3-colorable si et seulement si φ est satisfiable.
On commence par ajouter, dans G, n sommets (encore appelés x_1, ..., x_n par abus de notation) correspondant à x_1, ..., x_n, n sommets correspondant à $\neg x_1$, ..., $\neg x_n$ et 3 sommets T, F, B reliés 2 à 2.
Dans un 3-coloriage de G, V et F doivent être de couleurs différentes. Chaque variable x_i sera considérée comme fausse si le sommet correspondant est de la même couleur que F et vraie s'il est de la même couleur que T.

4. Expliquer comment ajouter des arêtes à G pour que chaque variable soit vraie ou fausse (c'est-à-dire coloriée avec la même couleur que F ou la même couleur que T).
5. Dessiner un graphe (un *gadget*) avec (au moins) 3 sommets l_1, l_2, s que l'on pourrait ajouter à G tels que :
   - Si e_1 et e_2 sont de la même couleur que F, alors s doit être de même couleur que F.
   - Si e_1 ou e_2 est de la même couleur que T, alors il existe un coloriage de G où s est de la même couleur que T.
6. Soit $l_1 \vee l_2 \vee l_3$ une clause de φ. Expliquer comment ajouter un gadget à G de façon à ce que $l_1 \vee l_2 \vee l_3$ soit vraie si et seulement si G est 3-coloriable.
7. Montrer que 3-COLOR est NP-complet.
8. Appliquer la réduction ci-dessus à la formule $\varphi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_4)$.

Cependant, 2-COLOR est dans la classe P :

9. Écrire une fonction OCaml `2color` qui prend en entrée un graphe G (représenté par liste d'adjacence) avec n sommets et p arêtes et qui renvoie un booléen indiquant si G est 2-colorable, en $O(n + p)$.

Exercice 3. Réparation de mots

On fixe un langage régulier L sur un alphabet Σ. Étant donné un mot $w \in \Sigma^*$, une réparation par insertion du mot w pour le langage L est une décomposition $w = uv$ de w en deux mots, et un mot $z \in \Sigma^*$, de sorte que le mot uzv appartienne à L.

1. On pose L_0 le langage régulier défini par l'expression rationnelle $(ab)^*$. Proposer une réparation par insertion du mot $w_0 = aab$ pour L_0. Proposer une réparation par insertion du mot $w_0' = abab$ pour L_0. Le mot $w_0'' = aa$ admet-il une réparation par insertion pour L_0 ?

2. Une réparation par insertion finale d'un mot $w \in \Sigma^*$ pour un langage régulier L est un mot $z \in \Sigma^*$ tel que wz appartienne à L. Proposer un algorithme qui, étant donné w et L, détermine s'il existe une réparation par insertion finale de w pour L. Préciser sa complexité en temps et en espace.

3. On souhaite modifier l'algorithme de la question 1 pour que, lorsqu'une réparation par insertion finale existe, l'algorithme calcule un z correspondant qui soit de longueur minimale. Expliquer comment procéder, et préciser la complexité en temps et en espace.

4. Proposer un algorithme qui, étant donné un mot $w \in \Sigma^*$ et un langage régulier L, détermine s'il existe une réparation par insertion de w pour L. Préciser sa complexité en temps et en espace.

5. Modifier l'algorithme de la question 3 pour que, lorsqu'une réparation par insertion existe, l'algorithme calcule une décomposition $w = uv$ et un z correspondant qui soit de longueur minimale. Préciser la complexité en temps et en espace.

6. Une réparation par suppression d'un mot $w \in \Sigma^*$ pour un langage L est une décomposition $w = uzv$ de w de sorte que $uv \in \Sigma^*$. Proposer un algorithme naïf qui, étant donné w et L, détermine s'il existe une réparation par suppression de w pour L, et si oui, calcule une telle suppression telle que z soit de longueur minimale. Préciser sa complexité en temps et en espace.

7. Proposer un algorithme plus efficace pour déterminer, étant donné w et L, s'il existe une réparation par suppression de w pour L. La complexité de l'algorithme doit être linéaire en w.

8. Modifier l'algorithme de la question 6 pour que, quand une réparation par suppression existe, l'algorithme calcule une décomposition $w = uzv$ et un z correspondant qui soit de longueur minimale. On demande toujours une complexité linéaire en w.

