**TDM 7 - Automates et Langages**

novembre 2018

Implémentation de l'algorithme de Aho et Corasick

# 1 La construction de Aho et Corasick

Cet algorithme dû à Alfred Aho et Margaret Corasick permet de construire un automate déterministe de recherche d'un ensemble de mots dans un texte.

La donnée de l'algorithme est un ensemble non vide $\mathcal{M}$ de $n$ mots non vides, son résultat est un automate déterministe reconnaissant $L = X^*.\mathcal{M}$ (où $X$ désigne l'ensemble des caractères possibles).

## 1.1 Le squelette de l'automate

La première étape consiste à construire un automate **déterministe** reconnaissant $\mathcal{M}$, qui constitue la base de la construction (tous les états, et une partie des transitions, sont créés).
À cette étape, l'automate prend la forme d'un arbre dont la racine est l'état initial. Voici d'abord un exemple :

$$\mathcal{M} = \{try, cry, create, at\}$$

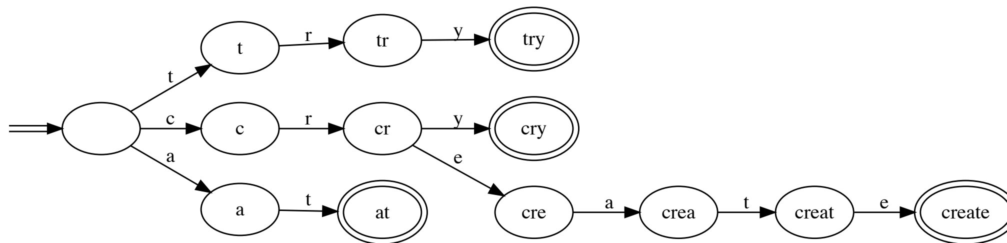


Figure 1 – squelette de l'automate

Il comporte un état pour chaque préfixe de $\mathcal{M}$. Ce préfixe a d'ailleurs été choisi comme nom.

$$Pref(M) = \{\varepsilon, a, c, t, at, cr, tr, cre, cry, try, crea, creat, create\}$$

D'une manière plus générale, l'automate squelette est défini par

— $\mathcal{Q} = Pref(\mathcal{M})$ (par convention, chaque préfixe est aussi choisi comme nom d'un état ; l'état $\varepsilon$ sera plus souvent appelé *racine*, ou *root*).
— état initial : *racine* (alias $\varepsilon$)
— états acceptants : $\mathcal{M}$
— transitions : soit $x$ une lettre. Si $u \in \mathcal{Q}$ et $u.x \in \mathcal{Q}$ alors $\delta(u, x) = u.x$

Chaque état sera aussi désigné par un numéro : son rang de création (en attribuant 0 à la racine) .
**Les états doivent être créés (et numérotés) par « profondeur croissante »** (tout état situé à distance $p$ de la racine est créé **avant** tout autre état situé à distance $p + 1$).
Voici le même automate, avec les états numérotés

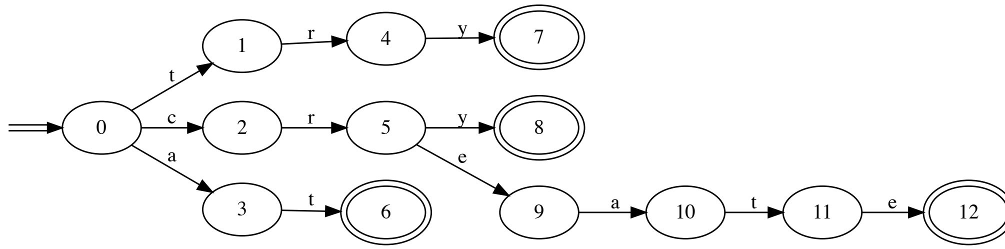


FIGURE 2 – squelette de l'automate avec numéro des états (ordre de création)

