Des NFA-ϵ aux NFA

Projet à rendre par courier électronique (`Luc.Maranget@inria.fr`) avant le mercredi 14 février, minuit.

1 Preliminaires

Le projet est une extension du dernier TP sur les automates. Sont fournies deux classes `State` et `Auto` qui sont une solution au TP.

Par rapport à celle du TP, notre classe `State` présente une grosse et une petite différence.

- Les transitions étiquetées sont représentées non plus par un tableau d'états indexé par les charactères mais par une table d'associations des charactères (`Character`) vers les ensembles d'états (`Set<State>`) :

```
/* Les deux sortes de transitions */
private Map<Character,Set<State>>  transition ; // Étiquetées
private Set<State>  epsilon;                    // Spontanées
```

  En outre, les champs ci-dessus sont privés, de sorte que tout travail sur les transitions se fait par le truchement de méthodes, que voici :

  - D'abord lecture des transitions :

```
// Retourne toutes les transitions (étiquetées) issues de this
Map<Character, Set<State>> getAllTransitions()
// Retourne le nouvel ensemble d'etats apres une transition avec la lettre c
Set<State> doTransition(char c)
// Retourne le nouvel ensemble d'etats apres une transition spontanée.
Set<State> doTransitions()
```

  - Ensuite création des transitions :

```
// Ajouter la transition this -c-> next
void addTransition(char c, State next)
// Ajouter toutes les transitions this -c-> next, next ∈ nexts
void addTransition(char c, Set<State> nexts)
// Ajouter une transition this --> next
void addTransition(State next)
```

    Un exemple simple d'usage de ces méthodes est donné plus loin.

- La méthode de fermeture transitive des transitions spontanées s'appelle `epsilonClosure`, existe en deux exemplaires, et surtout est non destructive :

```
/* Fermeture transitive des transitions spontanées */

// Renvoie ϵ*(this)
Set<State> epsilonClosure()
// Renvoie ϵ*(cs), cs n'est pas modifié.
static Set<State> epsilonClosure(Set<State> cs)
```

On peut noter pour la suite que la méthode **static** `Set<State> epsilonClosure(Set<State> cs)` implémente exactement la fonction notée ϵ^* dans le cours.

De son côté, la méthode `Set<State> doTransition(`**char** `c)` implémente $c(\{$ **this** $\})$.

2 Fabriquer un NFA sans transitions spontanées

Ici nous utilisons largement les concepts et les notations du cours 09.

En théorie

Soit A un NFA-ϵ, $A = (Q, \delta, q_0, F, \epsilon)$. Nous définissons d'abord les états significatifs. Sont significatifs :

- L'état initial;
- Tout état $q \in Q$ tel qu'il existe un état $q' \in F$, avec
$$q \longrightarrow^* q';$$
- Tout état $q \in Q$ tel qu'il existe un caractère c et un état $q' \in Q$, avec
$$q \xrightarrow{c} q'.$$

Nous definissons ensuite un NFA équivalent au NFA-ϵ, $B = (Q', \delta', q'_0, F')$

- Les états Q' sont les états significatifs de A.
- Soient q et q' dans Q', on a $q \xrightarrow{c} q'$ dans B, si et seulement si, dans l'automate A, on a
$$q \longrightarrow^* q'' \xrightarrow{c} q''' \longrightarrow^* q'.$$
- L'état initial q'_0 est q_0.
- L'état $q \in Q'$ est final pour B, si et seulement si il existe $q' \in F$, avec, dans l'automate A,
$$q \longrightarrow^* q'.$$

Un algorithme

Les structures de données des classes `State` et `Auto` interdisent le partage des états entre les automates A et B — essentiellement parce que les transitions sont des champs des états `State`. Il va donc falloir procéder à une copie des états significatifs de A.

1. Créer une table d'associations M, des états de A vers les états de B. Poser $Q' = \emptyset, F' = \emptyset$.
2. Pour chaque état significatif o de A :
   (a) Créer un nouvel état n et ajuster $Q' \leftarrow Q' \cup \{ n \}$.
   (b) Ajouter l'association $o \mapsto n$ à M, en outre :
      i. Si o est q_0, alors q'_0 est n.
      ii. Si $\epsilon^*_A(o) \cap F \neq \emptyset$, alors $F' \leftarrow F' \cup \{ n \}$.

   À ce stade, tous les états de Q' sont connus, il reste à calculer les transitions δ'.
3. Pour chaque état significatif o de A :
   (a) Retrouver $n = M(o)$.

(b) Calculer $O_1 = \epsilon_A(o)$, pour chaque o_1 dans O_1.

i. Pour chaque transition (dans A)

$$o_1 \xrightarrow{c} o_2,$$

A. Calculer $O_3 = \epsilon_A(o_2)$.

B. Pour chaque o_3 de O_3, ajouter la transition suivante à B.

$$n \xrightarrow{c} M(o_3).$$

Avec les notations du cours, on vient de calculer $c_B(n) = \epsilon_A^*(c_A(\epsilon_A^*(o)))$.

Un peu d'aide

Pour implémenter l'algorithme il est bon de savoir que...

