Rapport de projet

Techniques de compilation

Ecrire un programme qui transforme un automate fini non-déterministe en un automate fini déterministe équivalent

Auteur : DOUIH Zakaria & NAIT-EL-HAJ Abderrahmane

Encadré par : M. KABBAJ Adil

Date : 8 janvier 2025

Filière DSE
INSEA

Table des matières

Introduction

Les automates finis sont des modèles mathématiques fondamentaux dans le domaine de la théorie des langages formels et de la compilation. Ils sont largement utilisés pour modéliser et analyser des systèmes discrets tels que les expressions régulières, les analyseurs lexicaux, et même certains aspects des protocoles de communication. Deux types majeurs d'automates finis sont couramment étudiés : les automates finis non déterministes (AFN) et les automates finis déterministes (AFD).

Un automate fini non déterministe est un modèle dans lequel, pour un état donné et une entrée donnée, plusieurs transitions peuvent être possibles. Ce caractère non déterministe rend les AFN souvent plus simples et plus intuitifs à construire. Cependant, cette nature non déterministe peut poser des défis lorsqu'il s'agit d'implémenter ces automates dans des environnements logiciels ou matériels où le déterminisme est essentiel.

(a) Automate déterministe

(b) Automate non déterministe

FIGURE 1 – Exemples d'automates

Pour pallier ces limitations, il est courant de transformer un AFN en un automate fini déterministe équivalent (AFD). Cette transformation repose sur l'algorithme de déterminisation, qui repose sur des concepts tels que l'ensemble des états et la construction d'ensembles d'états accessibles. L'AFD résultant permet de garantir un comportement univoque pour chaque entrée, rendant ainsi l'automate directement utilisable dans des contextes pratiques.

Dans ce projet, nous visons à implémenter un programme informatique capable de convertir un automate fini non déterministe donné en un automate fini déterministe équivalent. Cette transformation, bien que conceptuellement bien établie, comporte plusieurs défis liés à la gestion des ensembles d'états et à la représentation efficace des transitions. Ce rapport présente le contexte théorique, les étapes de conception, l'algorithme utilisé, ainsi que les résultats obtenus à travers notre implémentation.

Chapitre 1

Définition d'un automate

1.1 Définition formelle d'un automate

Un **automate fini déterministe (AFD)** est défini comme un quintuplet $A = (Q, \Sigma, \delta, q_0, F)$, où :

— Q est un ensemble fini d'états ;

— Σ est un alphabet fini (l'ensemble des symboles d'entrée) ;

— $\delta : Q \times \Sigma \to Q$ est une fonction de transition définissant le passage d'un état à un autre en fonction d'un symbole de l'alphabet ;

— $q_0 \in Q$ est l'état initial ;

— $F \subseteq Q$ est l'ensemble des états finaux ou acceptants.

Un **automate fini non déterministe (AFN)** est similaire, mais sa fonction de transition est définie comme une relation :

$$\delta : Q \times \Sigma \to \mathcal{P}(Q),$$

où $\mathcal{P}(Q)$ désigne l'ensemble des parties de Q.

Langage accepté : Un automate A accepte un mot $w \in \Sigma^*$ si, en partant de l'état initial q_0, il existe une séquence de transitions aboutissant à un état final $q_f \in F$. Le langage reconnu par A, noté $L(A)$, est l'ensemble des mots acceptés par A :

$$L(A) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \cap F \neq \emptyset\},$$

où δ^* est l'extension de δ à Σ^*.

