Travaux Pratiques
Théorie des langages
Licence 3 Informatique

Julien Bernard

Table des matières

Généralités

Objectifs

Le but de cette série de travaux pratiques est de réaliser une bibliothèque manipulant des automates finis ainsi qu'une application de démonstration. Le découpage en TP représente à peu près le temps que vous devez consacrer à chaque partie pendant trois heures. Si vous êtes en retard par rapport à ce planning, il est nécessaire de prendre du temps en dehors des heures encadrées pour rattrapper le retard.

En pratique

L'application sera codée en langage C++. Le choix de la structure de données pour représenter l'automate est libre. Cependant, des indications vous sont données pour vous guider. Vous devrez également, dans le cadre de l'UE *Outils pour la programmation*, définir un ensemble de tests unitaires pour cette bibliothèque. Il est fortement recommandé d'utiliser le framework de test unitaire Google Test et d'écrire vos tests avant d'écrire votre implémentation.

Une archive contenant un squelette de code pour la bibliothèque, un programme de démonstration et un programme de test sont données sur MOODLE. Elle contient également un fichier `CMakeLists.txt` que vous pouvez utiliser pour construire l'ensemble des programmes.

En plus, pendant vos développements, vous n'aurez pas le droit d'allouer de la mémoire explicitement. Vous devrez utiliser les structures de données existantes dans la bibliothèque standard, et plus particulièrement `std::vector` (tableau dynamique), `std::set` et `std::map` (arbre binaire de recherche pour les ensembles et les tableaux associatifs), `std::queue` (file) et éventuellement d'autres [1].

Évaluation

Il est attendu une application de démonstration qui fabrique des automates et applique les fonctions implémentées. Cette application devra être largement commentée pour convaincre le correcteur de la pertinence de votre démonstration. L'application de tests sera également utilisé pour évaluer la correction de votre implémentation.

Vous serez évalué sur la correction de vos algorithmes, mais également sur la propreté de votre code, sur le choix de vos structures et sur la complexité de vos algorithmes. En particulier, une moulinette sera chargée de vérifier la correction des algorithmes via des tests (qui ne seront pas diffusés), il est donc primordial de bien respecter les interfaces données !

1. voir `http://en.cppreference.com/w/cpp/container`

Travaux Pratiques de Théorie des Langages n°1

Dans ce premier TP, vous allez choisir une structure de données pour représenter un automate puis implémenter des fonctions de base pour construire un automate.

Exercice 1 : Choix de la structure pour l'automate

Le but de cet exercice est de choisir la structure de donnée qui vous semble être la plus adéquate (ou celle que vous saurez maîtriser le mieux). Les conseils donnés ici sont juste des conseils, vous êtes libre de choisir une autre structure si vous le désirez. Vous pouvez réaliser ce choix en même temps que vous faites l'exercice suivant, pour avoir une idée des algorithmes à mettre en œuvre sur chacune des structures. Enfin, n'oubliez pas de commenter votre code pour justifier les choix techniques que vous faites.

La première étape consiste à choisir la structure de données pour représenter un automate. L'implémentation de l'automate sera dans une classe `Automaton`, elle-même dans un espace de nom `fa`. Dans la suite, seule l'interface de cette classe sera décrite, c'est-à-dire les méthodes publiques qu'un utilisateur peut appeler. L'implémentation concrète est entièrement libre tant qu'elle respecte l'interface. En plus, des indications de complexité sont données en fonction de n le nombre d'état, m le nombre de transitions et s la taille de l'alphabet. Ces indications doivent également être respectées.

```
namespace fa {
  class Automaton {
    // your implementation
  };
}
```

Un automate sera défini par :
— un alphabet ;
— un ensemble d'états ;
— un ensemble de transitions.

Concernant l'alphabet, toute lettre imprimable ASCII peut être utilisée. Les lettres seront donc représentées à l'aide du type **char**. Un état sera représenté par un entier positif de type **int**. La représentation interne d'un état doit également prendre en compte le fait que l'état peut être initial et/ou final. L'ensemble des états peut être représenté de multiples manières, soit grâce à un tableau dynamique, soit grâce à un tableau associatif.

Pour la gestion des transitions, de multiples façons de procéder existent. Elles peuvent être définies dans la structure qui gère les états, elles peuvent être définies dans une structure à part. Il faudra prendre garde à ne pas pouvoir ajouter deux fois la même transition.

Exercice 2 : Création d'un automate

Le but de cet exercice est de réaliser les premières fonctions de gestion d'un automate et de construire l'automate exemple de la figure 1.