### 1.1.1 Algorithme de création du squelette

**Require :** *mots* : ensemble non vide de mots non vides
  **function** SQUELETTE(*mots*)
    *Fifo file*; ▷ file d'attente des suffixes de mots encore à traiter
      ▷ elle contient des triplets :
      ▷ (mot, longueur du préfixe déjà traité, état atteint)
    *État racine* ← nouvel état ;
    **for all** *mot in mots* **do**
      ajouter à la file : $(mot, 0, racine)$ ;
    **end for**
    **while** *file* non vide **do**
      ▷ traiter la prochaine lettre du 1er mot en attente
      $(mot, l, etatCourant)$ ← prélever la tête de*file*
      *État* $q \leftarrow \delta(etatCourant, mot[l])$ ;
      **if** $q$ est indéfini **then** ▷ créer un état et une transition
        $q \leftarrow ajouterNouvelEtat(etatCourant, mot[l])$)
      **end if**
      ▷ $q$ est l'état atteint par le préfixe de *mot* de longueur $l+1$
      **if** $(l+1) == length(mot)$ **then** ▷ fin du mot
        marquer $q$ comme acceptant ;
      **else** ▷ il reste un suffixe à traiter
        ajouter $(mot, l+1, q)$ à la file ;
      **end if**
    **end while**
  **end function**
  **function** AJOUTERNOUVELETAT(etatParent,lettre)
    *État nouveau* ← *créer un état* ;
    ajouter la transition : $\delta(etatParent, lettre) \leftarrow nouveau$ ;
    **return** nouveau ;
  **end function**

Cet algorithme sera amendé à la section suivante.

## 1.2 Les états de repli

À chaque état (sauf l'état racine), nous allons maintenant associer un « état de repli ». Il s'agit de l'état vers lequel se replier quand la progression sur la branche n'est pas possible. Par exemple pour *crea*, l'état de repli est *a* car *a* peut être le début d'un mot de $\mathcal{M}$.
Si $u \in \mathcal{Q},\ u \neq racine$ est un état de l'automate, alors

$$repli(u) = v,\ v \text{ est le } \underline{\text{plus long suffixe}} \text{ de } u \text{ tel que } \ v \neq u \text{ et } v \in \mathcal{Q}$$

Reprenons l'exemple précédent :

— le repli de l'état `creat` est l'état `at`. (`reat` n'est pas prefixe de $\mathcal{M}$, ni `eat`, mais `at` l'est)
— le repli de l'état `crea` est l'état `a`. (`rea` n'est pas prefixe de $\mathcal{M}$, ni `ea`, mais `a` l'est)

— le repli de l'état `at` est l'état `t`.
— le repli de l'état `cre` est l'état $\varepsilon$. (`cr` n'est pas prefixe de $\mathcal{M}$, ni `e`, mais $\varepsilon$ l'est)
— vous pourrez vérifier que pour tous les autres états, l'état de repli est $\varepsilon$ (encore appelé « root »).

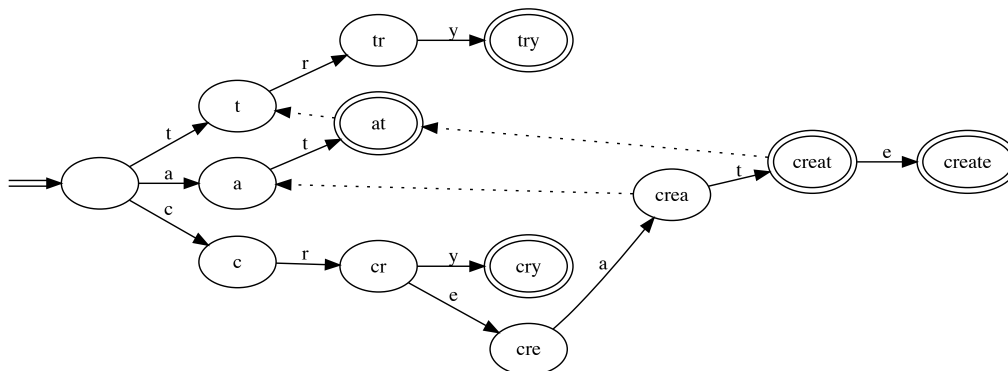


FIGURE 3 – Replis : représentation simplifiée (les replis vers la racine ne sont pas dessinés)

