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 suite de tests unitaires pour cette bibliothèque. 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 définir un ensemble de tests unitaires pour cette bibliothèque. Vous utiliserez le framework de test unitaire Google Test. Il est fortement recommandé d'écrire vos tests avant d'écrire votre implémentation.

Une archive contenant un squelette de code pour la bibliothèque 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. Vous ne devez pas ajouter de fichier dans ce projet, ni renommer un fichier.

De 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

Vous aurez à rendre les fichiers `Automaton.h`, `Automaton.cc` et `testfa.cc`, dans un répertoire `automate`, le tout dans une archive `automate.tar.gz`. Le script `prepare-archive.sh` permet de créer l'archive correctement dans le répertoire courant.

Vous serez évalué de deux manières :

1. sur votre implémentation de la classe `Automaton` qui sera testée avec des tests très complets (et non-diffusés) ;
2. sur vos tests qui seront passés sur une implémentation correcte de la classe `Automaton`, ainsi que sur des mutants pour vérifier que vous détectez les implémentations erronées.

Une partie de cette évaluation sera effectuée à l'aide d'une moulinette impitoyable, il est donc primordial de bien respecter les interfaces données ainsi que les consignes. En particulier, vos tests ne doivent pas couvrir les questions bonus. L'autre partie de cette évaluation sera réalisée par vos encadrants qui

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

regarderont la propreté de votre code, le choix de vos structures et la complexité de vos algorithmes.

Le projet est à faire en monôme.

Triche

La frontière est mince entre *l'oubli* de sources et le plagiat. N'oubliez rien, ne trichez pas. Voler des idées, maquiller des résultats ou mal citer ses sources sont des péchés mortels en recherche. Tricher, c'est reconnaître qu'on va devoir changer de métier, car ceux pris à tricher en science n'ont pas de seconde chance.

En revanche, il est conseillé de discuter du projet et d'échanger des idées avec tous vos collègues. Pour ne pas franchir la ligne jaune, **ne lisez jamais de code écrit par un autre étudiant, et ne permettez pas aux autres étudiants de lire votre code**. Vous ne pouvez rendre que du code écrit par vous-même, et vous devez détailler brievement vos sources d'inspiration sur internet dans les commentaires de votre code. Soyez spécifique : si par exemple vous avez trouvé des réponses sur Stack Overflow, pointez les pages utilisées.

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.

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

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

Concernant l'alphabet, toute lettre ASCII qui possède une représentation graphique peut être utilisée (Indice : `isgraph(3)`). Les lettres sont donc représentées à l'aide du type **char**. De plus, un ε est défini par la constante `fa::Epsilon`.

Un état est 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 faut prendre garde à ne pas pouvoir ajouter deux fois la même transition.

Les automates pris en paramètres ainsi que les automates retournés par les fonctions sont toujours des *automates valides*. Un automate valide est un automate avec un alphabet non-vide et un ensemble d'états non-vide. En particulier, il est inutile de faire des tests qui appellent des fonctions avec des automates non valides. Vous pouvez par exemple mettre des `assert` au début des fonctions concernées pour vous assurer que l'API est utilisée correctement et que l'automate d'entrée est bien valide.

Dans vos tests, les automates considérés ne doivent pas avoir d'ε-transition. Ce problème fait l'objet du dernier exercice du TP et il est optionnel.

Exercice 2: Création d'un automate

Le but de cet exercice est de réaliser les premières fonctions de gestion d'un automate. Il est recommandé d'écrire les tests associés à ces fonctions en premier de manière à ce que, si vous changez la représentation interne de votre automate, vous puissiez avoir l'assurance que votre nouvelle implémentation est toujours correcte par rapport à vos tests.

Question 2.1 Écrire une fonction qui détermine si un automate est valide.

```
/**
 * Tell if an automaton is valid.
 *
 * A valid automaton has a non-empty set of states and
 * a non-empty set of symbols
 */
bool isValid() const;
```

Question 2.2 Écrire les méthodes de gestion de l'alphabet.

```
/**
 * Add a symbol to the automaton
 *
 * Epsilon is not a valid symbol.
 * Returns true if the symbol was effectively added
 */
bool addSymbol(char symbol);

/**
 * Remove a symbol from the automaton
 *
 * Returns true if the symbol was effectively removed
 */
bool removeSymbol(char symbol);

/**
 * Tell if the symbol is present in the automaton
 */
bool hasSymbol(char symbol) const;

/**
 * Count the number of symbols
 */
std::size_t countSymbols() const;
```

Question 2.3 Écrire les méthodes de gestion des états.

```
/**
 * Add a state to the automaton.
 *
 * By default, a newly added state is not initial and
 * not final.
 * Returns true if the state was effectively added and
 * false otherwise.
 */
bool addState(int state);

/**
 * Remove a state from the automaton.
 *
 * The transitions involving the state are also
 * removed.
 * Returns true if the state was effectively removed
    and
 * false otherwise.
 */
bool removeState(int state);

/**
 * Tell if the state is present in the automaton.
 */
bool hasState(int state) const;

/**
 * Compute the number of states.
 */
std::size_t countStates() const;
```