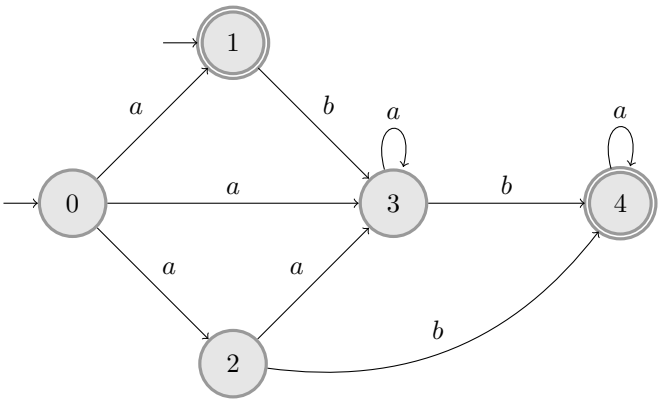

FIGURE 1 – Automate d'exemple

Question 2.1 Écrire des méthodes de gestion des états.

```
/**
 * Add a state to the automaton.
 *
 * The state must not be already present. By default,
 * a newly added state is not initial and not final.
 *
 * Expected complexity: O(log n)
 * Max allowed complexity: O(n)
 */
void addState(int state);

/**
 * Remove a state from the automaton.
 *
 * The state must be present. The transitions
 * involving the state are also removed.
 *
 * Expected complexity: O(log n)
 * Max allowed complexity: O(n)
 */
void removeState(int state);

/**
 * Tell if the state is present in the automaton.
 *
 * Expected complexity: O(log n)
 * Max allowed complexity: O(n)
 */
bool hasState(int state) const;

/**
 * Compute the number of states.
```

```
 *
 * Expected complexity: O(1)
 */
std::size_t countStates() const;
```

Question 2.2 Écrire des méthodes pour gérer les états initiaux et finaux.

```
/**
 * Set the state initial.
 *
 * The state must be present.
 *
 * Expected complexity: O(log n)
 * Max allowed complexity: O(n)
 */
void setStateInitial(int state);

/**
 * Tell if the state is initial.
 *
 * The state must be present.
 *
 * Expected complexity: O(log n)
 * Max allowed complexity: O(n)
 */
bool isStateInitial(int state) const;

/**
 * Set the state final.
 *
 * The state must be present.
 *
 * Expected complexity: O(log n)
 * Max allowed complexity: O(n)
 */
void setStateFinal(int state);

/**
 * Tell if the state is final.
 *
 * The state must be present.
 *
 * Expected complexity: O(log n)
 * Max allowed complexity: O(n)
 */
bool isStateFinal(int state) const;
```

Question 2.3 Écrire des méthodes de gestion des transitions.

```
/**
 * Add a transition
 *
 * The two states must be present. The character is
 * added to the alphabet.
 *
 * Expected complexity: O(log n + log m)
 * Max allowed complexity: O(n + m)
 */
void addTransition(int from, char alpha, int to);

/**
 * Remove a transition
 *
 * The transition must be present. The character is
 * not deleted from the alphabet, even if it is the
 * last transition with this letter.
 *
 * Expected complexity: O(log n + log m)
 * Max allowed complexity: O(n + m)
 */
void removeTransition(int from, char alpha, int to);

/**
 * Tell if a transition is present.
 *
 * The two states must be present.
 *
 * Expected complexity: O(log n + log m)
 * Max allowed complexity: O(n + m)
 */
bool hasTransition(int from, char alpha, int to) const
    ;

/**
 * Compute the number of transitions.
 *
 * Expected complexity: O(n + m)
 */
std::size_t countTransitions() const;
```

Question 2.4 Écrire une méthode qui renvoie la taille de l'alphabet

```
/**
 * Get the size of the alphabet
 *
 * Expected complexity: O(1)
 * Max allowed complexity: O(m)
```

```
 */
std::size_t getAlphabetSize() const;
```

Question 2.5 Écrire une méthode pour afficher un automate.

```
/**
 * Print the automaton in a friendly way
 */
void prettyPrint(std::ostream& os) const;
```

Par exemple, l'automate de la figure 1 peut être affiché de la manière suivante :

```
Initial states:
         0 1
Final states:
         1 4
Transitions:
        For state 0:
                For letter a: 1 2 3
                For letter b:
        For state 1:
                For letter a:
                For letter b: 3
        For state 2:
                For letter a: 3
                For letter b: 4
        For state 3:
                For letter a: 3
                For letter b: 4
        For state 4:
                For letter a: 4
                For letter b:
```

Question 2.6 Dans un programme de test, définir l'automate de la figure 1 puis l'afficher.

