



Rapport de projet

Déterminisation et minimisation
d'un automate fini non
déterministe



Raffaelle GIANNICO IATIC 4

Rayane HAMMOUMI IATIC 4

Professeur: M. Devan SOHIER

Janvier 2023

Table des matières

Introduction	3
Structures de données	4
Bilan	8

Introduction

La performance est devenue primordiale dans l'informatique, mais les enjeux énergétiques n'ont jamais été aussi importants qu'aujourd'hui.

L'analyse lexicale vise à identifier les mots d'un langage. Elle est la première étape de la traduction d'un programme en exécutable par le compilateur.

Étant donné le nombre de lignes de codes qu'il peut y avoir dans un programme, et le fait que le monde entier compile et exécute des programmes, cette étape se doit d'être optimisée.

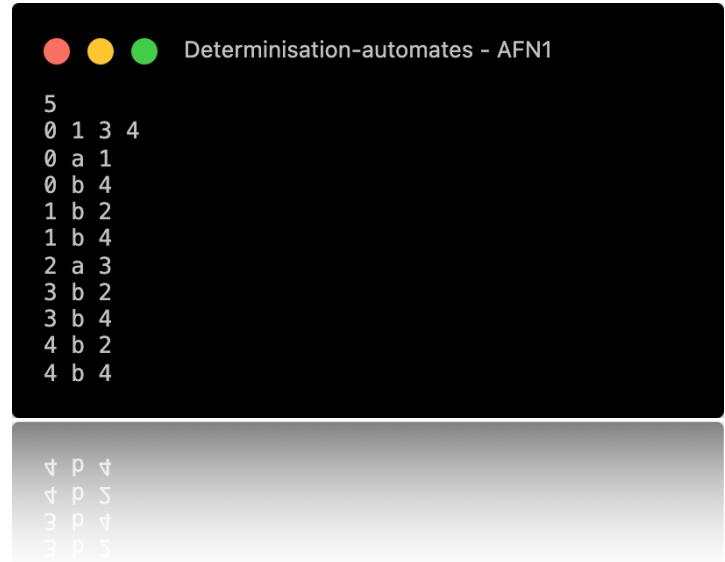
Dans cet esprit, la déterminisation et la minimisation d'un automate permettent au compilateur de déterminer plus rapidement si un mot fait partie du langage reconnu par l'automate en réduisant son nombre de transitions possibles.

Structures de données

AFN

Grâce au format de l'AFN imposé par le sujet, nous avons déjà une méthode de stockage d'automate (dans un fichier).

Nous avons choisi de ne pas stocker à nouveau tout l'automate dans une nouvelle structure de donnée à cette étape car cela ne nous semblait pas nécessaire et plus couteux en mémoire.



Exécution des mots sur l'automate

execute_mots_sur_automate(int argc, char *argv[]) exécute donc les mots contenus dans argv[] en parcourant le fichier d'un automate.

Cette fonction est réutilisable pour un AFD ou un automate minimal déterministe.

Variables utiles:

- char **argv[mot_utilise][i]** contient les mots tapés par l'utilisateur lors du lancement de la commande. mot_utilise commence à 2 et ++ quand on passe au prochain mot. i++ pour passer au prochain caractère, i — pour revenir au caractère précédent
- int **positions_interdites []**: on stocke ftell(fichier) quand on revient à l'état précédent après un "ko" pour éviter de reprendre une transition qu'on a déjà testé et qui ne fonctionne pas

- char **caractere_lu** prend la valeur de fgetc(fichier) pour stocker un état d'arrivée, de départ ou le caractère d'une transition
- int **compteur_positions_interdites**: +1 quand on stocke dans positions_interdites pour conserver l'index de la dernière case remplie
- int **nb_transitions_prises**: commence à 0, +1 quand on prend une transition, - - quand on revient à l'état précédent après un “ko”
- int **historique_etats []** à chaque fois qu'on prend une transition, on stocke l'état de départ pour pouvoir y revenir plus tard
- int **etat_actuel**: commence à 0, devient l'état destinataire quand on prend une transition ou historique_etats[nb_transitions_prises]) quand on revient en arrière

Algo:

On recherche une transition qui part bien de l'état actuel et qui emprunte le caractère recherché. Si une telle transition est trouvée, et que sa position n'est pas dans positions_interdites, on la prend et on la printf().

On relis le fichier depuis le début (fseek(fichier, 0, SEEK_SET) après avoir mis à jour les variables ci-dessus.

Si on finit de parcourir le fichier :

- si nb_transitions_prises>0, alors on revient à une transition précédente et on continue
- si nb_transitions_prises == 0 , alors le mot ne peut pas être accepté et on passe au mot suivant

On réinitialise les variables nécessaires pour la bonne exécution du mot suivant.

AFD

Pour la déterminisation et la minimisation, nous avons choisi de stocker l'automate dans une structure. Cela car nous trouvons plus simple de modifier les valeurs des états et des caractères de l'automate dans une variable plutôt que dans un fichier.

cree_automate() remplit donc une variable de type Automate à partir d'un automate dans un fichier.

Cette variable est ensuite prise en paramètre dans determinise_automate(). Cette fonction modifie les valeurs des champs afin de rendre l'automate déterministe.