9. Une réparation par insertions d'un mot $w \in \Sigma^*$ pour un langage L est un mot $w' \in \Sigma^*$ tel que w' appartienne à L et w soit un sous-mot de w', c'est-à-dire qu'il existe une fonction strictement croissante ϕ de $\{1, \ldots, |w|\}$ dans $\{1, \ldots, |w'|\}$ telle que $w_i = w'_{\phi(i)}$ pour tout $1 \leq i \leq |w|$, où $|w|$ dénote la longueur de w et w_i dénote la i-ème lettre de w.

   Proposer un algorithme qui détermine, étant donné L et w, s'il existe une réparation par insertions de w pour L, et le cas échéant calcule une telle réparation qui réalise un nombre minimal d'insertions, c'est-à-dire un w' de longueur minimale. Préciser la complexité en temps et en espace.

10. On souhaite à présent autoriser des réparations par insertion et par suppression de caractères du mot initial : partant de w, on supprime un certain nombre k_1 de caractères, puis on insère un nombre k_2 de caractères, de sorte à obtenir un mot de L. Le coût de cette réparation est $k_1 + k_2$. Étant donné L et w, calculer le coût minimal d'une réparation et un exemple de réparation de coût minimal.

1. Pour $w_0 = aab$, on pose $u_0 := a, v_0 := ab, z_0 := b$, et on obtient $u_0 z_0 v_0 = abab \in L_0$.

   Pour $w_0' = abab$, on a $w_0' \in L_0$ donc on peut par exemple poser $u_0' := w_0', v_0' := \varepsilon, z_0' := \varepsilon$, et on obtient $u_0' z_0' v_0' = abab \in L_0$.

   Pour $w_0'' = aa$, la seule décomposition de w_0'' qui évite de terminer par a (ce qui n'est le cas d'aucun mot de L_0) est $u_0'' := w_0'', v_0'' := \varepsilon$, mais alors pour tout $z_0'' \in \Sigma^*$ on a que $u_0'' z_0'' v_0''$ commence par aa donc n'appartient pas à L_0. Ainsi, w_0'' n'admet aucune réparation par insertion pour L_0.

2. Discussion avec le candidat: comment le langage L est-il représenté en entrée? On conviendra de le représenter par un automate fini non-déterministe.

   On demandera un pseudo-code de l'algorithme.

   L'algorithme calcule d'abord l'ensemble des états Q_w défini comme suit : un état q est dans Q_w si et seulement s'il existe une exécution de l'automate qui lise w et aille d'un état initial à q. On peut le calculer itérativement sur w : l'ensemble Q_ε est celui des états initiaux, et une fois déterminé Q_u pour un préfixe u de w, étant donné la lettre suivante $a \in \Sigma$ de w, l'ensemble Q_{ua} est l'ensemble des états accessibles par une transition étiquetée a à partir d'un état de Q_u. Ce calcul s'effectue en temps $O(|w| \times |\delta|)$ où $|\delta|$ est le nombre de transitions de l'automate : à chaque lettre du mot, on considère l'ensemble des transitions, et le nouvel ensemble des états où l'on peut se trouver sont les états cibles des transitions dont la source se trouve dans l'ensemble précédent des états où l'on peut se trouver.

   Ensuite, il suffit de déterminer si Q_w contient un état co-accessible. Il suffit pour cela de déterminer les états co-accessibles par un test d'accessibilité à partir des états finaux suivant le renversement des transitions : ce calcul s'effectue en temps $O(|\delta|)$. (À noter que, dans le cas idiot où A a plus d'états que de transitions, on peut utiliser δ pour se restreindre aux états finaux qui apparaissent dans δ pour l'exploration.) On vérifie ensuite si Q_w contient un tel état. Ainsi, la complexité en temps est de $O(|A| + |w| \times |\delta|)$, en prenant en compte la complexité de lire l'automate en entrée. La complexité en espace est de $O(|Q|)$ puisqu'il suffit au plus de mémoriser un ensemble d'états du calcul initial et une structure de données (par exemple une file d'états) pendant le test d'accessibilité.

