TP n°3

Implémentation de quelques algorithmes sur les automates

1 Présentation

Nous utiliserons dans ce TP des classes `Automate` et `Etat` semblables à celles du TP2, avec quelques modifications : on manipulera principalement les états via leur `id`, et non plus directement en tant qu'objet. L'`id` d'un état est un entier entre 0 et $n-1$, pour un automate à n états. Les classes `Automate` et `Etat` sont déjà implémentées dans `Automate.java` et `Etat.java`. Il n'est normalement pas nécessaire de les modifier pour ce TP : on peut faire tout le TP en utilisant la classe `Automate` comme « boîte noire », en utilisant ses méthodes.

1.1 La classe Automate

Voici un aperçu des méthodes de la classe `Automate` :

- le constructeur `Automate(int n)` crée un automate avec n états, numérotés de 0 à $n-1$.
- `getEtats()` (*resp.* `getEtat(int i)`) renvoie un tableau contenant tous les états de l'automate (*resp.* l'état dont l'`id` vaut `i`).
  `nbEtats()` renvoie le nombre d'états de l'automate, c'est-à-dire la taille du tableau renvoyé par `getEtats()`.
- `setInitial(int i, boolean initial)` et `setAcceptant(int i, boolean acceptant)` permettent de rendre l'état `i` initial/non-initial (ou acceptant/non-acceptant), en fonction de la valeur du paramètre booléen.
- `toString()` convertit en chaîne de caractères et permet d'utiliser `System.out.println` sur un automate.
- `ajouteTransition(int idDebut, int idFin, char c)` ajoute une transition étiquetée par `c` entre l'état `idDebut` et l'état `idFin`.
- `nombreTransitions(), alphabet(), estComplet(), estDeterministe()` sont des fonctions similaires à celles implémentées pendant le TP2.
- `successeurEnsemble(Set<Integer> etatsActuels, char c)` renvoie l'ensemble de tous les états `q` pour lesquels il existe un état `p` dans `etatsActuels` avec une transition étiquetée par `c` de `p` à `q`. Il s'agit de l'algorithme de transition non-déterministe entre ensembles d'états.
- `accepte(String mot)` renvoie `true` si et seulement si le mot appartient au langage de l'automate (cette fonction utilise `successeurEnsemble`).

1.2 Exemples dans la classe Main

La classe `Main` contient deux exemples :

- un exemple de création d'automate, avec ajout de transitions et définition des états initiaux et acceptants, dans la fonction `exemple`,

— un exemple de manipulation d'automate : la fonction `remplaceAParC` prend en paramètre un automate, et renvoie un nouvel automate qui a la même structure, mais les transitions étiquetées par des « a » deviennent étiquetées par des « c » (voir la Figure 1 pour un exemple). Un exemple d'utilisation de cette fonction est donné dans la fonction `exemple`.

FIGURE 1 – Illustration de la fonction `remplaceAParC` sur l'automate de a^*b^+.

2 Algorithmes à implémenter

Pour ce TP, on demande d'implémenter quelques algorithmes qui manipulent des automates, à la manière de `remplaceAParC`. Le squelette du code est fourni dans `Main.java`. Ne pas hésiter à créer des fonctions auxiliaires si nécessaire. Il faudra régulièrement tester les fonctions implémentées avec quelques exemples d'automates et de mots !

2.1 Manipulations basiques

Implémenter les algorithmes suivantes :

1. `public static Automate automateMiroir(Automate a)` qui calcule un nouvel automate `b` qui est l'automate miroir de `a`, *i.e.* si `a` reconnaît *abc*, `b` doit reconnaître *cba*, etc. Pour cela, il faut inverser toutes les transitions, rendre les états initiaux acceptants, et vice versa.
2. `public static Automate automateComplete(Automate a)` qui renvoie un nouvel automate qui est le « complété » de `a`. Pour cela, tester s'il est complet, et s'il ne l'est pas, créer un automate avec un état de plus, qui sera le puits.

Tester les fonctions sur quelques automates et quelques mots.

2.2 Déterminisation et complémentaire

On veut maintenant implémenter l'algorithme qui permet d'obtenir le complémentaire de `a`. On rappelle que pour cela, il faut d'abord créer un automate déterministe équivalent à `a`, puis le compléter, et enfin inverser les états acceptants et non-acceptants.

2.2.1 Déterminisation

L'algorithme vu en cours pour déterminiser `a` consiste à construire un automate déterministe `b` dont les états sont étiquetés par des ensembles d'états de `a`. Ici, nos états sont étiquetés par des entiers. Il faut donc trouver une méthode pour faire correspondre des entiers à des ensembles d'états, et réciproquement. Une solution est de passer par la représentation binaire : un `1` à la i-ième position signifie que l'état i est dans l'ensemble. Attention, le bit à la position 0 est le bit le plus *à droite* dans la représentation binaire. Par exemple, le nombre `0b0001101` représente l'ensemble d'états $\{0, 2, 3\}$. Inversement, l'ensemble d'états $\{1, 4\}$ correspond au nombre `0b00010010`.

Écrire deux fonctions auxiliaires qui réalisent les opérations décrites ci-dessus :

1. `public static int ensembleVersEntier(Set<Integer> s)`, qui prend un ensemble d'entiers $\{a_i\}$ et qui renvoie l'entier $n = \sum_i 2^{a_i}$,
2. sa réciproque, `public static Set<Integer> entierVersEnsemble(int i)`, qui prend un entier et renvoie l'ensemble d'entiers correspondant.

Utiliser ces deux fonctions pour écrire la fonction `public static Automate automateDeterministe (Automate a)` qui renvoie un nouvel automate déterministe équivalent à `a`.
Tester les fonctions sur quelques automates et quelques mots.

Bonus : (revenir à cette question après avoir fini les autres !) on peut remarquer que l'automate ainsi construit contient beaucoup d'états non-accessibles ou non-co-accessibles. Implémenter une fonction `automateEmonde` qui renvoie une version émondée (c'est-à-dire sans les états non-accessibles ou non-co-accessibles) de l'automate passé en paramètre.

2.2.2 Complémentaire

Implémenter la fonction `public static Automate automateComplementaire(Automate a)` qui calcule un nouvel automate qui est l'automate complémentaire de `a`. Utiliser les méthodes `automateDeterministe` et `automateComplete` écrites précédemment.
Tester la fonction sur quelques automates et quelques mots.

2.3 Automate produit

On veut maintenant construire l'automate produit de deux automates complets, qui permet de construire un automate pour l'union et l'intersection, comme vu au TD 3.

1. Premièrement, implémenter la fonction `automateProduit(Automate a1, Automate a2)`, qui prend deux automates `a1` avec n états et `a2` avec m états, et qui renvoie l'automate produit de `a1` et `a2` avec $n \times m$ états.
   Dans cet automate, les états correspondent à des couples (i, j) d'états de `a1` et `a2`, et il y a une transition $(i, j) \xrightarrow{a} (i', j')$ si $i \xrightarrow{a} i'$ dans `a1` **et** $j \xrightarrow{a} j'$ dans `a2`.
   On peut encoder l'état (i, j) par l'entier $i \times m + j$. Ne pas se préoccuper des états initiaux et acceptants pour l'instant.
2. Utiliser la fonction `automateProduit(Automate a1, Automate a2)` pour implémenter la fonction `automateIntersection(Automate a1, Automate a2)`, qui construit un automate qui accepte les mots acceptés par `a1` et par `a2`. Il faut s'assurer que les automates sont complets (mais pas nécessairement déterministes), et bien choisir les états initiaux et finaux.

Tester les fonctions sur quelques automates et quelques mots.