TP n°2

Automates : reconnaissance et déterminisation en Java

Le but de ce TP est de manipuler des automates finis en Java. En particulier, on programmera l'acceptation d'un mot par un automate non déterministe, et l'algorithme de déterminisation des automates finis non déterministes. Des classes Java à compléter sont fournies sur Moodle.

Attention : pour une meilleure compréhension et pour plus de facilité, contrairement à la définition générale, les automates non déterministes codés ici ont un seul état initial.

Le code fourni

Le code est réparti dans trois fichiers.

— `Etat.java` fournit la classe `Etat` possédant :
  — un identifiant `int id`,
  — un booléen `boolean etatAcc` pour spécifier si l'état est acceptant,
  — toutes les transitions sortantes `Map<Character, Set<Etat>> transitions` associant à chaque lettre l'ensemble des états accessibles en lisant cette lettre,

  et des méthodes déjà programmées ;

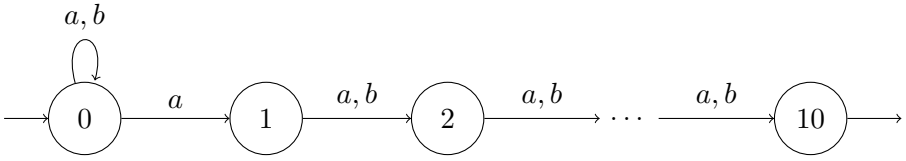
— `Automate.java` fournit la classe `Automate` qui étend `HashSet<Etat>` (ensemble d'états) en précisant notamment un état initial, et qui construit grâce à la méthode `boolean initialiseAutomate(Etat e)` l'ensemble des états accessibles à partir de l'état initial ;

— `Main.java` pour définir des automates et tester votre code.

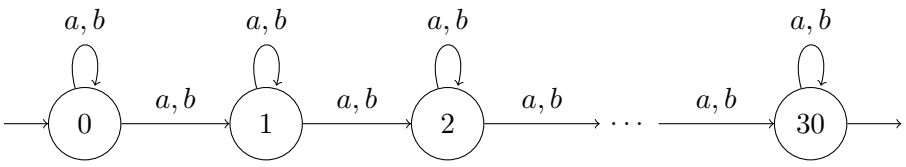
Lisez attentivement ce code, faites des essais pour bien comprendre son fonctionnement.

Automates et acceptation

1. Dans `Main.java`, définissez deux automates non déterministes :
   — $\mathcal{A}_1$ a 11 états et accepte les mots dont la 10^{e} lettre en partant de la fin est a :



   — $\mathcal{A}_2$ a 31 états et accepte tous les mots de longueur ≥ 30 :



2. Programmer une fonction dans `Automate.java` qui renvoie le nombre de transitions de l'automate. Tester la fonction sur les automates $\mathcal{A}_1$ et $\mathcal{A}_2$.
3. Programmer une fonction dans `Automate.java` qui renvoie l'alphabet de l'automate (c'est à dire l'ensemble des lettres utilisées). Tester la fonction sur les automates $\mathcal{A}_1$ et $\mathcal{A}_2$.

4. Programmer une fonction dans `Automate.java` qui renvoie `true` si l'automate est déterministe (c'est-à-dire s'il a au plus une transition sortante pour chaque état et chaque lettre). Tester la fonction sur les automates $\mathcal{A}_1$ et $\mathcal{A}_2$.
5. Un algorithme d'acceptation `boolean accepte(String mot)` est fourni dans `Automate.java`. Le tester sur les automates $\mathcal{A}_1$ et $\mathcal{A}_2$ avec le mot a^{30}. Qu'observe-t-on ? Pourquoi ?
6. Programmer un algorithme d'acceptation plus efficace qui maintient à chaque étape l'ensemble des états accessibles depuis l'état initial :
   — entrée : un automate non déterministe $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ et un mot $u = u_1 \dots u_n$;
   — $S \leftarrow I$ (seuls états accessibles sans lire de lettre) ;
   — pour i de 1 à n faire :
      — $S \leftarrow \bigcup_{q \in S} \delta(q, u_i)$;
   — accepter si et seulement si $S \cap F \neq \emptyset$.
   Tester votre algorithme sur les automates $\mathcal{A}_1$ et $\mathcal{A}_2$ avec le mot a^{30}.
7. Programmer une fonction dans `Automate.java` pour enlever les états non co-accessibles, c'est-à-dire qui ne permettent pas d'arriver à un état acceptant.

Déterminisation (plus difficile)

Programmer l'algorithme de déterminisation par sous-ensembles vu en cours via une méthode `Automate determinisation()` qui renvoie un automate déterministe (c'est-à-dire ayant au plus une transition sortante pour chaque état et chaque lettre).

On rappelle que les états de l'automate $\mathcal{A}'$ obtenu par la déterminisation d'un automate non déterministe $\mathcal{A}$ sont habituellement étiquetés par des ensembles d'états de $\mathcal{A}$. On pourra donc ajouter à la classe `Etat` un ensemble d'entiers afin d'identifier, pour chaque nouvel état de $\mathcal{A}'$, l'ensemble d'états de $\mathcal{A}$ correspondant.

Annexe : rappels Java

Set et Hashset

`Set<E>` est une interface représentant un ensemble d'objets de type `E`. Une implémentation possible est `HashSet<E>`. On pourra donc écrire `Set<Integer> s = new HashSet<Integer>();`

Les méthodes les plus courantes pour cette classe sont :

- `boolean contains(E e)` ;
- `boolean isEmpty()` ;
- `int size()` ;
- `boolean add(E e)` qui tente d'ajouter l'élément à l'ensemble, renvoie `false` s'il est déjà présent, et `true` sinon ;
- `boolean remove(Object o)` ;
- `boolean addAll(Set<E> s)` qui rajoute à l'ensemble courant tous les éléments d'un ensemble `s` ;
- `boolean equals(set<E> s)` qui teste si l'ensemble courant est égal à l'ensemble `s`.

A noter qu'il est possible de parcourir les éléments d'un `Set<E> s` à l'aide d'une boucle comme celle-ci :

```
for(E e : s){...}
```

qui peut être vue comme un équivalent pour un tableau `E[] t` de

```
for(int i = 0; i < t.length ; i++){E e = t[i]; ....}
```

Map et HashMap

L'interface `Map<E,F>` permet de représenter des fonctions sur un ensemble `E` dans un ensemble `F`. Une implémentation pratique est `HashMap(<E,F>)`. L'ensemble de départ `E` est appelé ensemble de clés et l'ensemble `F` ensemble de valeurs. Les associations clés-valeurs définissant la fonction sont stockées sous forme de paires. On récupère la valeur associée à une clé par la méthode `F get(E e)` (qui renvoie `null` si aucune valeur n'est associée à cette clé) et on ajoute une nouvelle association clé-valeur à l'aide de la méthode `put(E e, F f)`. Il existe d'autres méthodes pratiques, voir la documentation.