Travaux Pratiques
Théorie des langages
Licence 3 Informatique

Julien Bernard

Table des matières

Le but de cette série de travaux pratiques est de réaliser une application manipulant des automates finis. À la fin, vous aurez implémenté tous les algorithmes vu en cours. 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 en annexe. 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.

Le découpage en TP représente à peu près le temps que vous devez consacrer à chaque partie pendant une séance de 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.

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

La première étape consiste à choisir la structure de données pour représenter un automate. Elle sera nommée `struct fa` *(Finite Automaton)*.

```
struct fa {
  size_t alpha_count;
  size_t state_count;
  // ...
};
```

Un automate sera défini par :
- la taille de l'alphabet utilisé qu'on représentera par un entier de type `size_t` (`alpha_count` dans la structure précédente) ;
- un nombre d'états de l'automate qu'on représentera par un entier de type `size_t` (`state_count` dans la structure précédente) ;
- un ensemble d'états ;
- un ensemble de transitions.

Concernant l'alphabet, dans le cadre de cette série de TP, on utilisera uniquement des lettres minuscules. De plus, l'ordre alphabétique sera respecté, ce qui veut dire que si l'alphabet contient k lettres, ce seront les k premières lettres de l'alphabet qui seront utilisées.

L'ensemble des n états peut être défini de manière implicite, en numérotant les états de 0 à $n-1$ par exemple. Dans ce cas, il faut des structures de données pour savoir si un état est initial ou final, par exemple à l'aide de tableaux.

```
struct fa {
  // ...
  bool *initial_states;
  bool *final_states;
};
```

On peut aussi définir les états de manière explicite, en créant une structure adéquate qui contiendra les informations sur le fait que l'état soit initial ou final. Dans ce cas, un état est encore représenté par un entier entre 0 et $n-1$ qui est l'indice de l'état dans le tableau.

```
struct state {
  bool is_initial;
  bool is_final;
  // ...
};

struct fa {
  // ...
```

```
  struct state *states;
};
```

Enfin, concernant les transitions, il y a de multiples manières de les représenter. Il faut d'abord être capable de représenter un ensemble d'états. On a principalement deux choix : un tableau dynamique et une liste chaînée. Dans les deux cas, il convient de conserver la structure triée de manière à ne pas inclure deux fois le même état dans l'ensemble. Maintenir la structure triée permettra aussi de faciliter le test d'égalité entre deux ensembles d'états.

```
// dynamic array

struct state_set {
  size_t size;
  size_t capacity;
  size_t *states;
};

// linked list

struct state_node {
  size_t state;
  struct state_node *next;
}

struct state_set {
  struct state_node *first;
};
```

Ensuite, une fois qu'on a représenté un ensemble d'états, on peut représenter un ensemble de transitions avec un tableau à deux dimensions d'ensembles d'états. La première dimension représente les états de départ, tandis que la seconde dimension représente les lettres de l'alphabet. Et l'ensemble d'états représente les états d'arrivée. Là encore plusieurs choix sont possibles.

On peut utiliser un tableau de tableaux. Il est alors nécessaire de réaliser une première allocations pour le tableau principal d'une taille `state_count` puis, dans chaque case de ce tableau, une seconde allocation pour les tableaux secondaires d'une taille `alpha_count`. L'accès à cette structure est alors la même qu'un tableau à deux dimensions statique.

```
struct fa {
  // ...
  struct state_set **transitions;
};
```

On peut également utiliser un tableau à une seule dimension qui aura alors une taille $\texttt{state\_count} * \texttt{alpha\_count}$. Là, il n'y a qu'une seule allocation à faire, mais l'accès est un peu plus complexe. Pour l'état i et la lettre j, il faut considérer la case $i * \texttt{alpha\_count} + j$.

```
struct fa {
```

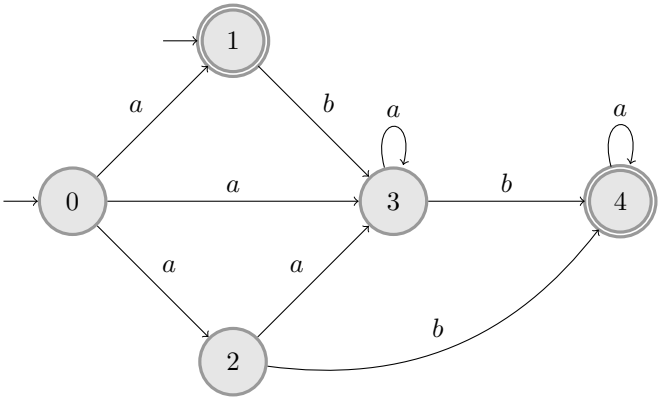

FIGURE 1 – Automate d'exemple

```
  // ...
  struct state_set *transitions;
};
```

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.

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. Pour cela, vous créerez un fichier d'entête `fa.h` dans lequel vous déclarerez votre structure ainsi que toutes les fonctions demandées par la suite, puis vous implémenterez ces fonctions dans un fichier `fa.c`. Enfin, vous réaliserez un petit programme de démonstration de vos fonctions dans un fichier `testfa.c`. Vous n'oublierez pas de créer un `Makefile` pour automatiser la compilation de votre projet.

