Projet - Grammaires et langages

Transformation et minimisation d'automate

Faculté des Sciences et Techniques

Licence 3 Informatique

Encadré par :
Pr Philippe GABORIT

Réalisé par :
Nicolas NAN
Jérémy DRON

1 Algorithme de transformation d'un AFN en AFD

En théorie des automates, il existe plusieurs types d'automates ayant des caractèristique bien défini :

- **AFD [Automate Fini Déterministe] :** Une seule transition au plus pour chaques symboles depuis un état
- **AFN [Automate Fini Non-déterministe] :** Automate pouvant posséder plusieurs transitions pour chaques symboles depuis un état
- **AFN-ϵ [Automate Fini Non-Déterministe avec ϵ transition] :** Automate possédant des transitions avec le symbole du vide

Il est possible de passer d'un type d'automate à un autre en lui appliquant quelques modifications. Pour passer d'un AFN à un AFD, il faut créer des super-états qui seront la combinaisons de plusieurs états. Un état sera final si un des états qui le compose l'est.Notre algorithme ne prendra en entrée que les automates suivant :

A={Q,ϵ,δ,q_0,F}
Q => L'ensembles des états de l'automate avec $|Q| <= 10$
ϵ => L'alphabet de symbole fini avec $\epsilon = a, b$
δ => Transitions de l'automate
q_0 => L'etat initial de l'automate
F => L'ensemble des états finaux

1.1 Représentation d'un automate et structure de données

Dans un premier temps, il nous a fallu pouvoir représenter un automate de manière que notre programme puisse le comprendre et l'utiliser. Pour cela, nous avons réaliser pour chaques symboles une matrice. Ces matrices continennent uniquement des "0" ou des "1" et chaques lignes indiquent l'état de départ d'une transition et chaque colonne l'état d'arrivé.

	0	1	2	3
0	1	1	0	0
1	1	1	1	0
2	1	1	1	0
3	1	1	0	0

Le tableau ci-dessus représente un matrice concernant un symbole d'un automate.

1.2 Principe de l'algorithme

1.3 Programme et execution

2 Algorithme de minimisation d'un automate

A présent, nous cherchons à minimiser un AFD, cest-à-dire que nous cherchons à avoir un automate avec le minimum d'état qui a le même comportement que celui actuel. En effet, certains automates ont des états qui peuvent être combiner entre-eux. Notre algorithme ne prendra en entrée que les automates suivant :

A={Q,ϵ,δ,q_0,F}
Q => L'ensembles des états de l'automate avec $|Q| <= 10$
ϵ => L'alphabet de symbole fini avec $\epsilon = a, b$
δ => Transitions de l'automate
q_0 => L'etat initial de l'automate
F => L'etat final de l'automate

2.1 Représentation d'un automate et structure de données

Premièrement, nous avons cherché une représentation, pour notre AFD, que notre programme pourrait facilement utiliser. Nous nous sommes alors basés de ce que nous avions trouvé dans la première partie.
Nous avons pour chaque symbole de notre alphabet une matrice lui correspondant. Dans le cas précédent notre

matrice contenait l'information de quels états de départ pointaient sur quels états d'arrivés. Dans ce cas présent, c'est l'inverse, nous cherchons à savoir quels états est pointés par quels état. Du au fait, que nous travaillons uniquement avec des AFDs nous pouvons concidérer que :

Soit M = "La matrice des transitions de départ vers arrivé" alors M^t = "La matrice des transitions des arrivés vers les départs"

Par ailleurs, pour minimiser notre automate nous avons utilisé l'algorithme vu en cours. Dans notre programme nous utiliserons une file qui contiendra les couples d'états à vérifier. Nous avons fait le choix de faire notre programme dans le paradygme orienté objet pour une optimisation de la compléxité cognitive. Nous avons réaliser le diagramme de classe de notre programme pour faciliter la compréhension de la structure du programme.