3. On demandera un pseudo-code de l'algorithme obtenu à partir du pseudo-code de la question précédente.

   On modifie l'algorithme précédent: au lieu de simplement calculer les états co-accessibles, on calcule, pour chaque état co-accessible, la longueur du plus court chemin de cet état à un état final. En effet, cette longueur est celle de la plus petite insertion finale permettant d'atteindre un état final à partir de cet état, et un exemple de mot à insérer est donné par l'étiquette d'un tel chemin de longueur minimale.

   Le calcul mentionné peut se faire avec un BFS, avec les mêmes complexités en temps et en espace. Pour construire un exemple de chemin, il suffit de stocker pour chaque état atteint par le parcours un état prédécesseur, et on trouve un chemin allant d'un état donné à un état final en suivant les prédécesseurs.

   On obtient ainsi une réparation de longueur minimale en prenant l'état co-accessible de Q_w où la distance à un état final est minimale. Les complexités sont inchangées.

4. Pas de pseudo-code demandé. On calcule les ensembles d'états Q_u itérativement pour tous les préfixes u de w. On calcule itérativement, de la même manière, les ensembles d'états Q_v' pour chaque suffixe v de w définis comme suit: un état q est dans Q_v' si et seulement s'il existe un chemin de q à un état final étiqueté par v.

   On considère ensuite chaque position de w : en notant u, v le préfixe et le suffixe correspondant, on peut effectuer une réparation par insertion à cet endroit si et seulement s'il existe une paire $(q_u, q_v) \in Q_u \times Q_v'$ telle qu'il y ait un chemin de q_u à q_v dans l'automate. On peut déterminer cela par un test d'accessibilité en $O(|\delta|)$. La complexité en temps est toujours de $O(|A| + |w| \times |\delta|)$. La complexité en espace est à présent de $O(|w| \times |Q|)$ vu qu'il faut mémoriser tous les ensembles intermédiaires.

   Remarque : si le mot d'entrée est très long, une optimisation judicieuse en pratique serait probablement de déterminiser l'automate (ceci prend un temps indépendant du mot d'entrée, et assure que les ensembles Q_u seront tous de cardinal au plus 1). Ainsi, les ensembles d'états pour les préfixes sont en fait des singletons, donc on peut les calculer (ce qui prend un espace $O(|w|)$), puis calculer les ensembles pour les suffixes et faire le test. Ainsi, la complexité en espace tombe à $O(|w| + |Q|)$.

5. Pas de pseudo-code demandé. Il suffit de calculer un plus court chemin comme en question 2. Les complexités sont inchangées.

   Autre remarque: On pourrait par ailleurs envisager de pré-calculer, pour chaque paire d'états de l'automate, la distance d'un plus court chemin de l'un à l'autre, par exemple avec l'algorithme de Floyd-Warshall $\left(O\left(|Q|^3\right)\right)$. Ainsi, lors de la deuxième phase, au lieu de faire un BFS à chaque position du mot, il suffit de considérer chaque état de Q_v' et prendre le min de la distance avec les états de Q_u (ou l'unique état de Q_u s'il existe, dans le cas d'un automate déterministe).

On peut ensuite ne faire le calcul explicite du chemin que pour une décomposition de longueur minimale. Malgré tout, cette optimisation ne change pas la complexité asymptotique pour le temps d'exécution, qui reste $O(|A| + |w| \times \times |\delta|)$, i.e., dominée par le calcul des ensembles Q_u et Q'_v.

6. Pas de pseudo-code demandé. On teste simplement toutes les décompositions possibles de $w = uzv$, et on regarde pour chacune d'entre elles si $uv \in L$. On peut ensuite facilement en renvoyer une de longueur minimale (en testant les décompositions en prenant des z de longueur croissante). La complexité est en $O\left(|A| + |w|^3 \times |\delta|\right)$, puisqu'il y a $O\left(|w|^2\right)$ décompositions à tester, et que chaque test prend $O(|w| \times |\delta|)$.

