Projet – Apprentissage automatique de programmes de validation de chaînes de caractères

Octobre 2023

1 Consignes

Les projets seront faits en **groupes de 3**. Vous rendrez votre projet sur l'Université Virtuelle, dans l'espace prévu à cet effet, sous la forme d'un fichier zip qui sera intitulé :

NOM1_prénom1_NOM2_prénom2_NOM3_prénom3.zip

Ce fichier zip contiendra à la fois le code et un rapport détaillé. Le code devra être fait en langage **Python version 3** et devra être **commenté**.

Le squelette de code est disponible sur l'UV. Vous ne devez remettre que le fichier `project.py`, vous ne pouvez donc pas modifier les fichiers `utils.py` et `tests.py`. Vous n'avez pas le droit d'importer d'autres modules que (i) la bibliothèque standard ; (ii) `pysat` ; (iii) `automata`. Le rapport **doit** être écrit intégralement (pas de photo de vos formules...) avec LaTeX (vous pouvez utiliser Overleaf) ou `Typst`, veillez à bien rendre le fichier source en plus du PDF exporté.

Le rapport et le code valent chacun pour la moitié des points. La date limite de la remise finale du projet est fixée au **15 / 12 / 2023**. Chaque jour de retard entraînera une diminution de la note du projet de **2 points**.

2 Présentation du problème

Exemple concret

Partons d'un exemple concret. Supposons que Bob demande à Alice d'écrire un programme qui valide des chaînes de caractères (qu'on appellera juste chaînes par la suite) d'une certaine forme comme, par exemple, respectant le format d'une adresse e-mail. Bob ne sait pas précisément expliquer à Alice le format des chaînes que le programme doit vérifier, mais il est par contre capable de lui donner quelques exemples, *positifs* (des chaînes valides) et *négatifs* (des chaînes

invalides). Sur base de ces exemples, Alice émet une hypothèse qui *généralise* les exemples et écrit un programme qui valide les exemples positifs (et possiblement d'autres) et invalide les exemples négatifs (et possiblement d'autres). Par exemple, prenons des chaînes formées uniquement de 0 et de 1. Bob donne à Alice les deux ensembles suivants P (exemples positifs) et N (exemples négatifs) :

$$P = \{1, 01, 001, 0001\} \qquad N = \{10, 010, 00, 0010\}$$

Sur base de ces exemples, Alice écrit un programme $\mathcal{P}_{val}$ qui vérifie que la chaîne donnée en entrée est une suite de 0 (possiblement vide) qui se termine par un 1. Son programme $\mathcal{P}_{val}$ valide bien tous les exemples de P et rejette tous les exemples de N. Bien sûr, puisqu'il y a une infinité de chaînes possibles et que Bob ne donne qu'un nombre fini d'exemples, il y a une infinité de généralisations possibles, c'est-à-dire une infinité de programmes $\mathcal{P}_{val}$ possibles. Elle aurait pu inférer un autre programme $\mathcal{P}'_{val}$, qui testerait que les chaînes ne se terminent pas par 0 : il valide bien les positifs et rejette les négatifs.

Généralisation des exemples

S'il y a en général une infinité de généralisations possibles, quelle serait une bonne manière de généraliser des exemples ? Dans la littérature, il existe une multitude de notions. Une manière naturelle de définir ce qu'est une "bonne" généralisation, est d'avoir la règle la plus simple possible qui permet de discriminer les exemples négatifs des exemples positifs. Suivant cette intuition, dans ce projet, nous ferons l'hypothèse que plus un programme est court, plus il est simple : on voudrait donc un programme le plus court possible qui accepte les exemples positifs et rejette les exemples négatifs.

Automates : introduction par un exemple

Dans ce projet, nous allons considérer un modèle de programmes de validation de chaînes appelé *automates finis*. Les automates finis ont l'avantage d'avoir une syntaxe très simple, leur permettant d'être représentés graphiquement comme des graphes finis étiquetés. On peut donner le programme $\mathcal{P}_{val}$ d'Alice sous forme de l'automate fini de la Figure 1. Ce programme (cet automate), peut-être dans trois états possibles, qu'on note ici q_0, q_1 et q_2. La lecture de la chaîne commence dans l'état q_0. L'automate lit les mots de la gauche vers la droite. À chaque fois qu'un caractère est lu, l'automate suit l'arête (appelée *transition*) dont l'étiquette est la même que le caractère lu. Par exemple, en lisant la chaîne [1] 001, l'automate commence en q_0 (l'état de début est indiqué par une flèche entrante sans source), lit le premier caractère 0, reste dans l'état q_0, lit le deuxième caractère 0, reste toujours dans l'état q_0, puis lit 1 et passe dans l'état q_1. La validation du mot se fait en désignant des états spéciaux, les *états acceptants*. On les représente graphiquement par un double cercle. Dans la figure, q_0, q_2 ne sont pas acceptants et q_1 est acceptant. La lecture du mot 001

1. la terminologie usitée en théorie des automates est *mot* et non chaîne.

