TP n° 6 : Opérations sur les automates

Vous trouverez sur ma page Web (`http://www.pps.jussieu.fr/~delatail/AF3.html`) une correction des premiers TPs sur les automates :

- le fichier « `correctionAD.java` » donne la fonction de reconnaissance et construit deux automates sur lesquels on teste la fonction de reconnaissance. *Vous pouvez récupérer ce code, mais il est indispensable que vous ayez compris le fonctionnement de la fonction de reconnaissance et la construction des automates donnés en exemple, sans quoi la suite des TPs ne sera qu'un long et douloureux calvaire.*
- le fichier « `correctionLE.java` » donne les algorithmes de lecture et écriture dans un fichier. Vous pouvez récupérer ce code (il est préférable d'avoir compris l'idée générale de ces fonctions, mais il n'est pas indispensable d'en comprendre tous les détails).

1 Opérations sur les automates déterministes

Dans cette section, les fonctions à définir supposent que l'on génère un nouvel automate à partir de l'automate courant (celui sur lequel la méthode est appelée). Cela implique donc que l'on recrée entièrement la structure de l'automate (tous les états et toutes les transitions sont réalloués, afin de ne pas modifier l'automate originel).

La façon la plus naturelle de procéder est d'écrire une méthode qui crée une copie de l'automate. Elle ressemblera, comme on peut s'en douter, à votre constructeur lisant dans un fichier, à ceci près qu'il suffit de lire les informations directement dans l'automate existant.

Mais vous pouvez aussi "tricher" en utilisant les fonctions d'écriture/lecture dans un fichier : vous écrivez l'automate dans un fichier "poubelle", et vous le lisez immédiatement après. Cette façon de faire n'est pas très propre, mais vous dispense de réécrire une fonction de copie.

Exercice 1 : Standardisation

Écrivez une méthode « `Automate standardise()` » qui renvoie l'automate standardisé.

Exercice 2 : Complété

Définissez une classe « `Alphabet` » qui contient un tableau des lettres correspondant à l'alphabet considéré.

Ajoutez une méthode « `boolean estComplet(Alphabet abc)` » à votre classe `Automate`, puis une méthode « `Automate complete(Alphabet abc)` » qui calcule le complété par rapport à l'alphabet `abc`.

Exercice 3 : Complémentaire

Ajoutez à votre classe `Automate` une méthode « `static Automate complementaire(Automate A, Alphabet abc)` » qui construit un automate reconnaissant le complémentaire du langage reconnu par l'automate `A` par rapport à l'alphabet `abc`.

Exercice 4 : Automate produit

Écrivez des méthodes « `static Automate unionLangages(Automate A, Automate B)` » et « `static Automate intersectionLangages(Automate A, Automate B)` » qui calculent les automates correspondant à l'union et l'intersection.

2 Reconnaissance non déterministe

La structure de données que nous utilisons pour représenter un automate permet d'implémenter aussi bien des automates non déterministes (avec plusieurs états initiaux et plusieurs transitions portant la même lettre et partant d'un même état) que déterministes ; par contre, notre fonction de reconnaissance ne marche que pour les automates déterministes. L'objectif de cette section est d'implémenter la reconnaissance d'un mot par un automate non déterministe. Il est bien sûr indispensable d'avoir bien compris la fonction de reconnaissance déterministe avant de passer à sa version non déterministe.

Exercice 5 :

Dans votre classe `Automate`, ajoutez une méthode « `boolean reconnAux(Etat q, String s)` » qui reconnaît *de manière récursive* si un mot `s` peut être lu à partir de l'état `q`.

Exercice 6 :

Écrivez une méthode « `boolean reconnaissanceND (String s)` » qui implémente la reconnaissance dans un automate non déterministe.

Exercice 7 :

Testez vos méthodes sur les exemples suivants :

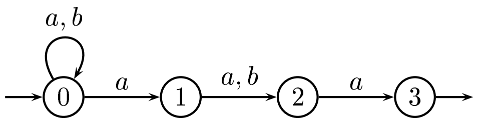

FIG. 1 – Automate $\mathcal{A}_1$

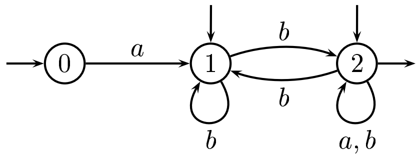

FIG. 2 – Automate $\mathcal{A}_2$

On implémentera le premier exemple "à la main", et le deuxième exemple à partir d'un fichier.

Exercice 8 : Miroir

Écrivez une méthode « `Automate miroir()` » qui renvoie l'automate miroir. Testez cette méthode sur l'exemple $\mathcal{A}_1$.

3 Déterminisation

Pour implémenter la reconnaissance non déterministe, on propose un autre algorithme, similaire au cas déterministe sauf que l'on navigue sur une liste d'états au lieu d'un seul état. On introduira donc une nouvelle classe « `ListeEtats` » qui implémente une liste d'états.

Exercice 9 :

Dans votre classe `Automate`, ajoutez une méthode qui retourne la liste des états initiaux.

Exercice 10 :

Dans votre classe `Automate`, ajoutez une méthode « `boolean reconnaissanceNDbis (String s)` » qui implémente le nouvel algorithme de reconnaissance non déterministe.

L'algorithme de déterminisation fonctionne sur la même idée, mais cette fois-ci il faut être capable de dire si une liste d'états a déjà été atteinte. On construira donc une fonction « `static` `boolean compare(ListeEtats l1, ListeEtats l2)` » qui dit si les deux listes `l1` et `l2` contiennent les mêmes états. On pourra aussi ajouter à la classe `Automate` une fonction « `Alphabet alph()` » qui génère l'alphabet des lettres utilisées par l'automate.

Exercice 11 :

Dans votre classe `Etat`, ajoutez une méthode « `ListeEtats cibleND(char c)` » qui retourne la liste des états accessibles par une transition portant la lettre `c`.

Exercice 12 : Déterminisation

Dans votre classe `Automate`, ajoutez une méthode « `Automate determinise()` » qui retourne l'automate déterminisé. On pourra procéder en deux passes : une pour déterminer le nombre d'états du nouvel automate, une autre pour générer les transitions proprement dites.