Exercice 003 *Algorithme de McNaughton-Yamada*

On utilisera dans cette partie des types et fonctions présents dans `yamada.ml`.

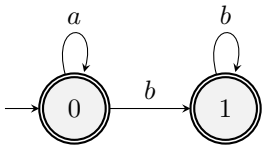
De multiples méthodes existent pour construire une expression régulière reconnaissant le même langage qu'un automate. On se propose ici d'étudier l'algorithme de McNaughton-Yamada, dont nous allons détailler le fonctionnement.

Soit $\mathcal{A} = (Q, \Sigma, I, F, \delta)$ un automate. On suppose que Q est l'ensemble $[\![0; |Q| - 1]\!]$ quitte à renommer les états. On va tenter de calculer les langages suivants :

$$\forall p, q \in Q, L_{p,q} = \left\{ u \in \Sigma^* \backslash \{\varepsilon\} \mid p \xrightarrow{u}_* q \text{ dans } \mathcal{A} \right\}$$

Autrement dit, $L_{p,q}$ est le langage des mots non-vides qui permettent de passer de l'état p à l'état q dans l'automate $\mathcal{A}$; on remarquera alors que les chemins considérés doivent être non-vides.

1. Donner sans justifier les langages $L_{p,q}$ pour $p, q \in \{0, 1\}$ pour l'automate suivant :



L'algorithme de McNaughton-Yamada réalise le calcul de ces langages par programmation dynamique, en calculant en fait des langages intermédiaires $L_{p,q,k}$ (pour $k \in [\![0; |Q|]\!]$) ; chaque $L_{p,q,k}$ est le langage des mots qui sont l'étiquette d'un chemin non vide de p à q dont tous les états intermédiaires[1] sont dans $[\![0; k-1]\!]$.

2. Que vaut $L_{p,q,0}$ dans le cas général ?

Pour réaliser le calcul des langages $L_{p,q,k}$, on va calculer, pour tout k, une matrice M_k telle que, pour deux états p, q, la case $(M_k)_{p,q}$ est une expression régulière qui engendre le langage $L_{p,q,k}$.

3. Définir un type `exp_matrix` qui représente ces matrices d'expressions régulières.
4. Implémenter une fonction `matrice_initiale : automate -> exp_matrix`, qui renvoie la matrice M_0 pour un automate donné en entrée.

On cherche maintenant à calculer M_{k+1} à partir de M_k.

5. En vous inspirant de l'algorithme de Floyd-Warshall, proposer une relation de récurrence permettant de calculer $L_{p,q,k+1}$, et la justifier rapidement.
6. Implémenter une fonction `matrice_finale : automate -> exp_matrix`, qui renvoie la matrice $M_{|Q|}$ pour un automate donné en entrée.
7. Combien d'expressions régulières calcule cette fonction au total ?

Une fois les $L_{p,q,|Q|}$ calculés, l'algorithme construit facilement une expression régulière qui engendre le même langage que $L(\mathcal{A})$.

8. Proposer une expression de $L(\mathcal{A})$ permettant de conclure. *On sera attentif au fait que les langages calculés par l'algorithme ne contiennent jamais ε.*
9. Implémenter une fonction `mcnaughton_yamada : automate -> exp_reg`, qui renvoie une expression régulière qui engendre le même langage qu'un automate donné en entrée.
10. Tester cette fonction sur l'automate donné en exemple. Que remarque-t-on sur l'expression régulière construite ?

Pour estimer correctement la complexité de l'algorithme, il faut être capable d'estimer la taille des expressions régulières qu'on va construire.

11. En supposant qu'on construise les expressions régulières de manière naïve, sans chercher à les simplifier, donner la taille maximale d'une expression régulière de M_k. On pourra écrire une relation de récurrence pour s'aider.

[1] Donc en excluant p et q.

12. En déduire la complexité de l'algorithme.

On peut améliorer son efficacité en implémentant une fonction de simplification d'expressions régulières. Il est possible de réfléchir à des règles de simplification, notamment en se basant sur les fonctions auxiliaires fournies dans `regex.ml`*. Cependant, certaines simplifications demandent d'effectuer des factorisations, et sont difficiles à implémenter directement.*