### 1.2.1 Calcul de l'état de repli

1. remarquons tout d'abord que $repli(u)$ est un mot strictement plus court que $u$, il correspond donc à l'un des états créés <u>avant</u> celui de $u$ par l'algorithme de la partie 1. Ainsi, on pourra établir quelle est la valeur de $repli(u)$ dès la création de l'état $u$.
2. la séquence définie par

$$r_1(u) = repli(u), \quad r_i(u) = repli(r_{i-1}(u)),\ pour\ i > 1,\ et\ r_{i-1}(u) \neq \varepsilon$$

   contient tous les suffixes stricts de $u$ appartenant à $\mathcal{Q}$, ordonnés par longueurs décroissantes. Cette séquence est finie et se termine par $\varepsilon$

   Par exemple $repli(repli(u))$ est un suffixe de $repli(u)$, donc un suffixe de de $u$. Il est le plus long suffixe strict de $repli(u)$ appartenant à $\mathcal{Q}$, donc le 2ème plus grand suffixe strict de $u$ appartenant à $\mathcal{Q}$.
3. considérons un mot $u \in \mathcal{Q}$ se terminant par la lettre $x$. Il s'écrit $u'.x$ où $u'$ est l'état « parent » de $u$. Un suffixe de $u$ est soit vide, soit un mot de la forme $v'.x$, où $v'$ est un suffixe de $u'$.

   $repli(u)$ est donc le plus long mot $v'.x$ tel que, à la fois, $v'$ est suffixe strict de $u'$, $v' \in \mathcal{Q}$ et $v'.x \in \mathcal{Q}$

   Or nous venons de voir que la séquence $r_i(u')$ contient exactement tous les suffixes stricts de $u'$ appartenant à $\mathcal{Q}$, par longueurs décroissantes. Si $repli(u) \neq \varepsilon$ alors il est le successeur par $x$ de l'un de $r_i(u')$.

   L'algorithme de calcul de l'état de repli d'un état $u = u'.x$ consistera donc à parcourir la séquence d'états $r_i(u')$, c'est à dire $repli(u'), repli(repli(u')), repli(repli(repli(u'))), ...$ jusqu'à en trouver un qui possède un successeur pour la lettre $x$. Si aucun ne convient $repli(u) = \varepsilon$
4. enfin, remarquons que si $repli(u)$ est un état acceptant, c'est qu'il se termine par un mot de $\mathcal{M}$, et donc que $u$ se termine par ce même mot : $u$ doit alors également être un état acceptant.

   Par exemple $repli(creat) = at$ qui est acceptant. *creat* doit donc aussi être acceptant (il se termine par *at*).

