

TP 3 : Automates

Exercice 1 : Automates finis déterministes

On représente en C un automate fini déterministe par le type suivant :

```
struct AFD {  
    int Q;  
    int Sigma;  
    int qI;  
    bool* finaux;  
    int** delta;  
};  
  
typedef struct AFD afd;
```

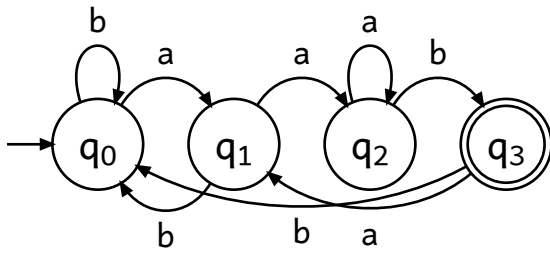
Si A est un automate représentant $A = (Q, \Sigma, q_I, F, \delta)$, alors :

- A.Q représente $|Q|$ et A.Sigma représente $|\Sigma|$;
- $Q = \llbracket 0, |Q| - 1 \rrbracket$ et $\Sigma = \llbracket 0, |\Sigma| - 1 \rrbracket$;
- A.qI représente l'état initial q_I ;
- A.finaux est un tableau de booléens de taille $|Q|$ tel que $q \in F \Leftrightarrow A.finaux[q] == \text{true}$;
- A.delta représente un tableau de tableaux d'entiers tel que si $q \in Q$ et $a \in \Sigma$, alors $A.delta[q][a]$ vaut $\delta(q, a)$ s'il existe, ou -1 sinon (on utilise ici une définition utilisant une fonction de transition et non un ensemble de transitions).

On supposera pour l'ensemble du sujet que les mots sont des chaînes de caractères contenant les 26 lettres de l'alphabet en minuscule. On rappelle que si a est un caractère, alors $a - 97$ est un entier compris entre 0 et 25.

- 1) Télécharger le fichier TP03.c.
- 2) Écrire une fonction `void liberer_afd(afd A)` qui libère la mémoire occupée par un AFD A. On pensera à libérer la mémoire occupée par **tous** les tableaux.
- 3) Écrire une fonction `afd init_afd(int Q, int Sigma, int qI)` qui crée un AFD sans transition et aucun état final dont la taille de Q, la taille de Σ et l'état initial sont donnés en arguments.
- 4) Écrire une fonction `void ajout_transition_afd(afd A, int q, char a, int p)` qui modifie l'automate A en ajoutant une transition $\delta(q, a) = p$. On pensera à transformer le caractère a (dont la valeur entière est comprise entre 97 et 122) en un entier de $\llbracket 0, |\Sigma| - 1 \rrbracket$.

5) Décommenter les lignes commentées du main pour créer un automate A1 correspondant à l'automate suivant :



6) En utilisant les fonctions précédentes, créer dans le main un AFD A2 reconnaissant les mots commençant par a dont la taille est un multiple de 3, sur l'alphabet $\{a, b\}$.

7) Écrire une fonction `int delta_etoile_afd(afd A, int q, char* u)` qui prend en arguments un automate A , un état q et un mot u et renvoie l'état dans lequel on arrive en lisant le mot u à partir de l'état q (on note parfois cela $\delta^*(q, u)$).

La fonction renverra -1 si cet état n'est pas défini.

8) En déduire une fonction `bool reconnu_afd(afd A, char* u)` qui détermine si un mot u est reconnu par un automate A .

9) Tester la fonction précédente sur les deux automates précédents :

- vérifier que $u = abbabbabaab \in L(A_1)$ et $v = baababbbba \notin L(A_1)$;
- vérifier que $u \notin L(A_2)$, $v \notin L(A_2)$ et $w = aaabababb \in L(A_2)$.

10) Écrire une fonction `afd completer_afd(afd A)` qui prend en paramètre un AFD A et qui renvoie l'automate obtenu en complétant A .

11) Écrire une fonction `bool accessible_afd(afd A, int q)` qui détermine si l'état q de l'AFD A est accessible.

12) Écrire une fonction `bool coaccessible_afd(afd A, int q)` qui détermine si l'état q de l'AFD A est co-accessible.