Question 2.7 (Bonus) Écrire une méthode qui affiche un automate en format DOT [2]. Vous pourrez vous inspirer d'exemples [3]. Vous devrez faire appel au programme `dot(1)` pour réaliser le rendu de votre automate dans un fichier image.

```
/**
 * Print the automaton with respect to the DOT
 * specification
 */
void dotPrint(std::ostream& os) const;
```

2. http://www.graphviz.org/
3. http://www.graphviz.org/Gallery/directed/fsm.html

Travaux Pratiques de Théorie des Langages n°2

Dans ce second TP, le but est de compléter l'ensemble de fonctions pour manipuler un automate et obtenir des informations sur l'automate.

Exercice 3 : Propriétés d'un automate et transformations simples

Question 3.1 Écrire une méthode qui établit si l'automate est déterministe.

```
/**
 * Tell if the automaton is deterministic
 *
 * Expected complexity: O(n * s)
 */
bool isDeterministic() const;
```

Question 3.2 Écrire une méthode qui établit si l'automate est complet.

```
/**
 * Tell if the automaton is complete
 *
 * Expected complexity: O(n * s)
 */
bool isComplete() const;
```

Question 3.3 Écrire une méthode qui complète un automate.

```
/**
 * Make the automaton complete
 *
 * Expected complexity: O(n)
 */
void makeComplete();
```

Question 3.4 Écrire une méthode qui transforme un automate en son complémentaire.

```
/**
 * Transform the automaton to the complement
 *
 * The automaton must be deterministic and complete.
 *
 * Expected complexity: O(n + m)
 */
void makeComplement();
```

Travaux Pratiques de Théorie des Langages n°3

Dans ce troisième TP, le but est d'implémenter une fonction qui test si le langage accepté par l'automate est le langage vide. Pour ce faire, il est nécessaire de voir l'automate comme un graphe et de déterminer s'il existe un chemin entre un des états initiaux et un des états finaux. Si c'est le cas, c'est qu'il existe au moins un mot dans le langage. Dans le cas contraire, le langage est vide.

Vous aurez donc besoin de pouvoir parcourir l'automate comme un graphe, soit en construisant explicitement un graphe à partir de l'automate, soit en utilisant directement l'automate sans prendre en compte les étiquettes sur les transitions. Savoir s'il existe un chemin dans un graphe revient à faire un parcours en profondeur à partir d'un état initial et à voir si on a atteint un état final. Pour rappel, voici l'algorithme de parcours en profondeur d'un graphe :

function DepthFirstSearch(G, s)
  Visited(s) ← `true`
  for u **in** adjacent(G, s) **do**
    if not Visited(u) **then**
      DepthFirstSearch(G, u)
    end if
  end for
end function

Exercice 4 : Test du vide

Question 4.1 Écrire une méthode qui détermine si un automate accepte le langage vide. Pour rappel, le langage accepté par un automate est non vide si et seulement s'il existe un chemin allant d'un état initial à un état final.

```
/**
 * Check if the language of the automaton is empty
 *
 * Expected complexity: O(n)
 */
bool isLanguageEmpty() const;
```

Exercice 5 : Suppression des états inutiles

En utilisant le parcours en profondeur, on peut maintenant déterminer les état inutiles et les supprimer.

Question 5.1 Écrire une méthode qui supprime tous les états qui ne sont pas accessibles.

```
/**
 * Remove non-accessible states
 *
 * Expected complexity: O(n)
 */
void removeNonAccessibleStates();
```

Question 5.2 Écrire une méthode qui supprime tous les états qui ne sont pas co-accessibles.

```
/**
 * Remove non-co-accessible states
 *
 * Expected complexity: O(n)
 */
void removeNonCoAccessibleStates();
```

Travaux Pratiques de Théorie des Langages n°4

Dans ce quatrième TP, le but est de déterminer si l'intersection de deux langages acceptés par des automates est non-vide.

Exercice 6 : Produit d'automate

Pour pouvoir calculer l'intersection, il faut réaliser le produit synchronisé de deux automates.

Pour implémenter le produit d'automates, la difficulté réside dans le codage des états du produits par un seul entier (et non un couple). Supposons que $\mathcal{A}_1$ soit un automate à n_1 états et $\mathcal{A}_2$ un automates à n_2 états. L'automate produit de ces deux automates aura $n_1 * n_2$ états. On propose le codage suivant : l'état (q_1, q_2) du produit sera codé en pratique par le numéro d'état $q_1 * n_2 + q_2$. Réciproquement, un état codé par l'entier r correspondra dans le produit au couple $(r/n_2, r\%n_2)$.

Évidemment, dans notre cas, les états ne sont pas numérotés entre 0 et $n-1$. Il faut donc trouver une correspondance entre les états et un indice de manière à pouvoir appliquer l'astuce précédente.