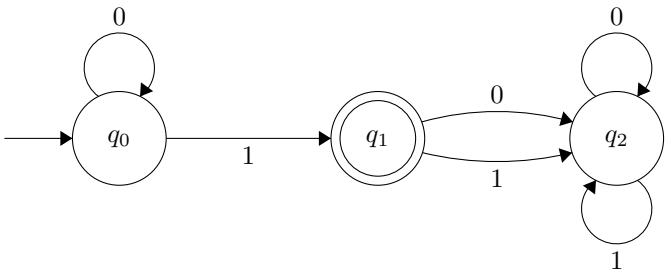

FIGURE 1 – L'ensemble des mots acceptés sont ceux qui consistent en une séquence de 0 suivie d'un seul 1.

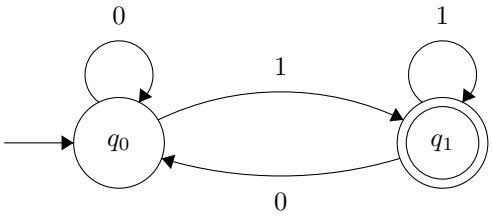

FIGURE 2 – L'ensembles des mots acceptés sont ceux qui se terminent par 1.

se terminant dans l'état acceptant q_1, le mot est dit *accepté* par l'automate. La lecture du mot 0010 se termine dans l'état q_2, qui n'est pas acceptant : le mot 0010 est rejeté.

Un autre automate qui généralise correctement les exemples est celui de la Figure 2 : il accepte tous les mots qui se terminent par 1 et rejettent tous les autres. Cet automate ne nécessite que deux états, sa définition est plus simple, il est donc préférable à l'automate de la Figure 1, si on a pour objectif de générer des *petits* programmes. De fait, la définition informelle de l'automate de la Figure 2 (accepter tous les mots qui se terminent par 1) est plus simple que celle de l'automate de la Figure 1 (accepter tous les mots qui se terminent par 1 et qui ne contiennent qu'un seul 1).

Définitions formelles

Dans cette sous-section, nous donnons les définitions mathématiques des notions d'alphabet, mots sur un alphabet, langage et automates. Un alphabet Σ est un ensemble fini. On notera souvent ses éléments $a, b, c, 0, 1, \ldots$ (qu'on appelle *lettres*). Un mot sur un alphabet Σ est une suite finie, possiblement vide, de lettres de Σ. On note ϵ la suite vide, qu'on appelle *mot vide* (si vous pensez à une liste, cela correspond à la liste vide). Par exemple, prenons l'alphabet $\Sigma_b = \{0, 1\}$. Les ensembles P et N précédents sont des ensembles de mots sur l'alphabet Σ_b. On notera Σ^* l'ensemble des mots sur un alphabet Σ.

Un *langage* sur un alphabet Σ est un sous-ensemble de mots de Σ, potentiellement infini. Par exemple, l'ensemble des mots qui se terminent par 1.

Un automate fini sur un alphabet Σ est un quadruplet $A = (Q, q_0, F, \delta)$ où :

— Q est un ensemble fini d'éléments appelés *états* ;

— $q_0 \in Q$ est un élément de Q appelé *état initial* ;

— $F \subseteq Q$ est un sous-ensemble d'états appelés *états acceptants* ;

— $\delta : Q \times \Sigma \to Q$ est une fonction partielle appelée *fonction de transition.*

Par exemple, dans la Figure 1, l'ensemble des états est $Q = \{q_0, q_1, q_2\}$, représentés par des noeuds du graphe, l'état initial est q_0 (représenté par le fait qu'il a une arête entrante sans source), et il y a un seul état final ($F = \{q_1\}$). Il y a six transitions : par exemple, de l'état q_0 vers lui-même en lisant 0, de q_0 vers q_1 en lisant 1, de q_1 vers q_2 en lisant 0 ou 1, etc.