- La table d'associations M peut s'implémenter indifféremment par un `TreeMap<State,State>` ou un `HashMap<State,State>`, tous deux implémentent l'interface `Map<State,State>`. En outre la classe `State` définit les méthodes nécessaires dans les deux cas.
- Pour itérer sur un ensemble d'états (interface `Set<State>`), on utilise la boucle « *foreach* » (voir le cours 04, pour quelques détails).

```
Set<State> qs = ... ;
for (State q : qs) {
  ...
}
```

- Pour itérer sur toutes les transitions étiquetées issues d'un état `q`, on peut procèder ainsi (voir encore le cours 04) :

```
State q = ... ;
Map<Character,Set<State>> allTrans = q.getAllTransitions() ;
for (Map.Entry<Character, Set<State>> : allTrans.entrySet()) {
  char c = e.getKey() ;
  Set<State> rs = e.getValue() ;
  for (State r : rs) {
    // Agir sur la transition q -c-> r
  }
}
```

Pour vous donner une idée plus précise des techniques employées, la classe `Auto` fournie, comporte une méthode de copie des automates.

```
// Exemple utile : copie de l'automate this.
Auto copy() {
  // Le résultat
  Auto r = new Auto () ;
  // Association des anciens états (de this) vers les nouveaux (de r)
  Map<State, State> old2new  = new TreeMap<State,State> () ;

  /* Fabriquer les états de la copie */
  for (State o : states) {
    State n = null ;
    if (o == initial) { // Dans ce cas, le nouvel état existe déjà
      n = r.initial ;
    } else { // Ici il  faut le créer
      n = new State () ;
      r.states.add(n) ;
```

```
      }
      old2new.put(o, n) ; // Enregistrer que n est la copie de o
      // o final <=> n final
      if (accept.contains(o)) r.accept.add(n) ;
    }

    /* Fabriquer les transitions de la copie */
    for (State o : states) {
      State n = old2new.get(o) ; // Récupérer la copie de l'état o
      // Copier les transitions spontanées
      for (State oo : o.doTransitions()) {
        n.addTransition(old2new.get(oo)) ;
      }
      // Copier les transitions étiquetées
      for (Map.Entry<Character,Set<State>> e :
             o.getAllTransitions().entrySet()) {
        char c = e.getKey() ;
        for (State oo : e.getValue()) {
          n.addTransition(c, old2new.get(oo)) ;
        }
      }
    }
    return r ;
  }
```

Noter que vous n'avez normalement pas besoin d'appeler `copy`, c'est juste un exemple.

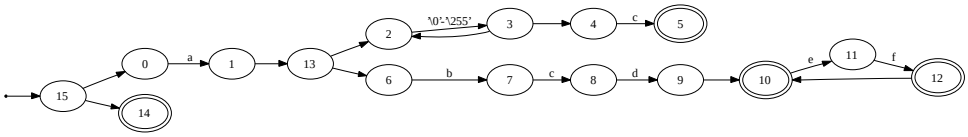
3 Ce qu'il faut faire

Enrichir la classe `Auto` de deux méthodes.

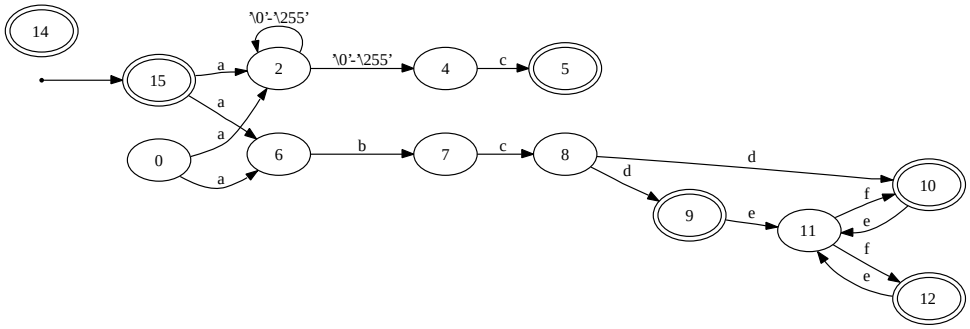
- Une méthode `Auto noE()` qui renvoie un NFA sans transitions spontanées et qui reconnaît le même langage que **this**.
- Une méthode **boolean** `run(String txt)` qui décide de la reconnaissance de `txt` par **this**. On utilisera obligatoirement le *backtracking*. Vous pouvez supposer que **this** est un automate sans transitions spontanées, ou mieux ne pas le supposer.

Pour tester votre code, vous disposez des moyens suivants.

- Visualisation des automates par le méthode `dump`, comme durant le TP. Par exemple, avec la méthode `main` donnée, on obtient normalement le NFA-ϵ



où `'\0'-'\255'` correspond à tous les caractères jugés possibles ici. Une fois écrite la méthode `noE`, on obtient le NFA



La méthode `main` donnée produit deux fichiers au format `dot`, `nfae.dot` et `nfa.dot` que vous pouvez visualiser par :

```
% dot -Tgif nfae.dot | xv -
% dot -Tgif nfa.dot | xv -
```

Notez que rien ne vous empêche de procéder avec des automates plus simples.

- Comparaison des résultats de `a.match(s)`, de `a.noE().match(s)` et de `a.noE().run(s)` (cf. la méthode `Auto.main`).