Question 6.1 Écrire une méthode de classe qui crée un automate produit à partir de deux automates.

```
/**
 * Create the product of two automata
 *
 * The resulting alphabet is the intersection of the
 * two alphabets
 *
 * Expected complexity: O(n_1 * n_2)
 */
static Automaton createProduct(const Automaton& lhs,
    const Automaton& rhs);
```

Question 6.2 Écrire une méthode qui détermine si l'intersection entre deux automates est vide ou pas.

```
/**
 * Tell if the intersection with another automaton is
 * empty
 */
bool hasEmptyIntersectionWith(const Automaton& other)
    const;
```

Travaux Pratiques de Théorie des Langages n°5

Dans ce cinquième TP, le but est d'implémenter l'algorithme de déterminisation d'un automate.

Exercice 7 : Lecture d'un mot

Pour déterminiser un automate, il sera nécessaire de savoir faire une dérivation avec des ensembles d'états. Cette opération de base est également utile pour lire un mot et déterminer si ce mot appartient au langage accepté par l'automate.

Question 7.1 Écrire une méthode qui lit un mot et calcul l'ensemble d'état issu de la dérivation avec ce mot.

```
/**
 * Read the string and compute the state set after
 * traversing the automaton
 */
std::set<int> readString(const std::string& word)
   const;
```

Question 7.2 Écrire une méthode qui détermine si un mot appartient au langage accepté par l'automate. Cette méthode peut servir pour tester vos algorithmes de création d'automates.

```
/**
 * Tell if the word is in the language accepted by the
 * automaton
 */
bool match(const std::string& word) const;
```

Exercice 8 : Déterminisation d'un automate

La difficulté dans la déterminisation d'un automate est de numéroter les états de l'automate déterminisé qui sont des ensembles d'états de l'automate initial. Pour cela, il faut donner à chaque ensemble rencontré lors de la déterminisation un numéro et garder une table de correspondance à jour.

Question 8.1 Écrire une méthode de classe qui crée un automate déterministe à partir d'un automate non-déterministe.

```
/**
 * Create a deterministic automaton from another
 * possibly non-deterministic automaton
 */
static Automaton createDeterministic(const Automaton&
   automaton);
```

Question 8.2 Écrire une méthode qui détermine si un langage accepté par un automate est inclus dans un autre langage accepté par un autre automate. Pour rappel, $A \subset B \iff A \cap \overline{B} = \varnothing$

```
/**
 * Tell if the langage accepted by the automaton is
 * included in the language accepted by the other
 * automaton
 */
bool isIncludedIn(const Automaton& other) const;
```

Travaux Pratiques de Théorie des Langages n°6

Ce sixième TP ne doit être abordé que si l'ensemble des cinq premiers TP est terminé et correct. Il comporte deux exercices.

Exercice 9 : Minimisation d'un automate

Le but de cet exercice est d'implémenter un des algorithmes de minimisation d'automate. Il y a plusieurs manières de procéder : l'algorithme de Moore vu en cours, l'algorithme de Brzozowski ou l'algorithme d'Hopcroft. Des ressources sur ces deux algorithmes sont données sur MOODLE. L'algorithme de Brzozowski est assez facile à implémenter compte tenu de tout ce que vous avez déjà implémenté avant. L'algorithme d'Hopcroft est en revanche plus difficile conceptuellement et algorithmiquement, il doit être abordé en dernier par les plus rapides uniquement.

Question 9.1 Écrire une méthode de classe qui minimise un automate par l'algorithme de Moore.

```
/**
 * Create an equivalent minimal automaton with the
 * Moore algorithm
 *
 * Expected complexity: O(s n^2)
 */
static Automaton createMinimalMoore(const Automaton&
   automaton);
```

Question 9.2 (Bonus) Écrire une méthode de classe qui minimise un automate par l'algorithme de Brzozowski.

```
/**
 * Create an equivalent minimal automaton with the
 * Brzozowski algorithm
 */
static Automaton createMinimalBrzozowski(const
   Automaton& automaton);
```

Question 9.3 (Bonus) Écrire une méthode de classe qui minimise un automate par l'algorithme d'Hopcroft.

```
/**
 * Create an equivalent minimal automaton with the
 * Hopcroft algorithm
 *
 * Expected complexity: O(s n log n)
 */
static Automaton createMinimalHopcroft(const Automaton
   & automaton);
```

Exercice 10 : Gestion des transitions instantanées

Le but de cet exercice est de prendre en compte les ε-transitions. Dans la suite, une ε-transition sera représentées par le caractère NUL de code ASCII 0.

Question 10.1 Écrire une méthode de classe qui supprime les ε-transitions d'un automate.

```
/**
 * Create an equivalent automaton with the epsilon
 * transition removed
 */
static Automaton createWithoutEpsilon(const Automaton&
    automaton);
```