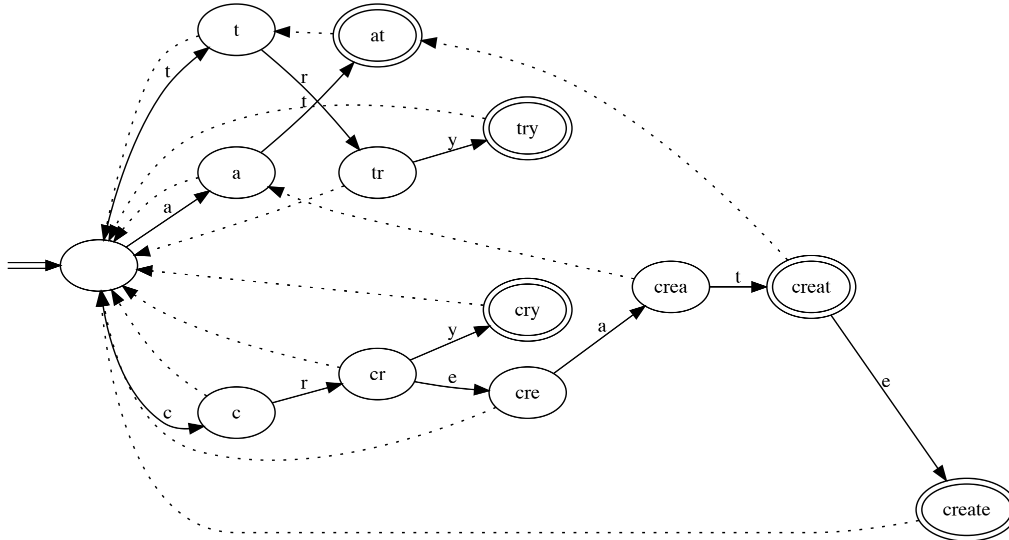


Figure 4 – Replis : représentation complète

Le calcul des états de repli se réalise dans le même temps que la création du squelette, en l'intégrant à la fonction *ajouterNouvelEtat* :

### 1.2.2 Algorithme

**Require :** *repli* : une map associant à chaque état son état de repli. Initalement toutes les valeurs sont égales à null
**function** AJOUTERNOUVELETAT(etatParent,lettre)
  *État nouveau* $\leftarrow$ *créer un état*;
  ajouter la transition : $\delta(etatParent, lettre) \leftarrow nouveau$;
  *État* $r \leftarrow repli(parent)$
  **while** $r \neq null$ **and** $\delta(r, lettre)$ *est indéfini* **do**
    $r \leftarrow repli(r)$;
  **end while**
  **if** $r = null$ **then** ▷ Aucun $r_i$ ne possède un successeur pour *lettre*
    $repli[nouveau] \leftarrow racine$ ▷ Seul repli possible
  **else** ▷ $\delta(r, lettre)$ existe : c'est l'état de repli
    $repli[nouveau] \leftarrow \delta(r, lettre)$
    **if** $repli[nouveau]$ est acceptant **then**
      marquer *nouveau* comme état acceptant
    **end if**
  **end if**
  **return** nouveau;
**end function**

## 1.3 Ajouter des transitions

Il nous reste à compléter l'automate par de nouvelles transitions :

Pour chaque état $u$ et chaque lettre $x$, si $\delta(u, x)$ n'est pas définie, alors on ajoute la transition $\delta(u, x) = \delta(repli(u), x)$.

Par exemple,
— si la lettre $r$ est rencontrée à partir de l'état `at`, il faut aller dans l'état `tr` (ce peut être le début du mot *try*). Autrement dit :

la transition $\delta(\texttt{at}, r)$ était indéfinie. Elle prend maintenant la valeur de
$\delta(repli(\texttt{at}), r) = \delta(\texttt{t}, r) = \texttt{tr}$

— de la même façon, si la lettre $r$ est rencontrée à partir de l'état `creat`, il faut aller en `tr` :
la transition $\delta(\texttt{creat}, r)$ était indéfinie. Elle prend maintenant la valeur de
$\delta(repli(\texttt{creat}), r) = \delta(\texttt{at}, r) = \texttt{tr}$

— cet exemple montre la nécessité, cette fois encore, de traiter les états « par profondeur croissante » (l'ordre de création des états convient).