Une *exécution* de A est une suite finie $e = p_0\sigma_1 p_1 \sigma_2 \dots p_{n-1}\sigma_n p_n$ ($n \geq 0$) telle que :

— pour tout $i \in \{0, \dots, n\}$, $p_i \in Q$

— $p_0 = q_0$

— pour tout $i \in \{1, \dots, n\}$, $\sigma_i \in \Sigma$

— pour tout $i \in \{0, \dots, n-1\}$, $\delta(p_i, \sigma_{i+1})$ est définie et vaut p_{i+1}.

On dit que l'exécution e est une exécution sur le mot $\sigma_1 \dots \sigma_n$, et que e est *acceptante* si l'état final après avoir lu $\sigma_1 \dots \sigma_n$ est acceptant, c'est-à-dire $p_n \in F$. Un mot est *accepté* par A s'il existe une exécution acceptante de l'automate sur ce mot, sinon il est rejeté. L'ensemble des mots acceptés par l'automate A est appelé *langage accepté par* A, noté $L(A)$:

$$L(A) = \{w \in \Sigma^* \mid \text{ il existe une exécution acceptante de } A \text{ sur } w\}$$

Remarquez qu'il n'existe pas nécessairement une exécution pour tout mot (même non-acceptante) car la fonction de transition n'est pas nécessairement définie partout. Par contre, si elle l'est (i.e. la fonction δ est totale), alors sur tout mot il existe exactement une exécution de l'automate. On dira que l'automate A est *complet* si δ est une fonction totale, c'est-à-dire qu'elle est définie pour toute paire $(q, \sigma) \in Q \times \Sigma$. L'automate de la Figure 1 est complet. Il est également équivalent [2] à l'automate de la Figure 3 qui lui n'est pas complet.

2. deux automates sont équivalents s'ils acceptent exactement les mêmes mots (et par conséquence rejettent les mêmes mots)

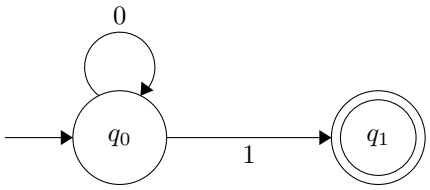

FIGURE 3 – Un automate incomplet équivalent à l'automate de la Figure 1 (les ensembles de mots qu'ils acceptent sont les mêmes, c'est-à-dire qu'ils acceptent le même langage)

3 Questions

Question 1 (5 points) Étant donné un alphabet Σ, deux ensembles finis P, N de mots sur Σ, et un entier naturel $k > 0$, donner une formule de la logique propositionnelle ϕ_{AUT} en FNC (fonction de Σ, P, N et k), qui est satisfaisable si et seulement si il existe un automate fini A tel que A possède au plus k états, et A est *consistant* avec P et N (c'est-à-dire qu'on a $P \subseteq L(A)$ et $L(A) \cap N = \emptyset$).

Donner la formule et expliquer sa construction dans le rapport. Idée : utiliser des variables booléennes pour coder les transitions et les états acceptants, ainsi que pour coder les exécutions sur les exemples positifs et négatifs.

Question 2 (5 points) À partir de votre réponse à la question précédente, implémenter un programme, en utilisant la librairie PySAT (voir cours sur le problème SAT), qui étant donnés Σ, P, N, k :

1. construit un ensemble de clauses ;
2. appelle un solveur SAT dessus ;
3. retourne (s'il existe) un automate fini sur Σ avec au plus k états, qui est consistant avec P et N ou qui retourne `None` s'il n'en existe pas.

Vous devez implémenter la fonction `gen_aut` *du squelette de code. Pour l'automate, vous devez utiliser le format indiqué dans le squelette de code (votre sortie doit être une instance de* `automata.fa.dfa.DFA`*), et tester votre implémentation avec la fonction de test* `test_aut`*. Cette fonction teste, sur un ensemble d'instances données, que votre code retourne bien* `None` *s'il n'y a pas de solution et, sinon, que votre automate a bien au plus k états, qu'il est complet et qu'il est consistant avec les exemples.*

Question 3 (2 points) En déduire un algorithme qui, étant donné Σ, P et N, retourne un automate fini sur Σ, minimal en nombre d'états qui est consistant avec P et N.

Expliquer brièvement comment faire dans le rapport et implémenter votre solution dans la fonction `gen_minaut` *à tester avec la fonction* `test_minaut`.

Question 4 (1 point) Même question que 1 et 2 mais cette fois on demande que l'automate retourné, s'il existe, soit complet.