13) Écrire une fonction `bool utile_afd(afd A, int q)` qui détermine si l'état q de l'AFD A est utile.

14) Écrire une fonction `afd emonder_afd(afd A)` qui prend en paramètre un AFD A et qui renvoie l'automate obtenu en émondant cet A .

Exercice 2 : Automates finis non déterministes

15) Télécharger les fichiers `dicts.c` et `dicts.h` et les placer dans le même répertoire que `TP03.c`.

On s'intéresse dans cet exercice à des automates non déterministes. On les définit en utilisant les types suivant :

```
struct Liste {
    int val;
    struct Liste* suivant;
};

typedef struct Liste liste;

struct AFND {
    int taille_Q;
    int taille_Sigma;
    bool* initiaux;
    bool* finaux;
    liste*** delta;
};

typedef struct AFND afnd;
```

Les différences avec le type d'AFD utilisé précédemment sont les suivantes : si B est un AFND, $B = (Q, \Sigma, I, F, \delta)$, alors :

- $B.\text{initiaux}$ est un tableau de booléens de taille $|Q|$ tel que $q \in I \Leftrightarrow B.\text{initiaux}[q] == \text{true}$;
- $B.\text{delta}$ représente un tableau de tableaux de listes tel que si $q \in Q$ et $a \in \Sigma$, alors $B.\text{delta}[q][a]$ est une liste chaînée contenant les états de $\delta(q, a)$.

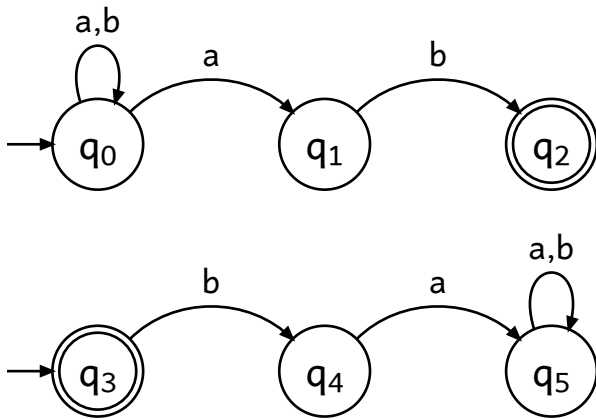
Attention, le type `liste` est déjà défini dans le fichier `dicts.h`, et on trouve dans `dicts.c` les fonctions `liste* cons(int x, liste* lst)` qui crée une nouvelle liste étant donnée une tête et une queue et `void liberer_liste(liste* lst)` qui libère l'espace mémoire occupé par une liste.

On se donne, comme pour les AFD, des fonctions de libération de la mémoire `liberer_afnd` et d'initialisation d'automate sans transition, sans état initial et sans état final `init_afnd`.

Avant de tester le code, on décommentera la ligne 7 (`#include "dicts.h"`) et les lignes à partir de la définition du type AFND et avant la fonction `main`.

16) Écrire une fonction `void ajout_transition_afnd(afnd B, int q, char a, int p)` qui modifie l'automate B et rajoute une transition $q \xrightarrow{a} p$. On supposera que cette transition n'existe pas déjà.

17) Décommenter les lignes commentées du main pour créer un automate B1 correspondant à l'automate suivant :



18) En utilisant les fonctions précédentes, créer dans le main un AFND B2 reconnaissant les mots contenant un a en avant-avant-dernière position, sur l'alphabet $\{a, b\}$.

19) Écrire une fonction `bool* delta_etats(afnd B, bool* X, char a)` qui prend en argument un AFND B, un tableau de booléens X représentant un ensemble d'états $X \subset Q$ et une lettre a et renvoie un tableau de booléens représentant $\delta(X, a)$.

20) En déduire une fonction `bool* delta_etoile_afnd(afnd B, bool* X, char* u)` qui prend en argument un AFND B, un tableau de booléens X représentant un ensemble d'états $X \subset Q$ et un mot u et renvoie un tableau de booléens représentant $\delta^*(X, u)$.