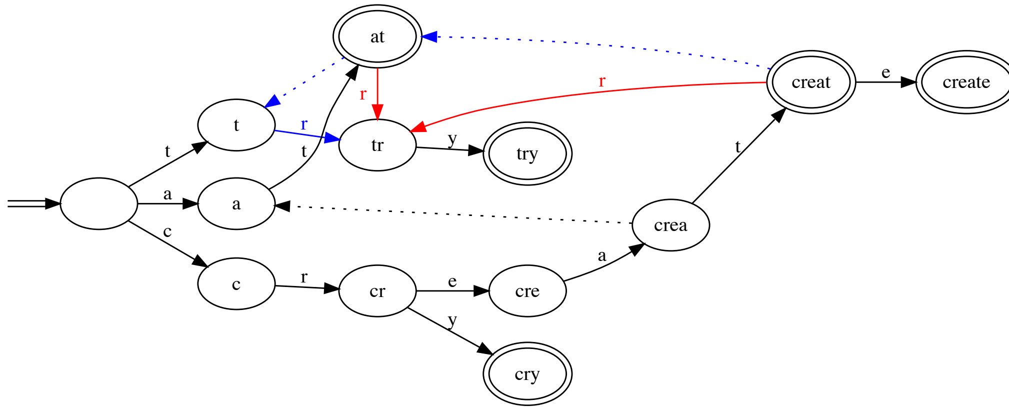


Figure 5 – Exemple d'ajout de deux transitions (en rouge)
en bleu : les replis et la transition du squelette qui ont permis cet ajout

### 1.3.1 Algorithme

**function** CompléterAutomate
  **for all** *État* $u$ (états à parcourir par ordre de création) **do**
    *État* $r \leftarrow repli[u]$
    **for all** transition $\delta(r, lettre)$ définie **do**
      **if** $\delta(u, lettre)$ est indéfinie **then**
        ajouter transition : $\delta(u, lettre) \leftarrow \delta(r, lettre)$
      **end if**
    **end for**
  **end for**
  ▷ toutes les transitions non encore définies doivent ramener à la racine
  ▷ NB : cette action est fastidieuse et coûteuse en place si l'alphabet est grand
  ▷ En pratique, on l'évitera en choisissant une implémentation d'automate
  ▷ prévoyant la définition d'une transition "par défaut"
  **for all** *État* $u$ **do**
    **for all** transition $\delta(u, lettre)$ indéfinie **do**
      ajouter transition : $\delta(u, lettre) \leftarrow racine$
    **end for**
  **end for**
**end function**

Voici l'automate final obtenu pour notre exemple
Là encore, pour des raisons de lisibilité, les flèches amenant à l'état racine n'ont pas été dessinées.