7. On calcule pour chaque préfixe u de w trois ensembles d'états Q_u, Q'_u, Q''_u :

   - L'ensemble Q_u est l'ensemble des états où on peut aboutir après lecture de u depuis un état initial (sans suppression), exactement comme à la question 1.
   - L'ensemble Q'_u est l'ensemble des états où on peut aboutir après lecture d'un préfixe de u (correspondant intuitivement à une suppression en cours) : on définit $Q'_\varepsilon := Q_\varepsilon$, et pour tout préfixe u et lettre suivante $a \in \Sigma$ on définit $Q'_{ua} := Q'_u \cup Q_{ua}$
   - L'ensemble Q''_u est l'ensemble des états où on peut aboutir après lecture d'une réparation par suppression de u. On définit $Q''_\varepsilon := Q_\varepsilon$, et pour tout préfixe u et lettre suivante $a \in \Sigma$ on définit $Q''_{ua} := Q'_{ua} \cup \delta\left(Q''_u, a\right)$, où $\delta\left(Q''_u, a\right)$ est l'ensemble des états accessibles par un état de Q''_u par une transition étiquetée a, comme dans le cas d'induction de la définition de Q_u à la question 1 .

   On peut montrer immédiatement la correction par récurrence : le seul point important est la définition inductive de Q''_{ua}, où on observe que les états accessibles par une suppression sur le préfixe ua sont exactement les états accessibles par une suppression qui se finit à la fin du préfixe ua, c'est-à-dire Q'_{ua}, et l'image après lecture de a des états accessibles par une suppression qui se finit strictement avant la fin du préfixe ua, c'est-à-dire $\delta\left(Q''_u, a\right)$.

   À la fin, pour décider l'existence d'une réparation par suppression, il suffit de vérifier si Q''_w contient un état final.

   Le calcul des ensembles a la même complexité qu'en question 1, c'est-à-dire $O(|w| \times |\delta|)$, et la complexité en espace est toujours $O(|Q|)$.

8. On modifie l'algorithme de la question 6. Au lieu de représenter Q'_u et Q''_u comme des ensembles, on va en faire des tableaux ayant pour valeur des entiers. Plus précisément, pour chaque état q de l'automate et préfixe u du mot w, l'entier $Q'_u[q]$ sera la longueur minimale du suffixe de u à supprimer pour aboutir à l'état q (ou une longueur spéciale ∞ s'il n'est pas possible du tout d'aboutir à cet état), et $Q''_u[q]$ sera la longueur minimale d'une suppression dans u pour aboutir à l'état q. À la fin, on va renvoyer le minimum de $Q''_w[q]$ pour q parcourant l'ensemble des états finaux.

   Pour le calcul de Q', on initialise $Q'_u[q] := 0$ pour les états q qui sont initiaux, et $Q'_u[q] := \infty$ sinon; et pour un préfixe u et une lettre a, on définit $Q'_{ua}[q]$ pour chaque q comme 0 si $q \in Q'_{ua}$ (on peut y aboutir par la suppression vide) et sinon comme $1 + Q'_u$ (on peut y aboutir en poursuivant la suppression précédente).

   Pour le calcul de Q'', on initialise Q'' de la même manière que Q', et pour un préfixe u et une lettre a, pour chaque état q, on définit $Q''_{ua}[q]$ comme le min de $Q''_u\left[q'\right]$ sur les q' tels qu'il y ait une transition étiquetée a de q à q' (on peut y aboutir par une suppression qui se finit plus tôt) et de $Q'_{ua}[q]$ (on peut y aboutir par une suppression qui se finit à ce moment).

   Il est clair par induction que les quantités des ensembles Q et des tableaux Q' et Q'' ont la sémantique prescrite, et ainsi l'algorithme est correct. La complexité est inchangée.

   Si on veut retrouver une suppression témoin, on modifie l'algorithme pour conserver toutes les valeurs calculées pour Q, Q', et Q'', de sorte que l'espace mémoire utilisé est à présent en $O(|Q| \times |w|)$. Une fois le calcul terminé, on choisit un état q final quelconque tel que $Q''_w[q]$ soit minimal parmi les finaux. Si $c := Q''_w[q]$ est nul alors la suppression vide est une réparation correcte. Sinon, comme les valeurs de Q'' ont été initialisées soit à 0 soit à ∞, par définition, il doit exister un préfixe x de w et un état q' tel que $c = Q'_x\left[q'\right]$ et il y a un chemin de q à q' étiqueté par le suffixe z de longueur $|w| - |x|$ de w. On considère un préfixe x de longueur maximale avec cette propriété. Par définition encore, pour y le préfixe de w de longueur $|x| - c$, on a $q \in Q_y$. Considérons à présent le mot xz. On sait que l'automate peut aboutir à l'état q en lisant x puisque $q \in Q_x$, et la définition de Q'' garantit que l'automate peut aller de q à q' en lisant z, ainsi on a bien obtenu une réparation.