21) En déduire une fonction `bool reconnu_afnd(afnd B, char* u)` qui détermine si un mot u est reconnu par un AFND B.

22) Tester la fonction précédente sur les automates B1 et B2 :

- vérifier que $u \in L(B_1)$, $v \in L(B_1)$ et $w \notin L(B_1)$;
- vérifier que $u \in L(B_2)$, $v \notin L(B_2)$ et $w \in L(B_2)$.

Exercice 3 : Détermination

On cherche à écrire une fonction de détermination d'automate non déterministe. Pour ce faire, on choisit d'assimiler une partie $X \subset Q$ à un entier compris entre 0 et $2^{|Q|} - 1$. On utilisera cette numérotation des états de l'automate des parties.

23) Écrire une fonction `int etats_vers_entier(bool* X, int taille_Q)` qui prend en argument un tableau de booléens représentant une partie $X \subset Q$ et un entier correspondant à $|Q|$ et renvoie un entier x de $\llbracket 0, 2^{|Q|} - 1 \rrbracket$ associé à X .

24) Écrire sa fonction réciproque `bool* entier_vers_etats(int x, int taille_Q)`.

25) En déduire une fonction `afd determiniser(afnd B)` qui détermine un automate non déterministe selon la construction de l'automate des parties.

26) Déterminer les automates B_1 et B_2 en vérifiant que les mots reconnus par les résultats sont bien cohérents.

La construction naïve de l'automate des parties crée systématiquement un automate déterministe de taille $2^{|Q|}$. Comme on l'a vu en cours, on peut se contenter de ne garder que les états accessibles depuis l'état initial. On cherche à écrire une nouvelle version de la fonction précédente en suivant ce principe. À cet effet, on se donne une structure de dictionnaire `dict` implémentée par une table de hachage. On dispose des primitives suivantes :

- `void dict_free(dict D)` libère la mémoire occupée par un dictionnaire ;
- `dict create(void)` crée un dictionnaire vide ;
- `int size(dict D)` renvoie la taille d'un dictionnaire ;
- `bool member(dict D, int k)` teste l'appartenance d'une clé à un dictionnaire ;
- `int get(dict D, int k)` renvoie la valeur associée à une clé dans un dictionnaire ;
- `void add(dict* D, int k, int v)` ajoute une association (clé, valeur) à un dictionnaire ;
- `void del(dict* D, int k)` supprime une association d'un dictionnaire ;
- `liste* key_list(dict D)` renvoie une liste chaînée contenant toutes les clés d'un dictionnaire.

On supposera que toutes les opérations sauf la première et la dernière sont en complexité $\mathcal{O}(1)$ en moyenne. Attention, certaines fonctions (celles qui modifient le dictionnaire) prennent en argument un pointeur de dictionnaire `dict*` et pas un dictionnaire `dict`.

27) Écrire une fonction `dict accessibles(afnd B)` renvoyant un dictionnaire dont les clés sont les numéros des parties accessibles de l'automate des parties d'un AFND B et les valeurs sont des numéros consécutifs partant de 0 permettant de renuméroter ces états.

28) En déduire une fonction `afd determiniser2(afnd B)` qui détermine un automate non déterministe en ne représentant que les états accessibles de l'automate des parties.

29) Vérifier la correction de cette fonction sur les automates B_1 et B_2 . Combien d'états contiennent les versions déterminisées ?

30) Quelle est la complexité de cette nouvelle fonction en fonction du nombre d'états accessibles ?

Exercice 4 : Automates finis non déterministes avec transitions instantanées (bonus)

31) Créer un type d'automate fini non déterministe avec transitions instantanées, ainsi que les fonctions d'initialisation, de libération et d'ajout de transitions adéquates.

32) Écrire une fonction qui élimine les transitions instantanées d'un AFND- ϵ

Exercice 5 : Théorème de Kleene (bonus)

33) Créer en OCaml un type permettant de représenter les AFD.

34) Écrire une fonction en OCaml qui implémente l'algorithme de Berry-Sethi.

35) Écrire une fonction en OCaml qui implémente l'algorithme d'élimination des sommets.

Il faudra pour cela créer un type permettant de représenter les automates généralisés.