Vous devez implémenter la fonction `gen_autc` *à tester avec la fonction* `test_autc`. *Expliquer dans le rapport comment vous vous y prenez.*

Question 5 (1 point) Même question que 1 et 2, mais cette fois on demande que l'automate retourné, s'il existe, soit un *automate réversible* : c'est-à-dire que sa fonction de transition reste une fonction si on inverse les flèches. Mathématiquement, cela veut dire que pour tout état q et lettre σ, il existe au plus un état q' tel que $\delta(q', \sigma) = q$. Un tel automate a pour propriété qu'on peut "rembobiner" ses exécutions, c'est-à-dire revenir en arrière de manière déterministe dans ses exécutions.

Vous devez implémenter la fonction `gen_autr` *à tester avec la fonction* `test_autr`. *Expliquer dans le rapport comment vous vous y prenez.*

Question 6 (2 points) Même question que 1 et 2, mais on se donne ici un entier supplémentaire $\ell \in \mathbb{N}$, avec la contrainte que l'automate fini complet retourné doit avoir au plus ℓ états acceptants.

Vous devez implémenter la fonction `gen_autcard` *à tester avec la fonction* `test_autcard`. *Expliquer dans le rapport comment vous vous y prenez. Astuce : vous pouvez utiliser des méthodes de la classe* `pysat.card.CardEnc` *ou la classe* `CNFPlus` *qui permet d'ajouter directement des contraintes de cardinalité (à utiliser uniquement avec le solveur* `MiniCard`*).*

Question 7 (4 points) On pose maintenant la même question que 1 et 2, mais où on remplace *automate fini* par *automate non-déterministe*. Un automate non-déterministe est défini comme un automate fini, à l'exception que les transitions entre états sont non-déterministes : à partir d'un état q et d'une lettre σ, on peut aller vers plusieurs états. On ne parlera donc pas de fonction de transition, mais de relation de transition.

Formellement, un automate non-déterministe sur un alphabet Σ est un quadruplet $A = (Q, q_0, F, \Delta)$ tel que :

— Q est un ensemble d'éléments appelés *états* ;
— $q_0 \in Q$ est l'état initial ;
— $F \subseteq Q$ est un sous-ensemble d'états, appelés états *acceptants* ;
— $\Delta \subseteq Q \times \Sigma \times Q$ est une relation appelée *relation de transition*.

Une exécution de A est une suite finie $e = p_0\sigma_1 p_1 \sigma_2 \dots p_{n-1}\sigma_n p_n$ $(n \geq 0)$ telle que :

— pour tout $i \in \{0, \dots, n\}$, $p_i \in Q$
— $p_0 = Q_0$
— pour tout $i \in \{1, \dots, n\}$, $\sigma_i \in \Sigma$
— pour tout $i \in \{0, \dots, n-1\}$, $(p_i, \sigma_{i+1}, p_{i+1}) \in \Delta$.

Elle est acceptante si $p_n \in F$. Un mot $u = \sigma_1 \dots \sigma_n$ est accepté par A s'il existe une exécution acceptante de A qui lit u. Notez que comme Δ est une relation, il peut exister plusieurs exécutions sur un même mot, certaines étant acceptantes, certaines non. Il suffit qu'au moins une soit acceptante pour que la mot soit accepté. À l'inverse, un mot n'est pas accepté si et seulement si aucune exécution sur ce mot n'est acceptante.

Prenons un exemple : on veut un automate dont le langage est l'ensemble des mots de longueur 3 au moins sur l'alphabet $\{\#, a, b\}$, qui commencent par # (qui doit apparaître une seule fois), et dont la dernière lettre apparaît au moins une deuxième fois après le #. Par exemple, *#aba*, *#bb*, *#aaaaaabb*, mais pas *#aaaaaaab*, ni #, ni *#a*.

On peut faire un automate non-déterministe qui accepte ce langage, donné en Figure 4. Comme il lit les mots de la gauche vers la droite, il ne peut pas savoir à l'avance quelle est la dernière lettre. Il va utiliser le non-déterminisme pour "deviner" quelle est la dernière lettre, et vérifier que ce qu'il a deviné est