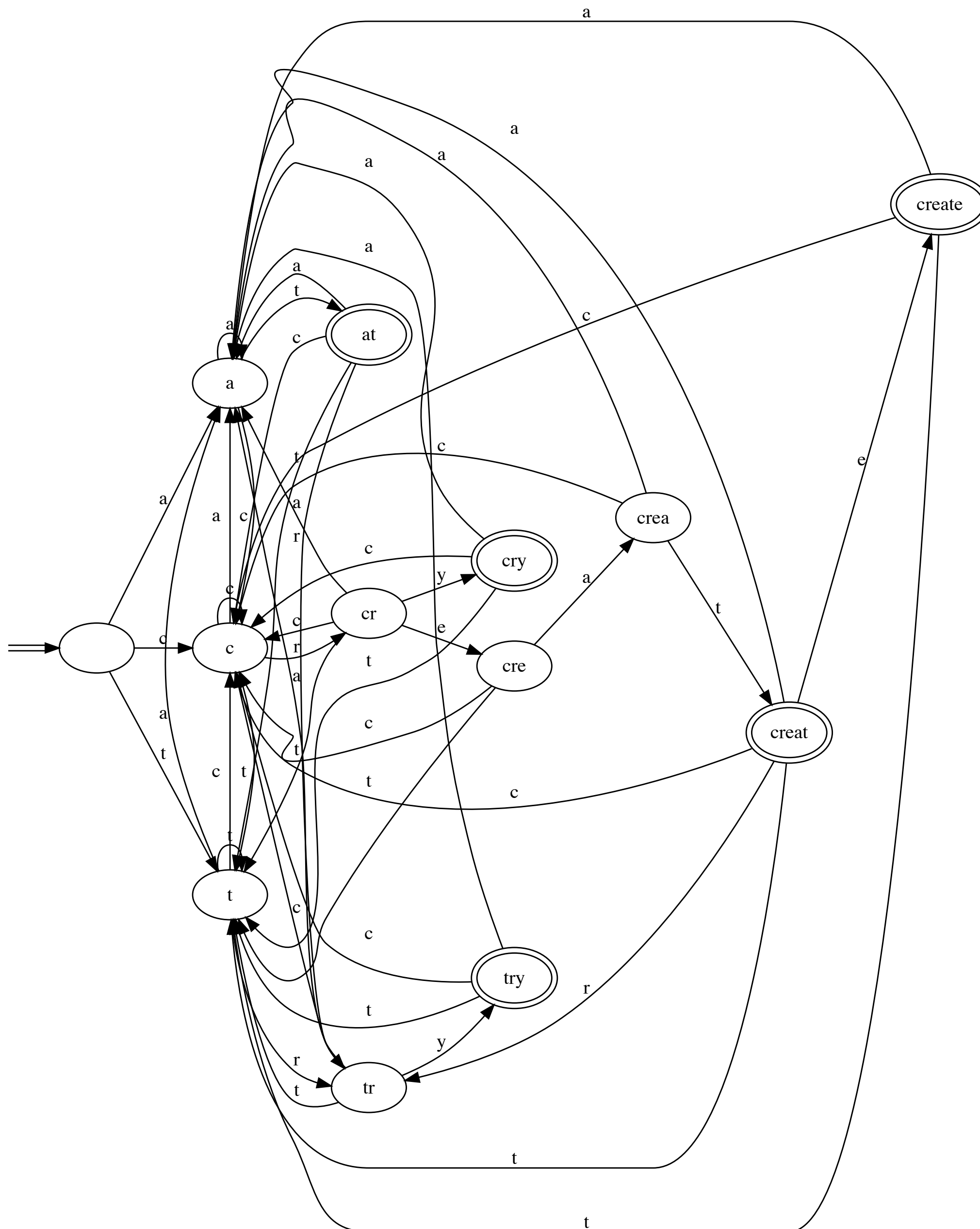


Figure 6 – automate final : représentation simplifiée (!).
En réalité l'automate est complet car toutes les flèches non dessinées ramènent à l'état initial

## 2 Travail à faire

Implémenter cet algorithme. Vous disposez de la base d'une classe `AhoCorasick`, à compléter.
La classe `AhoCorasick` implémente un automate, qui est instancié à partir d'une liste de mots, en suivant la construction de Aho et Corasick.
Voici une utilisation typique de la classe :

```
AhoCorasick a = new AhoCorasick("try","cry","create","at");
```

La classe implémente l'interface `Automaton` en ajoutant quelques méthodes publiques supplémentaires, en particulier

```
        Set<String> getFoundWords(State q);
```

qui renvoie la liste des mots reconnus quand l'on atteint l'état (supposé final) $q$.

La construction de la table foundWords ne présente pas de difficulté mais n'est pas décrite dans l'algorithme ci-dessus. Elle est à insérer dans les méthodes skeleton() et addNewState()
Les autres méthodes publiques spécifiques à la classe concernent la génération Graphviz et vous sont intégralement fournies.

### 2.1 Quelques indications

1. l'interface java.util.Queue<E> définit les méthodes d'une file (FIFO). Elle est implémentée par plusieurs classes de l'API standard , notamment java.util.ArrayDeque<E> ou java.util.LinkedList<E>.
2. il vous faudra définir une classe interne pour porter les objets à mettre dans la file.
3. les itérateurs sur l'ensemble renvoyé par la méthode getStates() (définie par les classes d'automate fournies) **garantissent** un parcours des états par numéros croissants (ordre de création). Ainsi les codes

   ```
   for (State q : a.getStates()) {...}
   ```

   ou

   ```
   Iterator<State> it =  a.getStates().iterator();
   while (it.hasNext()) {State q = it.next(); ...}
   ```

   réalisent un parcours des états dans l'ordre de leur création.