Question 2.4 Écrire des méthodes pour gérer les états initiaux et finaux.

```
/**
 * Set the state initial.
 */
void setStateInitial(int state);

/**
 * Tell if the state is initial.
 */
bool isStateInitial(int state) const;

/**
 * Set the state final.
 */
void setStateFinal(int state);
```

```cpp
/**
 * Tell if the state is final.
 */
bool isStateFinal(int state) const;
```

Question 2.5 Écrire des méthodes de gestion des transitions.

```cpp
/**
 * Add a transition
 *
 * Returns true if the transition was effectively
 * added and false otherwise.
 * If one of the state or the symbol does not exists,
 * the transition is not added.
 */
bool addTransition(int from, char alpha, int to);

/**
 * Remove a transition
 *
 * Returns true if the transition was effectively
 * removed and false otherwise.
 */
bool removeTransition(int from, char alpha, int to);

/**
 * Tell if a transition is present.
 */
bool hasTransition(int from, char alpha, int to) const
    ;

/**
 * Compute the number of transitions.
 */
std::size_t countTransitions() const;
```

Question 2.6 Écrire une méthode pour afficher un automate.

```cpp
/**
 * 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:
```

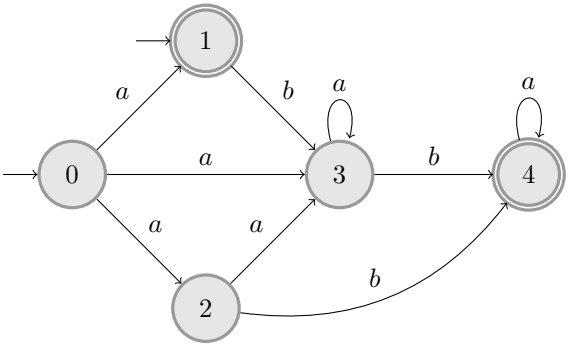

FIGURE 1 – Automate d'exemple

```
         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 2.7 (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.

À partir de ce point, toutes les méthodes doivent être appelées sur des automates valides. Ce qui veut dire :

1. vous n'avez pas à gérer le cas d'automates non-valides dans les méthodes qui suivent, vous pouvez donc insérer `assert(isValid())` au début de toutes vos méthodes ;
2. vous n'avez pas à tester ces méthodes sur des automates non-valides, en particulier vous ne devez pas vérifier qu'il y a bien un `assert` au début de la méthode.

Pour rappel, les automates passés en paramètres sont également valides et les mêmes remarques s'appliquent à ceux-ci.

Exercice 3: Propriétés d'un automate

Question 3.1 Écrire une méthode qui vérifie s'il y a une ε-transition.

```
/**
 * Tell if the automaton has one or more
 * epsilon-transition
 */
bool hasEpsilonTransition() const;
```

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

```
/**
 * Tell if the automaton is deterministic
 */
bool isDeterministic() const;
```

Question 3.3 Écrire une méthode qui établit si l'automate est complet.

```
/**
 * Tell if the automaton is complete
 */
bool isComplete() const;
```

Exercice 4: Transformations simples d'un automate

Question 4.1 Écrire une méthode qui crée un automate complet équivalent à un automate donné.

```
/**
 * Create a complete automaton, if not already
 * complete
 */
```

```
static Automaton createComplete(const Automaton&
    automaton);
```

Question 4.2 Écrire une méthode qui crée un automate qui accepte le complémentaire du langage accepté par l'automate donné. Cette fonction nécessite une fonction qui se trouve plus loin dans le TP mais vous pouvez dès maintenant implémenter l'essentiel de la fonction et faire des tests sur cette partie. Vous compléterez vos tests par la suite.

```
/**
 * Create a complement automaton
 */
static Automaton createComplement(const Automaton&
    automaton);
```

Question 4.3 Écrire une méthode qui crée un automate qui accepte le langage miroir du langage accepté par l'automate donné. Soit $w = w_1 w_2 \dots w_{n-1} w_n$ un mot sur un alphabet Σ, on définit w^R le mot miroir de w par $w^R = w_n w_{n-1} \dots w_2 w_1$. Soit L un langage, on définit L^R le langage miroir par $L^R = \{w^R, w \in L\}$.

```
/**
 * Create a mirror automaton
 */
static Automaton createMirror(const Automaton&
    automaton);
```

Exercice 5: 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 5.1 Écrire une méthode qui fait une dérivation depuis un ensemble d'états avec une lettre de l'alphabet.