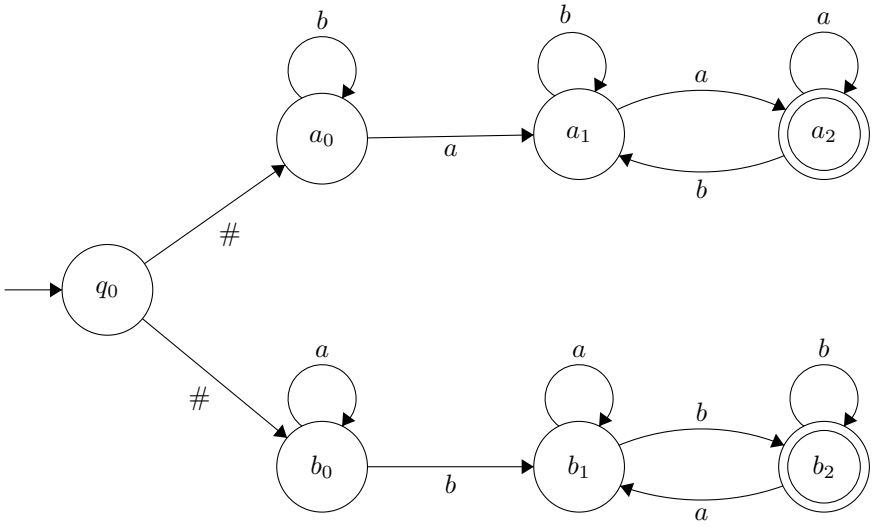

FIGURE 4 – Un automate non-déterministe qui accepte l'ensemble qui commencent par un unique #, de longueur au moins 3, et dont la dernière lettre apparaît au moins une deuxième fois. Notez le choix non-déterministe depuis l'état initial.

correct. Cela se traduit par l'existence de deux transitions lisant # à partir de l'état initial, vers a_0 et b_0. A partir de a_0, le comportement de l'automate est complètement déterministe : il vérifie que a est bien la dernière lettre, et qu'elle apparaît une deuxième fois (idem à partir de b_0 mais avec la lettre b). Le non-déterminisme n'apparaît qu'au début.

Prenons maintenant la partie supérieure (celle qui correspond à l'état initial a_0). L'automate retient qu'il a vu au moins un a (état a_1), et il n'accepte que si la dernière lettre lue est un a. En lisant le mot $\#abbaba$, il existe bien une exécution acceptante : l'exécution $q_0\#a_0aa_1ba_1ba_1aa_2ba_1aa_2$. En lisant le mot $\#aaab$, aucune exécution n'est acceptante : dans la partie supérieure, on a l'exécution $q_0\#a_0aa_1aa_2aa_2ba_1$ qui se termine par a_1 qui n'est pas acceptant, et dans la partie inférieure, on a l'exécution $q_0\#b_0ab_0ab_0ab_0bb_1$ qui se termine par b_1 qui n'est pas acceptant.

En utilisant le non-déterminisme, on aurait pu avoir moins de transitions, en prenant l'automate de la Figure 5 qui est équivalent à celui de l'automate de la Figure 4. Dans l'état a_1, on sait qu'on a lu au moins un a. En lisant encore des a à partir de l'état a_1, l'automate a le choix de rester en a_1 ou d'aller en a_2. S'il lit le dernier a et reste dans a_1, l'exécution ne sera pas acceptante. Par contre, s'il va dans l'état a_2, l'exécution sera acceptante. À l'inverse, si il lit un a qui n'est pas la dernier et va dans l'état a_2, alors l'exécution s'arrête et n'est donc pas acceptante.

L'ajout du non-déterminisme permet d'avoir des automates plus petits. Un

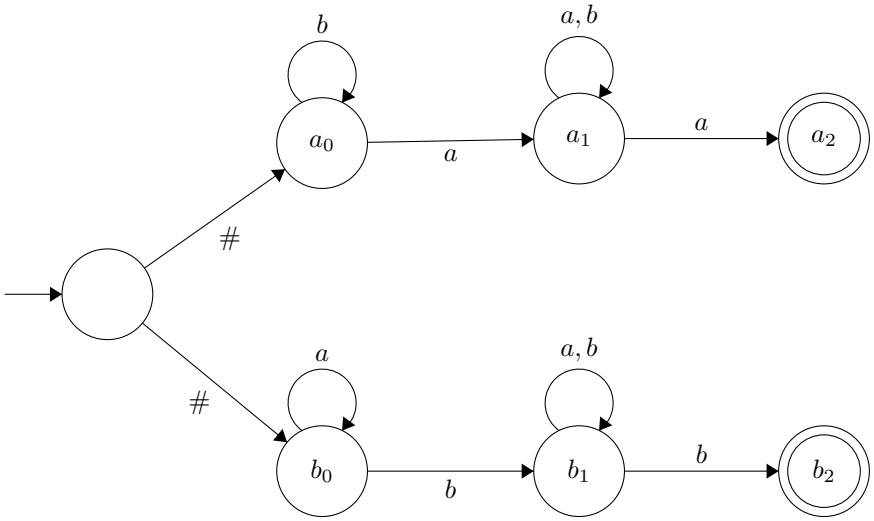

FIGURE 5 – Un automate non-déterministe équivalent à celui de la Figure 4.

automate fini minimal en nombre d'états et déterministe comme à la question 1, équivalent aux deux derniers, est donné en Figure 6.