Implémentation de l'automate en Java

Code Java avec descriptions des méthodes :

```java
import java.util.*;
import java.util.Map.Entry;

// Classe repr sentant un automate
public class Automate {
    Map<String, Map<String, Set<String>>> transitions;
```

```java
    // Chaque premi re cl  repr sente un  tat
    // Chaque seconde cl  repr sente un symbole terminal
    // Chaque ensemble contient les  tats  successeurs associ s

    public Set<String> etatsInitiaux;
    public Set<String> etatsFinaux;
    private String etatInitialAFD = "I0";

    // Constructeur : initialise l'automate avec des ensembles d' tats  initiaux et finaux
    public Automate(Set<String> etatsInitiaux, Set<String> etatsFinaux) {
        this.transitions = new HashMap<>();
        this.etatsInitiaux = etatsInitiaux;
        this.etatsFinaux = new HashSet<>(etatsFinaux);
    }

    // M thode pour ajouter un nouvel  tat  l'automate
    public void ajouterEtat(String etat) {
        transitions.putIfAbsent(etat, new HashMap<>());
    }

    // M thode pour ajouter une transition entre deux  tats  avec un symbole donn
    public void ajouterTransition(String etatSource, String symbole, String etatCible) {
        transitions.putIfAbsent(etatSource, new HashMap<>());
        transitions.get(etatSource).putIfAbsent(symbole, new HashSet<>());
        transitions.get(etatSource).get(symbole).add(etatCible);
    }

    // M thode pour afficher les d tails de l'automate :  tats  initiaux, finaux, et transitions
    public void afficher() {
        System.out.println(" tat  initial : " + etatsInitiaux);
        System.out.println(" tats  finaux : " + etatsFinaux);
        System.out.println("Transitions :");
        for (String etat : transitions.keySet()) {
            if (!transitions.get(etat).isEmpty()) {
                System.out.println("Depuis l' tat  " + etat + ":");
                for (String symbole : transitions.get(etat).keySet()) {
                    System.out.println("  Avec '" + symbole + "' -> " +
                        transitions.get(etat).get(symbole));
                }
            }
        }
    }

```

```
49      // M thode pour v rifier si une cha ne donn e est accept e par l'automate
50      public boolean accepter(String chaine) {
51          for (String etatInitial : etatsInitiaux) {
52              if (verifier(chaine, etatInitial)) {
53                  return true;
54              }
55          }
56          return false;
57      }
58
59      // M thode r cursive pour valider une cha ne    partir d'un  tat  donn
60      private boolean verifier(String chaine, String etatActuel) {
61          if (chaine.isEmpty()) {
62              return etatsFinaux.contains(etatActuel);
63          }
64          char symbole = chaine.charAt(0);
65          String reste = chaine.substring(1);
66
67          if (transitions.containsKey(etatActuel) &&
                transitions.get(etatActuel).containsKey(String.valueOf(symbole))) {
68              for (String etatSuivant :
                    transitions.get(etatActuel).get(String.valueOf(symbole))) {
69                  if (verifier(reste, etatSuivant)) {
70                      return true;
71                  }
72              }
73          }
74          return false;
75      }
76  }
```

Listing 1.1 – Classe `Automate` représentant un automate en Java

Description des méthodes :

- `Automate(Set<String> etatsInitiaux, Set<String> etatsFinaux)` : Initialise un automate avec des états initiaux et finaux.
- `ajouterEtat(String etat)` : Ajoute un nouvel état à l'automate.
- `ajouterTransition(String etatSource, String symbole, String etatCible)` : Définit une transition entre deux états.
- `afficher()` : Affiche les états initiaux, finaux, et les transitions de l'automate.
- `accepter(String chaine)` : Vérifie si une chaîne est acceptée par l'automate.
- `verifier(String chaine, String etatActuel)` : Méthode récursive pour valider une chaîne à partir d'un état donné.

Chapitre 2

Automate indéterministe

2.1 Définition formelle

Un **automate fini non déterministe** (AFN) est défini par un quintuplet $A = (Q, \Sigma, \delta, I, F)$, où :
— Q est un ensemble fini d'états.
— Σ est un alphabet fini.
— $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ est la fonction de transition, où $\mathcal{P}(Q)$ représente l'ensemble des parties de Q.
— $I \subseteq Q$ est l'ensemble des états initiaux.
— $F \subseteq Q$ est l'ensemble des états finaux.

Dans un AFN, la fonction de transition δ peut retourner plusieurs états (ou aucun) pour un état donné $q \in Q$ et un symbole donné $a \in \Sigma$. Par conséquent, l'automate peut suivre plusieurs chemins pour lire une chaîne donnée.

Exemple : Considérons un automate avec :
— $Q = \{q_0, q_1, q_2\}$,
— $\Sigma = \{a, b\}$,
— $I = \{q_0\}$,
— $F = \{q_2\}$,
— δ donnée par :

$$\delta(q_0, a) = \{q_0, q_1\}, \quad \delta(q_1, b) = \{q_2\}.$$

Pour accepter une chaîne, il suffit qu'au moins un chemin mène à un état final.