```
/**
 * Make a transition from a set of states with a
     character.
 */
std::set<int> makeTransition(const std::set<int>&
    origin, char alpha) const;
```

Question 5.2 Écrire une méthode qui lit un mot et calcul l'ensemble d'états 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 5.3 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. **Elle est très utilisée dans la suite de tests qui sert à vous évaluer !**

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

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 6: Test du vide

Question 6.1 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
 */
bool isLanguageEmpty() const;
```

Exercice 7: Suppression des états inutiles

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

Question 7.1 Écrire une méthode qui supprime tous les états qui ne sont pas accessibles.

```
/**
 * Remove non-accessible states
 */
void removeNonAccessibleStates();
```

Question 7.2 Écrire une méthode qui supprime tous les états qui ne sont pas co-accessibles.

```
/**
 * Remove non-co-accessible states
 */
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 8: Produit d'automate

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

La difficulté de cette fonction est de numéroter les états de l'automate produit qui sont des paires d'états des automates initiaux. Pour cela, il faut donner à chaque paire rencontrée lors de l'algorithme un numéro et garder une table de correspondance à jour.

Question 8.1 Écrire une méthode qui crée un automate qui reconnaît l'intersection de deux langages acceptés par les automates donnés.

```
/**
 * Create the intersection of the languages of two
    automata
 */
static Automaton createIntersection(const Automaton&
   lhs, const Automaton& rhs);
```

Question 8.2 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 9: 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 9.1 Écrire une méthode qui crée un automate déterministe équivalent à l'automate donné.

```
/**
 * Create a deterministic automaton, if not already
 * deterministic
 */
static Automaton createDeterministic(const Automaton&
   automaton);
```

Question 9.2 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 10: Minimisation d'un automate

Le but de cet exercice est d'implémenter 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 derniers 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 10.1 Écrire une méthode qui crée un automate minimal équivalent à l'automate donnée en utilisant l'algorithme de Moore.

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

Question 10.2 Écrire une méthode qui crée un automate minimal équivalent à l'automate donnée en utilisant l'algorithme de Brzozowski.

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

Question 10.3 (Bonus) Écrire une méthode qui crée un automate minimal équivalent à l'automate donnée en utilisant l'algorithme d'Hopcroft.

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

Exercice 11: Gestion des transitions instantanées

Le but de cet exercice est de prendre en compte les ε-transitions.

Question 11.1 (Bonus) Écrire une méthode de classe qui crée un automate équivalent à l'automate donné et sans ε-transition.

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

Annexe 1 : Liste des méthodes sûres

La table 1 donne la liste des méthodes sûres, c'est-à-dire les méthodes qu'on peut appeler et qui donneront un résultat qu'on peut utiliser de manière fiable dans un test après avoir appelé la méthode de la première colonne. Les autres méthodes sont donc considérées comme non-sûres et les utiliser vous expose à avoir des tests faux. Par exemple, après avoir appelé `createComplement`, les seules méthodes que vous pouvez appelez pour faire vos tests sur le résultat sont : `countSymbols` et `hasSymbol`. En particulier, vous ne pouvez pas appeler `hasState`.

La table 2 donne la liste des méthodes toujours sûres, c'est-à-dire celles qu'on peut utiliser de manière fiable dans un test à n'importe quel moment. En particulier, les méthodes `isValid`, `isLanguageEmpty` et `match` sont intéressantes à utiliser parce que leur implémentation ne repose normalement pas sur d'autres méthodes, à l'inverse de `hasEmptyIntersectionWith` et `isIncludedIn` qui nécessitent des méthodes parmi les plus difficiles à implémenter de ce TP.

Annexe 2 : Liste des choses à ne pas faire dans vos tests

Dans vos tests, il **ne** faut **pas** :

— faire appel à une méthode qui n'est pas dans la spécification ;
— faire appel à une méthode bonus (en particulier `dotPrint`) ;
— faire appel à une méthode définie à partir du TP 2 avec un automate non-valide ;
— faire appel à une méthode définies à partir du TP 2 avec un automate contenant une ε transition, sauf pour tester `hasEpsilonTransition`.

Méthode	Méthodes sûres

`createComplete`	`countSymbols`
	`hasSymbol`
	`isComplete`

`createComplement`	`countSymbols` `hasSymbol`,

`createMirror`	`countSymbols`
	`hasSymbol`
	`countStates`
	`countTransitions`

`removeNonAccessibleStates`	`countSymbols`
	`hasSymbol`
	`countStates`
	`hasState`
	`countTransitions`
	`hasTransition`

`removeNonCoAccessibleStates`	`countSymbols`
	`hasSymbol`
	`countStates`
	`hasState`
	`countTransitions`
	`hasTransition`

`createIntersection`	

`createDeterministic`	`countSymbols`
	`hasSymbol`
	`isDeterministic`

`createMinimalMoore`	`countSymbols`
	`hasSymbol`
	`countStates`
	`isDeterministic`
	`isComplete`

`createMinimalBrzozowski`	`countSymbols`
	`hasSymbol`
	`countStates`
	`isDeterministic`
	`isComplete`

TABLE 1 – Méthodes sûres

Méthodes toujours sûres
`isValid`
`isLanguageEmpty`
`hasEmptyIntersectionWith`
`match`
`isIncludedIn`

TABLE 2 – Méthodes sûres