Vous devez implémenter la fonction `gen_autn` *à tester avec la fonction* `test_autn`. *Expliquer dans le rapport comment vous vous y prenez.*

Question 8 (Bonus - 1 point) Dans ce projet, jusqu'à maintenant, nous avons pris en compte uniquement le nombre d'états et pas le nombre de transitions. On pourrait vouloir un automate qui a au plus k états et au plus k' transitions. Expliquer (sans implémentation) comment vous feriez pour prendre en compte ce nouveau paramètre.

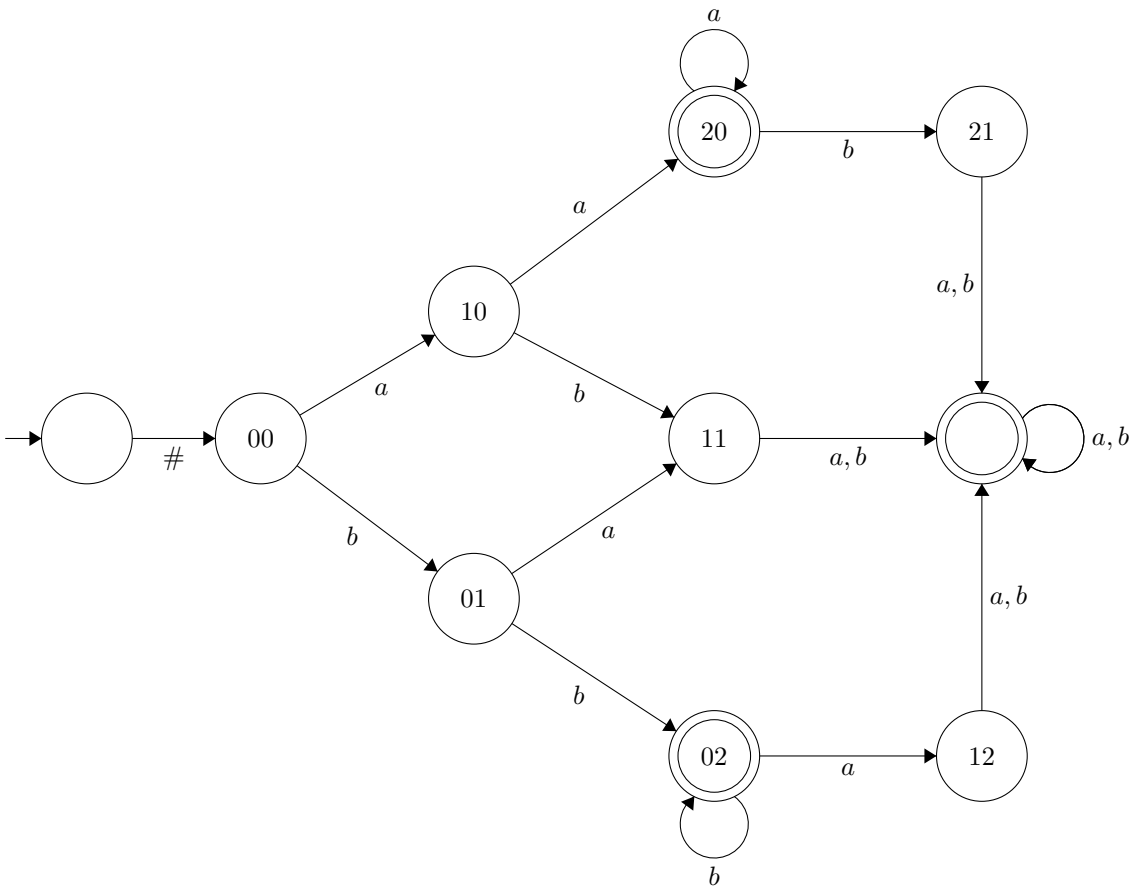

FIGURE 6 – Un automate minimal fini (déterministe) équivalent à celui de la Figure 4.