2.2 Méthode de déterminisation

La **déterminisation** consiste à transformer un AFN $A = (Q, \Sigma, \delta, I, F)$ en un automate fini déterministe (AFD) $A' = (Q', \Sigma, \delta', I', F')$ tel que $L(A) = L(A')$. La méthode suit les étapes suivantes :

1. **Construction des états de** A' : Chaque état de A' correspond à un sous-ensemble d'états de Q (c'est-à-dire un élément de $\mathcal{P}(Q)$).
2. **Définition de l'état initial** : $I' = \{I\}$, où I est l'ensemble des états initiaux de A.

3. **Définition de la fonction de transition** δ' : Pour chaque état $S \in Q'$ et chaque symbole $a \in \Sigma$, $\delta'(S, a)$ est défini comme l'union des ensembles suivants :

$$\delta'(S, a) = \bigcup_{q \in S} \delta(q, a).$$

4. **Définition des états finaux** : Un état $S \in Q'$ est final dans A' si et seulement si $S \cap F \neq \emptyset$, où F est l'ensemble des états finaux de A.
5. **Minimisation** (optionnelle) : Une fois l'AFD construit, on peut le minimiser pour réduire le nombre d'états.

Illustration : Soit l'automate donné dans l'exemple précédent. La déterminisation produit les états suivants :

— $S_0 = \{q_0\}$,
— $S_1 = \{q_0, q_1\}$,
— $S_2 = \{q_2\}$.

La fonction de transition de l'AFD devient alors :

$$\delta'(S_0, a) = S_1, \quad \delta'(S_1, b) = S_2.$$

2.3 Implémentation avec Java

```
public void enDeterministe() {
    // Ajout de l'état initial composite si plusieurs états initiaux existent
    if (!this.transitions.containsKey(etatInitialAFD) && etatsInitiaux.size() > 1)
        {
        this.transitions.put(etatInitialAFD, new HashMap<>());
        for (String pseudo : this.etatsInitiaux) {
            for (Map.Entry<String, Set<String>> pseudoEtat :
                this.transitions.get(pseudo).entrySet()) {
                this.transitions.get(etatInitialAFD).put(pseudoEtat.getKey(), new
                    HashSet<>());
            }
        }
    }

    // Construction des transitions pour l'état initial composite
    for (String etatComposant : this.etatsInitiaux) {
        for (String symboleActuel : this.transitions.get(etatComposant).keySet()) {
            for (String etatSuivant :
                this.transitions.get(etatComposant).get(symboleActuel)) {
                if (etatsInitiaux.size() > 1) {
                    this.transitions.get(etatInitialAFD).get(symboleActuel)
                    .add(etatSuivant);
                }
            }
```

```
21          }
22      }
23
24      // Initialisation des structures n cessaires pour la d terminisation
25      int iterateur = 0; // Compteur pour nommer les nouveaux  tats  composites
26      Hashtable<String, HashSet<String>> etatsComposites = new Hashtable<>(); //
            Table pour suivre les  tats  composites
27      List<String> etatsIteration;
28      boolean termine = false;
29
30      do {
31          etatsIteration = new ArrayList<>(this.transitions.keySet()); // Liste des
                tats  actuels  traiter
32          // Parcours de chaque  tat  pour analyser les transitions
33          for (String etat : etatsIteration) {
34              for (String symbole : this.transitions.get(etat).keySet()) {
35                  // V rification si un nouvel  tat  composite est n cessaire
36                  if (this.transitions.get(etat).get(symbole).size() > 1
37                          && !etatsComposites.
38                          containsValue(this.transitions.get(etat)
39                          .get(symbole))) {
40                      String nomComposite = "E" + iterateur;
41                      boolean exists = false;
42
43                      // V rification si un  tat  composite  quivalent  existe
                            d j
44                      for (HashSet<String> value : etatsComposites.values()) {
45                          if (value.equals(this.transitions.get(etat).get(symbole)))
                                {
46                              exists = true;
47                              break;
48                          }
49                      }
50
51                      if (!exists) {
52                          etatsComposites.put(nomComposite, new
                                HashSet<>(this.transitions.get(etat).get(symbole)));
53                          iterateur++;
54                      }
55
56                      // Ajout des transitions pour le nouvel  tat  composite
57                      if (!this.transitions.containsKey(nomComposite)) {
58                          this.transitions.put(nomComposite, new HashMap<>());
59                          for (String pseudo : etatsComposites.get(nomComposite)) {
60                              for (Map.Entry<String, Set<String>> pseudoEtat :
                                    this.transitions.get(pseudo).entrySet()) {
61                                  this.transitions.get(nomComposite).
```

```
                                put(pseudoEtat.getKey(), new HashSet<>());
                            }
                        }
                    }

                    for (String etatComposant : etatsComposites.get(nomComposite)) {
                        for (String symboleActuel : this.transitions.get(etatComposant).keySet()) {
                            for (String etatSuivant : this.transitions.get(etatComposant).get(symboleActuel)) {
                                this.transitions.get(nomComposite).get(symboleActuel).add(etatSuivant);
                            }
                        }
                    }
                }
            }
        }

        // V rification si tous les  tats  ont  t  trait s
        termine = true;
        for (String etat : this.transitions.keySet()) {
            for (String symbole : this.transitions.get(etat).keySet()) {
                if (this.transitions.get(etat).get(symbole).size() > 1
                        && !etatsComposites.containsValue(this.transitions.get(etat).get(symbole))) {
                    termine = false;
                    break;
                }
            }
        }
    } while (!termine);

    // Remplacement des ensembles d' tats  par leurs noms composites
    etatsIteration = new ArrayList<>(this.transitions.keySet());
    for (String etat : etatsIteration) {
        for (String symbole : this.transitions.get(etat).keySet()) {
            if (etatsComposites.containsValue(this.transitions.get(etat).get(symbole))) {
                for (Map.Entry<String, HashSet<String>> entry : etatsComposites.entrySet()) {
                    if (entry.getValue().equals(this.transitions.get(etat).get(symbole))) {
                        this.transitions.get(etat).put(symbole, new HashSet<>());
                        this.transitions.get(etat).get(symbole).add(entry.getKey());
```

```
                    }
                }
            }
        }
    }

    //   limination   des   tats   initiaux multiples
    if (this.etatsInitiaux.size() > 1) {
        for (String etat : this.etatsInitiaux) {
            this.transitions.get(etat).clear();
        }
        this.etatsInitiaux.clear();
        this.etatsInitiaux.add(etatInitialAFD);
    }

    // Ajout des   tats   composites qui sont finaux
    for (Map.Entry<String, HashSet<String>> entry : etatsComposites.entrySet()) {
        for (String etat : entry.getValue()) {
            if (this.etatsFinaux.contains(etat)) {
                this.etatsFinaux.add(entry.getKey());
            }
        }
    }
}
```