9. On explique d'abord comment calculer le coût minimal sans calculer un témoin. On va suivre un algorithme dynamique. On construit un tableau $Q_u[q]$ où u parcourt les préfixes de w et q les états de l'automate, stockant le nombre minimal d'insertions à réaliser dans u pour que l'automate puisse aboutir à l'état q.

   Le cas de base est de définir $Q_\varepsilon[q] := 0$ pour tous les états initiaux q, de définir $Q_\varepsilon[q] := \infty$ pour les états inaccessibles, et pour les autres états q' on définit $Q_\varepsilon\left[q'\right]$ comme étant égal à la longueur du plus court chemin d'un état initial à q' dans

l'automate. On peut calculer cela efficacement par un BFS.

L'induction est comme suit : pour tout préfixe u de w et lettre suivante a, pour tout état q, on assigne $Q_{ua}[q] := \min_{q'} Q_u[q'] + d_a(q', q') - 1$ où $d_a(q', q)$ dénote la longueur minimale d'un chemin de q à q' dans l'automate qui commence par a. Autrement dit, le nombre minimal d'insertions pour atteindre q en lisant le préfixe ua avec des insertions est le min, sur les états q' atteints avant a avec des insertions, du nombre d'insertions nécessaires pour atteindre q' avant a plus le nombre d'insertions à effectuer, après avoir lu a depuis q', pour atteindre q.

À la fin, on renvoie le min de $Q_w[q]$ sur les états finaux q. Il est clair que cet algorithme est correct. La complexité en espace est en $O(|Q|)$ vu qu'on ne mémorise que la colonne précédente du tableau. La complexité en temps est $O(|A|+|w|\times|Q|\times|\delta|)$.

Pour calculer également un témoin, on conserve tous les tableaux (donc un espace mémoire en $O(|w| \times |Q|)$) et on stocke, dans chaque case $Q_u[q]$, un choix d'état q' qui réalise le min. On peut alors calculer un témoin en remontant depuis la fin comme à la question 7 , en calculant un plus court chemin à chaque étape.

10. Il s'agit d'une variante de la question 8 qui s'inspire de l'algorithme de Levenshtein. On modifie simplement la définition inductive de $Q_{ua}[q]$ pour prendre le min de la définition précédente et de $1 + Q_u[q]$: on peut soit aboutir à l'état q en aboutissant à un état q' avant a puis en lisant a et en effectuant un certain nombre d'insertions, ou bien en aboutissant à l'état q juste avant a et en supprimant a (inutile de considérer la possibilité d'effectuer des insertions après a dans ce cas vu qu'on peut tout aussi bien décider de les faire toutes avant de supprimer a).

Exercice 4. Sujet 0 CCP MPI

1. Rappeler la définition d'un langage régulier.
2. Les langages suivants sont-ils réguliers? Justifier.

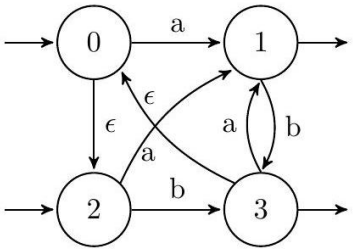
(a) $L_1 = \{a^n b a^m \mid n, m \in \mathbb{N}\}$

(b) $L_2 = \{a^n b a^m \mid n, m \in \mathbb{N}, n \leq m\}$

(c) $L_3 = \{a^n b a^m \mid n, m \in \mathbb{N}, n > m\}$

(d) $L_4 = \{a^n b a^m \mid n, m \in \mathbb{N}, n + m \equiv 0 \bmod 2\}$

3. On considère l'automate non déterministe suivant :



(a) Déterminiser cet automate.

(b) Construire une expression régulière dénotant le langage reconnu par cet automate.

(c) Décrire simplement avec des mots le langage reconnu par cet automate.