TP : Automates finis déterministes

Ce TD/TP a pour objectif de programmer en OCaml le calcul par automates finis déterministes. Nous étudierons également quelques constructions vues en cours : complétion d'un automate, complémentaire, automate produit.

1 Calcul d'un afd

Nous typons les automates de manière similaire au cours :

```
type etat = int;;

type alphabet = char list;;

type auto = {
    taille: int;
    init: etat;
    final: etat list;
    trans: (char * etat) list array};;
```

Ce type permet de représenter à la fois les automates finis déterministes et non déterministes.

1. Écrire la fonction `mem : 'a -> 'a list -> bool` testant l'appartenance à une liste.
2. Écrire la fonction `assoc : 'a -> ('a * 'b) list -> 'b` retrouvant dans une liste associative la valeur associée à une clef. On déclenchera l'exception `Not_found` si la clef n'existe pas.
3. En déduire une fonction `lecture_afd : auto -> etat -> string -> etat` tel que `lecture_afd a q u` retourne l'état q' dans lequel on arrive après avoir lu u depuis l'état q. On supposera que l'automate passé en argument est bien un afd. Cette fonction déclenchera `Not_found` en cas de blocage.
4. Écrire une fonction `recon_afd : auto -> string -> bool` déterminant si un mot est reconnu par un afd. Quel est sa complexité dans le pire cas en fonction de $|u|$ et $|\Sigma|$?
5. On suppose $\Sigma = \{a, b\}$. Construire des afd reconnaissant les langages suivants, puis les coder en `Caml` :
   (a) A_1 : Mots commençant par ab
   (b) A_2 : Mots contenant un nombre pair de a
   (c) A_3 : Mots u tels que $|u|_b \equiv 2 \pmod 3$

   On vérifiera que la reconnaissance se déroule comme attendu sur des exemples bien choisis.

2 Quelques constructions classiques

2.1 Complétion d'un automate

Soit A un automate fini déterministe on veut construire un afd A' reconnaissant le même langage. On procédera comme en cours en ajoutant un état puits à l'automate.

1. Réécrire la fonction `map : ('a -> 'b) -> 'a list > 'b list` telle que `map f l` calcule la liste des images des éléments de `l` par `f`.
2. Écrire une fonction `ajoute_fleches : alphabet -> etat -> ((char * etat) list) -> ((char * etat) list)` qui étant donné un alphabet et un état puits, ajoute à une liste de flèches les flèches étiquetées par les lettres manquantes en les faisant pointer vers l'état puits.
3. En déduire une fonction `completion : alphabet -> auto -> auto` qui complète un afd étant donné un alphabet.
4. Quelle est la complexité dans le pire cas de la fonction `completion` en fonction de $|\Sigma|$ et de la taille de l'automate ?
5. Compléter l'automate A_1.

2.2 Complémentaire d'un automate

Soit A un afd. On rappelle que pour construire un automate reconnaissant $\Sigma^* - L(A)$ il suffit de compléter A puis d'inverser les états finaux et non finaux.

1. Écrire une fonction `comp_int : int -> int list -> int list` telle que `comp_int n l` calcule une liste des entiers de $[|0, n|]$ n'appartenant pas à l.
2. Écrire la fonction `comp : alphabet -> auto -> auto` calculant un automate reconnaissant le complémentaire d'un langage reconnu par un autre automate.
3. Construire avec ce qui précède un afd reconnaissant les mots de Σ^* ne commençant pas par ab.

2.3 Automate produit

On s'intéresse maintenant à l'automate produit de deux afd A_1 et A_2. Pour rappel, l'automate produit consiste à exécuter simultanément A_1 et A_2. Voici sa définition : $A = A_1 \times A_2 = (Q, q_0, F, \delta)$ est défini par

- $Q = Q_1 \times Q_2$
- $q_0 = (q_0^1, q_0^2)$
- $F = F_1 \times F_2$

- $\forall a \in \Sigma,\ \delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$ sous réserve d'existence

Si $|Q_1| = n_1$ et $|Q_2| = n_2$ l'automate produit a donc $n_1 n_2$ états. On aura donc intérêt à considérer une bijection

$$\varphi : [|0, n_1 - 1|] \times [|0, n_2 - 1|] \to [|0, n_1 n_2 - 1|].$$

1. Écrire une fonction `bijection : int -> int -> (((int * int) -> int) * (int -> (int * int))` tel que `bijection n1 n2` renvoie un couple `(phi, psi)` contenant la bijection d'intérêt φ et sa réciproque ψ.
2. Écrire une fonction `produit_cartesien : 'a list -> 'b list -> ('a * 'b) list` telle que `produit_cartesien l1 l2` construit la liste des couples (x_1, x_2) avec $x_1 \in l_1$ et $x_2 \in l_2$. On ne cherchera pas à optimiser la complexité et on pourra utiliser la concaténation de listes.
3. Écrire une fonction `produit_fleches : (int -> (int * int)) -> (char * etat) list -> (char * etat) list -> (char * etat) list` telle que `produit_fleches phi l1 l2` construit la liste associative correspondant aux transitions sortantes de l'état $\varphi(q_1, q_2)$ où `l1` est la liste associative des transitions sortantes de q_1 et `l2` la liste associative des transitions sortantes de q_2.
4. Écrire la fonction `produit : auto -> auto -> auto` réalisant le produit d'automates.
5. En déduire un automate reconnaissant les mots ne commençant pas par ab et contenant un nombre pair de a
6. En déduire de même un automate reconnaissant les mots u tels que $|u|_b \equiv 2 \pmod 3$ et contenant un nombre pair de a.