Listing 2.1 – Méthode **enDeterministe** qui transforme l'automate cournat en automate déterministe

Chapitre 3

Interface graphique

Pour une meilleure organisation de gestion, voici une interface graphique construit par Swing.

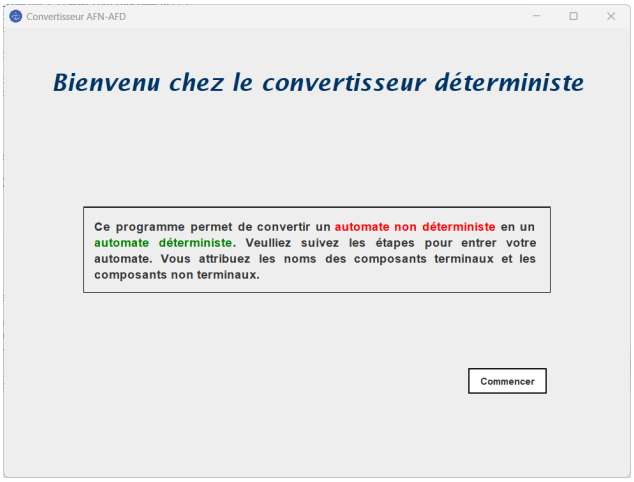
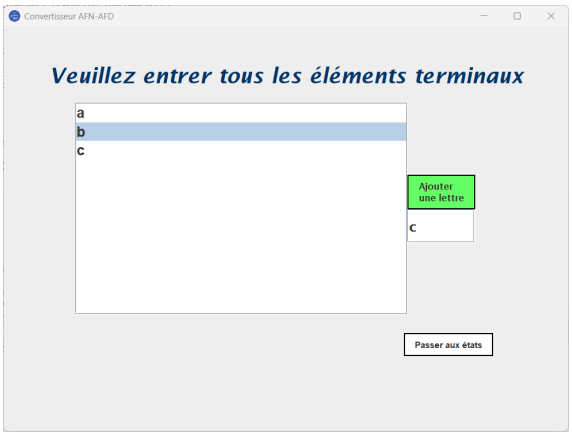
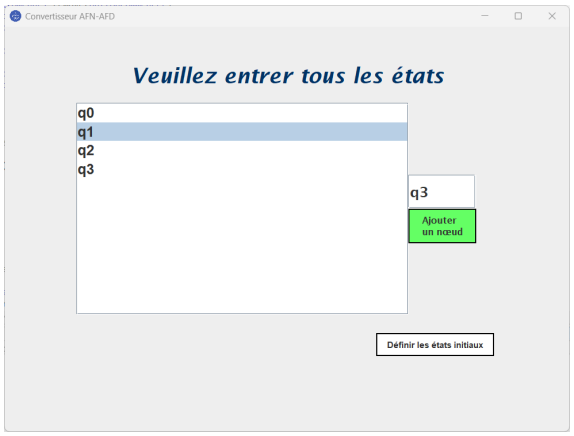