Pour cela, on utilise le tableau d'entiers etats_a_fusionner[].

Pour chaque état de l'automate et pour chaque caractère, dès qu'on trouve une transition qui a le même état et caractère que ceux qu'on recherche grâce aux indices des for:

- on remplit indices_transitions_a_supprimer[] et etats_a_fusionner[].
- compteur_transitions_non_deterministes++

Si (compteur_transitions_non_deterministes++) > 1, on cherche le nombre minimum dans les cases remplies.

Dans la liste des transitions, on vient remplacer tous les nombres différents de minimum et contenus dans etats_a_fusionner [] par le minimum.

Pareil dans etatsAccepteurs[] si au moins un entier de etats_a_fusionner[] est dans etatsAccepteurs[].

Enfin, les états d'arrivée et de départ deviennent -1 et le caractère devient '\0' dans transitions donc l'indice est stocké dans indices_transitions_a_supprimer[i] avec i de 1 à compteur_transitions_non_deterministes.

```
typedef struct Transition
{
    int etatDepart;
    char lettre;
    int etatArrivee;
} Transition;

typedef struct Automate
{
    int nbEtats;
    int *etatsAccepteurs;
    Transition *listeTransition;
} Automate;
```

```
} Automate;
```

```
    Transition *listeTransition;
    int *etatsAccepteurs;
    int nbEtats;
```

```
};
```

```
Automate *creerAutomate();
```

```
void determiniserAutomate(Automate *automate);
```

```
void afficherAutomate(Automate *automate);
```

```
void libererAutomate(Automate *automate);
```

```
void lireAutomate(Automate *automate);
```

```
void sauverAutomate(Automate *automate);
```

```
void creerAutomate(Automate *automate);
```

```
void libererAutomate(Automate *automate);
```

```
void lireAutomate(Automate *automate);
```

```
void sauverAutomate(Automate *automate);
```

```
void determiniserAutomate(Automate *automate);
```

```
void afficherAutomate(Automate *automate);
```

```
void libererAutomate(Automate *automate);
```

```
void creerAutomate(Automate *automate);
```

```
void libererAutomate(Automate *automate);
```

```
void determiniserAutomate(Automate *automate);
```

```
void afficherAutomate(Automate *automate);
```

```
void libererAutomate(Automate *automate);
```

```
void creerAutomate(Automate *automate);
```

```
void libererAutomate(Automate *automate);
```

```
void determiniserAutomate(Automate *automate);
```

```
void afficherAutomate(Automate *automate);
```

```
void libererAutomate(Automate *automate);
```

```
void creerAutomate(Automate *automate);
```

```
void libererAutomate(Automate *automate);
```

```
void determiniserAutomate(Automate *automate);
```

```
void afficherAutomate(Automate *automate);
```

```
void libererAutomate(Automate *automate);
```

```
void creerAutomate(Automate *automate);
```

```
void libererAutomate(Automate *automate);
```

```
void determiniserAutomate(Automate *automate);
```

```
void afficherAutomate(Automate *automate);
```

```
void libererAutomate(Automate *automate);
```

```
void creerAutomate(Automate *automate);
```

```
void libererAutomate(Automate *automate);
```

Minimisation

minimise_automate() utilise également un tableau d'entiers pour les groupes d'états (Groupe [])

La fonction minimise_automate prend en paramètre l'automate que l'on veut minimiser.

Elle fait appel à init_minimisation() qui va déterminer le nom de départ des différents groupe.

Afin de stocker ces noms, on utilise le tableau d'entiers groupe[] qui correspond aux noms des différents états. Les indices i correspondent aux états et les groupe[i] correspondent aux noms des différents groupes. S'ils sont la même valeur, c'est qu'il font partie du même groupe.

Nous avons aussi le tableau groupeAvant. Ces deux tableaux vont nous servir pour arrêter l'algorithme si on à ces deux tableaux identiques.

Nous utilisons aussi le tableau alphabet qui est le tableau contenant les caractères de l'alphabet de l'automate, que nous récupérons grâce à une fonction getAlphabet qui parcourt les transitions.

Aussi nous utilisons un tableau etapeMinimisation qui correspond aux différentes étapes de l'exécution de l'algorithme. C'est le tableau qui actualise les transitions. En effet, pour chaque état de départ on va déterminer, quel est son état d'arrivé. Grâce à ce tableau nous pouvons établir toutes les transitions du graphe.

Nous utilisons aussi un tableau de booléen qui nous permet de savoir si à chaque étape de la minimisation nous avons déjà donner un nouveau nom au groupe.

De plus, on utilise une fonction est_ds_tableau_int qui nous indique si un élément est dans le tableau. Celle-ci nous sera utile pour savoir si l'état d'arrivée d'une transition appartient au tableau des accepteurs. C'est utile pour supprimer les transitions qui ne mènent pas à un état accepteur.

Bilan

Ce projet nous a permis de confirmer l'importance d'utiliser un automate minimal déterministe.

En effet, nous avons maintenant la confirmation visuelle que le nombre de tests effectués pour savoir si un mot est accepté sur un automate non déterministe peut être bien plus important que sur un automate minimal déterministe.