Théorie des langages & Automates

Université de Tours – Département informatique de Blois

TP1 - Automates à états finis

*
* *

Contexte

Le but de ce TP est d'implémenter la structure formelle d'automates à états finis telle que vue dans le Cours 1, notamment des AFD, AFND et AFNDε.
Pour ce faire, il est vous est demandé de récupérer l'archive **Automate.zip** contenant une première implémentation à compléter. Ce TP est à réaliser en Java, en monôme ou binôme, et devra s'appuyer sur les classes existantes.
Le TP a une durée effective de 3h et devra être rendu pour évaluation au plus tard le 27 Octobre 2022 avant 23h55.

Contenu du projet Automate

Une première structure a déjà été implémentée. On donne la description succincte des classes suivantes :

- `State.java`

  Représente un état d'automate.

  Elle est composée d'un unique attribut `name (String)`.

  La classe contient les fonctions basiques de type `getters, toString, equals` et `hashCode`.

- `Symbol.java`

  Représente un symbole lu par un automate.

  Elle est composée d'un unique attribut `symbol (String)`.

  La classe contient les fonctions basiques de type `getters, toString, equals` et `hashCode`.

- `FSM.java`

  Classe abstraite représentant les éléments de base d'un automate à états finis (i.e. Finite State Machine).

  Elle est composée des attributs `states (Set<State>)`, `alphabet (Set<Symbol>)` et `ends (Set<State>)` représentant respectivement les ensembles Q, Σ et F dans le cours[1].

  La classe contient deux contructeurs :

[1] Les variables d'état initial (resp. s et S) et fonction de transition (δ et Δ) étant différentes pour les AFD et AFND, elles seront implémentées dans leur classe respective.

- Un contructeur par défaut qui prend en argument l'ensemble des attributs listés et,
- Un constructeur qui prend en argument un chemin vers un fichier .json `path (String)` représentant l'automate. La structure du fichier est détaillée dans la section suivante.

- `Transition.java`

  Représente l'argument de la fonction de transition dans un automate à états finis.

  Elle est composée des deux attributs : `p (T)` et `a (Symbol)`.

  La classe contient les fonctions basiques de type `getters, toString, equals` et `hashCode`.

  De façon générale, on supposera que le type générique `T  State=` ou `T  Set<State>=`. De cette façon, on pourra aisément construire les fonctions de transition de signature $Q \times \Sigma \to Q$ mais aussi $2^Q \times \Sigma \to 2^Q$.

- `DFA.java`

  Représente un automate fini déterministe (i.e. Deterministic Finite Automaton).

  Cette classe vient étendre la classe FSM précédente des deux attributs `start (State)` et `delta (Map<Transition<State>, State>)` représentant respectivement l'état initial de l'automate s et la fonction de transition δ.

  Ici, δ est représentée par une fonction de haschage (i.e. dictionnaire) prenant en clé une transition et retournant un état unique comme valeur.

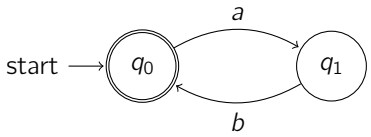
  La classe contient les fonctions basiques de type `getters, toString, equals` et `hashCode`.

Lecture d'un automate

Un automate est représenté par un fichier JSON qui liste ses différents attributs.
Pour rappel un fichier JSON se structure en un format de type `(clé, valeur)`, délimité par des virgules. La valeur peut être une valeur unique (i.e. `value`), un tableau (i.e. `array` délimité par des `[]`) ou un objet (i.e. `object`délimité par des `{}`) qui forme une suite de `(clé, valeur)`.
Dans notre cas, les clés du fichier pour nos automates sont :

- `states : array`
- `alphabet : array`
- `ends : array`
- `start : value` (Pour les AFD) / `starts : array` (Pour les AFND)
- `delta : object`
  - `state : value`
  - `symbol : value`
  - `image : value` (Pour les AFD) / `images : array` (Pour les AFND)

Par exemple, l'automate ci-dessous :



peut être implémenté par le fichier .json suivant :

```
{
      "states" : ["q_0","q_1"],
      "alphabet" : ["b","a"],
      "ends" : ["q_0"]
      "start" : "q_0",
      "delta" : [
        {
         "state" : "q_0",
         "symbol" : "a",
         "image" : "q_1"
        },
        {
         "state" : "q_1",
         "symbol" : "b",
         "image" : "q_0"
        }
      ]
}
```

Travail à faire

1. Importation des fichiers.
   (a) Créer un nouveau projet Java nommé "TDLA_TP1_NOM_PRENOM" et importer les différentes classes de l'archive Automate sur Celene.
   (b) Ajouter à votre projet les deux fichiers `json-20220320.jar` et `json-simple-1.1.jar` permettant la lecture des fichiers JSON.
2. Créer une classe `Main.java` puis créer l'automate DFA correspondant au fichier `DFA1.json`. Imprimer sa description à l'aide de la méthode `toString()`.
3. Compléter la méthode `boolean accept(String x)` qui retourne True si le mot `x` est accepté par un automate AFD et False sinon.
4. Tester votre méthode sur des fichiers de test DFA.
5. Créer une nouvelle classe `NFA.java` permettant l'implémentation d'un automate fini non-déterministe. On s'appuyera sur la classe DFA existante.
   (a) Modifier les attributs afin de tenir compte de l'existance de plusieurs états initiaux et que la fonction de transition retourne un ensemble d'états.

(b) Créer les contructeurs de la classe NFA. On implémentera un constructeur par défaut et un contructeur par fichier `NFA(String path)` où `path` est le chemin vers un fichier JSON. On veillera a respecter la nomenclature spécifiée dans la section **Lecture d'un automate**.

6. Créer une méthode `Set<State> applyDeltaTilde(Transition<Set<State>> t)` qui permet d'appliquer la fonction de transition Δ sur un ensemble d'états issus de la transition `t` et retourne l'ensemble d'états correspondant à l'image de t par Δ. On notera la signature `Set<State>`.

7. Créer une méthode `boolean accept(String x)` qui retourne True si le mot `x` est accepté par l'automate AFND et False sinon.

8. Créer une classe `AFNDe.java` permettant l'implémentation d'un AFND-ε. Pour cela, on pensera à ajouter le symbole ε à l'alphabet de l'automate.

9. Créer une méthode `Set<State> epsilonClause(Set<State> A)` qui retourne $C_\varepsilon(A)$, soit l'ensemble des états accessibles par ε-clôture depuis les états $q \in A$.

10. Créer une méthode `boolean accept(String x)` qui retourne True si le mot `x` est accepté par l'automate AFNDε et False sinon, on pensera à modifier la fonction de transition afin de tenir compte de l'ε-clôture.

11. Créer une fonction `DFA toDFA()` qui retourne la version déterministe d'un AFND ou AFNDε.

12. Créer une fonction `DFA minimize()` qui retourne la version minimal d'un automate AFD selon l'algorithme du cours de votre choix.