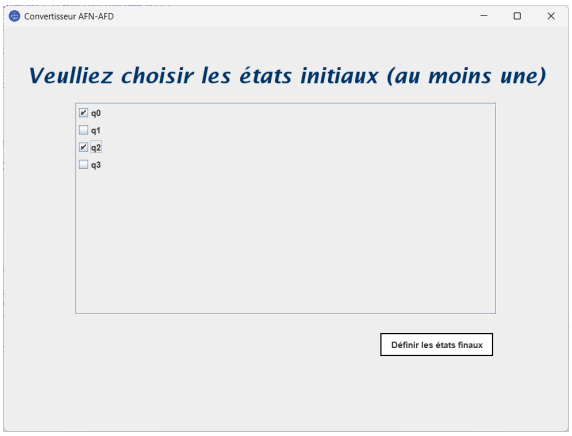
FIGURE 3.1 – Interface d'acceuil



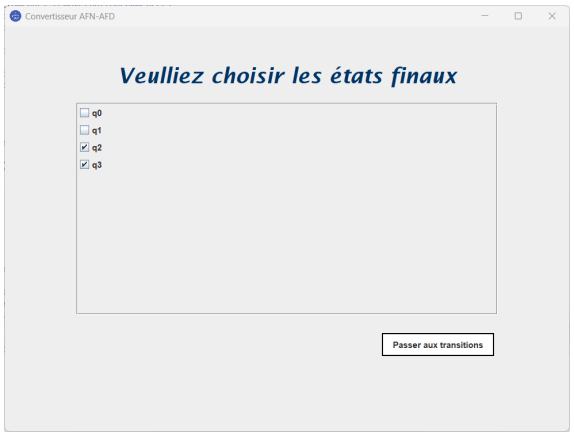
(a) Interface d'ajout des symboles



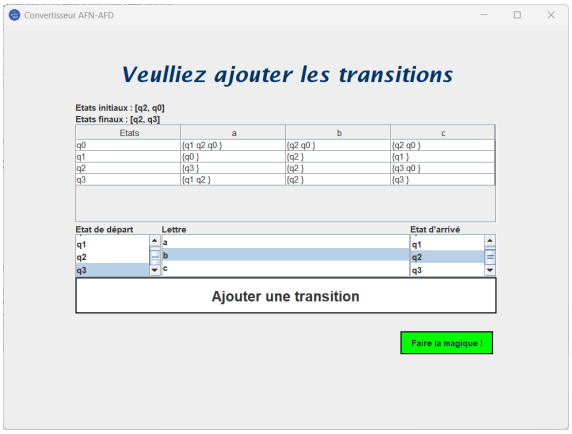
(b) Interface d'ajout des états



(c) Interface de choix des états initiaux



(d) Interface de choix des états finaux



(e) Interface de l'ajout des transitions

(f) Automate AFD

FIGURE 3.2 – Les étapes suivantes

Conclusion

En conclusion, ce travail a permis d'approfondir la compréhension des automates et de leurs propriétés, en mettant particulièrement l'accent sur le processus de détermination. À travers l'étude théorique et la mise en œuvre pratique, nous avons démontré comment un automate non déterministe peut être transformé en un automate déterministe tout en préservant son langage accepté.