Question 2.1 Écrire une fonction qui crée un automate.

```
void fa_create(struct fa *self,
                        size_t alpha_count, size_t state_count);
```

Question 2.2 Écrire une fonction qui détruit un automate. Toutes les structures allouées doivent être désallouées proprement.

```
void fa_destroy(struct fa *self);
```

Question 2.3 Écrire deux fonctions pour rendre un état initial et final.

```
void fa_set_state_initial(struct fa *self, size_t state);
void fa_set_state_final(struct fa *self, size_t state);
```

Question 2.4 Écrire une fonction pour ajouter une transition à l'automate.

```
void fa_add_transition(struct fa *self,
                            size_t from, char alpha, size_t to);
```

Question 2.5 Écrire une fonction pour afficher un automate.

```
void fa_pretty_print(const struct fa *self, FILE *out);
```

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

```
Initial states:
         0 1
Final states:
         0 1
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:
                For letter b: 3 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 et le détruire.

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

```
void fa_dot_print(const struct fa *self, FILE *out);
```

1. `http://www.graphviz.org/`
2. `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 : Gestion d'un automate et propriétés

Question 3.1 Écrire une fonction pour supprimer une transition de l'automate.

```
void fa_remove_transition(struct fa *self,
                          size_t from, char alpha, size_t to);
```

Question 3.2 Écrire une fonction pour supprimer un état.

```
void fa_remove_state(struct fa *self, size_t state);
```

Question 3.3 Écrire une fonction qui compte le nombre de transition d'un automate.

```
size_t fa_count_transitions(const struct fa *self);
```

Question 3.4 Écrire une fonction qui établit si l'automate est déterministe.

```
bool fa_is_deterministic(const struct fa *self);
```

Question 3.5 Écrire une fonction qui établit si l'automate est complet.

```
bool fa_is_complete(const struct fa *self);
```

Question 3.6 Écrire une fonction qui complète un automate.

```
void fa_make_complete(struct fa *self);
```

Question 3.7 Écrire une fonction qui fusionne deux états.

```
void fa_merge_states(struct fa *self, size_t s1, size_t s2);
```

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.

Exercice 4 : Test du vide

Pour implémenter le test du vide, il va falloir utiliser un graphe créé à partir de l'automate.

Question 4.1 Proposer une structure de données pour représenter un graphe orienté à l'aide d'une liste d'adjacence.

```
struct graph {
  // ...
};
```

Question 4.2 Écrire une fonction qui réalise un parcours en profondeur du graphe à partir d'un état. La fonction retournera un tableau de booléens (initialement tous à `false`) qui indiquera quels sont les états qui ont été visités pendant le parcours.

```
void graph_depth_first_search(const struct graph *self,
                                    size_t state, bool *visited);
```

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

Question 4.3 Écrire une fonction qui détermine si un chemin existe entre deux états.

```
bool graph_has_path(const struct graph *self,
                                        size_t from, size_t to);
```

Question 4.4 Écrire une fonction qui construit un graphe à l'aide d'un automate, sans tenir compte ni des états initiaux et finaux, ni des étiquettes des transitions. Le paramètre `inverted` indique si le graphe doit être inversé ou non, c'est-à-dire avec les arcs inversés ou non par rapport à l'automate.

```
void graph_create_from_fa(struct graph *self,
                            const struct fa *fa, bool inverted);
```

Question 4.5 Écrire une fonction qui détruit un graphe.

```
void graph_destroy(struct graph *self);
```

Question 4.6 Justifier que 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.

Question 4.7 Écrire une fonction qui détermine si un automate accepte le langage vide.

```
bool fa_is_language_empty(const struct fa *self);
```

Exercice 5 : Suppression des états inutiles

À l'aide des fonctions définies précédemment, on peut maintenant déterminer les état inutiles et les supprimer.

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

```
void fa_remove_non_accessible_states(struct fa *self);
```

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

```
void fa_remove_non_co_accessible_states(struct fa *self);
```

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 $(r/n_2, r\%n_2)$.

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

```
void fa_create_product(struct fa *self,
                       const struct fa *lhs, const struct fa *rhs);
```

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

```
bool fa_has_empty_intersection(
                       const struct fa *lhs, const struct fa *rhs);
```

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 : 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 7.1 Écrire une fonction qui crée un automate déterministe à partir d'un automate non-déterministe.

```
void fa_create_deterministic(struct fa *self,
                                          const struct fa *nfa);
```

Question 7.2 Écrire une fonction qui détermine si un langage accepté par un automate est inclus dans un autre langage accepté par un autre automate. Pour rappel, $A \subset B \iff A \cap \overline{B} = \varnothing$

```
bool fa_is_included(const struct fa *lhs, const struct fa *rhs);
```

Travaux Pratiques de Théorie des Langages n°6

Dans ce sixième TP, le but est d'implémenter des algorithmes de minimisation d'automate.

Exercice 8 : Minimisation d'un automate

Il y a plusieurs manières de procéder : une méthode naïve qui va chercher directement les états qui sont équivalents par la congruence de Nérode, puis l'algorithme de Moore qui permet de calculer itérativement les classes d'équivalences.

Question 8.1 Écrire une fonction qui détermine si deux états sont équivalents par la congruence de Nérode.

```
bool fa_are_nerode_equivalent(const struct fa *self,
                                          size_t s1, size_t s2);
```

Question 8.2 Écrire une fonction qui minimise un automate en fusionnant les états équivalents par la congruence de Nérode.

```
void fa_create_minimal_nerode(struct fa *self,
                                         const struct fa *other);
```

Question 8.3 Écrire une fonction qui minimise un automate par l'algorithme de Moore.

```
void fa_create_minimal_moore(struct fa *self,
                                         const struct fa *other);
```