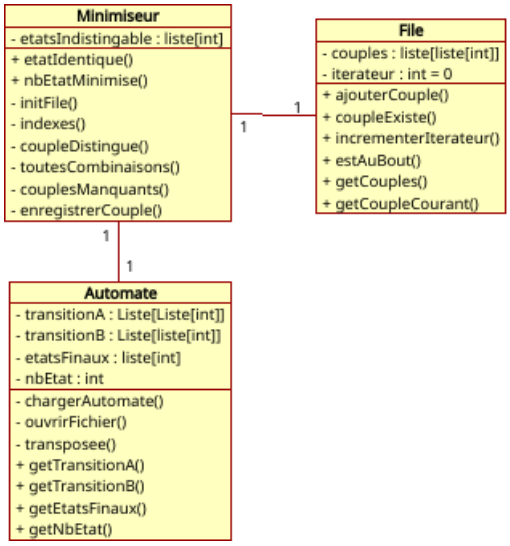

FIGURE 1 – Diagramme de classe de notre minimiseur d'automate

2.2 Principe de l'algorithme

En premier lieu, l'utilisateur doit créer un objet de type *Automate* et un objet de type *Minimiseur* puis il peut utiliser deux méthodes : une permettant de récuperer les états non distingables et l'autre le nombre d'état de l'automate minimisé.

```
1  from minimiseur import *
2  from Automate import *
3
4  def main():
5      automate : Automate = Automate("./automate.txt")
6      minimiseur : Minimiseur = Minimiseur(automate)
7
8      print(f"Les etats identiques de l'automate sont: {minimiseur.etatsIdentique()}")
9      print(f"L'automate minimiser fait {minimiseur.nbEtatMinimise()}")
10
11
12 if __name__ == '__main__':
13     main()
```

Concretement, l'objet *Automate* prend un fichier en entrée qui contient des informations sur l'automate ainsi la représentation de l'automate exprimé plus haut. Le fichier doit contenir sur la premier ligne le nombre d'état de l'automate, sur la deuxieme ligne les états terminaux et finalement les matrices de transitions sur les lignes suivantes. Le fichier peut ressembler à ça :

```
 7
 2
 0 1 0 0 0 0 0
 0 0 0 0 0 1 0
 0 0 0 0 0 0 0
 0 0 0 0 0 0 1
 0 0 1 0 0 0 0
 0 0 0 0 0 1 0
 0 0 0 0 0 1 0
 0 0 0 0 1 0 0
 0 0 1 0 0 0 0
 0 0 1 0 0 0 0
 0 0 0 0 1 0 0
 0 0 0 0 0 1 0
 0 0 0 1 0 0 0
 0 0 1 0 0 0 0
```

C'est grace à ce fichier que l'automate peut être charger dans le programme, donc le respect de la forme est essentiel pour le fonctionnement.

Pour minimiser l'automate, le *minimiseur* va initialiser la file avec les distingués trivialement, i.e. les terminaux et non terminaux. Par la suite, le programme va générer de nouveaux couples d'états distingués qu'il va ajouter en file si ces derniers n'existent pas déjà. Le programme va parcourir toute la file jusqu'à arriver en tête avant de récuperer les états des couples manquants. Pour connaitre le nombre d'état de l'automate final, le *minimiseur* soustrait au nombre d'état actuel le nombre d'état non distingué.

Pour trouver les états distingués engendré par un couple d'état passé en entrée de la fonction, le programme vient recupérer grace à la matrice les états pour lesquels il y a une transition avant de d'enregistrer dans la file toutes les combinaisons d'états non existant. Si un état d'entrée engendre aucun état de sortie, càd si un état ne possede pas de transition pour un symbole de l'alphabet, alors le programme ignore les transitions de ce symbole pour cet état.

```
    ################################
    # Fonction qui donne les nouveaux
    # couples distingues depuis celui entree
    #
    # ENTREE: Un tuple
    # SORTIE: liste entier de tuple
    ################################
    def __coupleDistingue(self, l_etats) -> list[list[int]]:
        couplesSortie :list[list[int]] = []
        coupleTmp : list[int] = [None,None]
        try:
            premierEtat :int = l_etats[0]
            secondEtat :int = l_etats[1]

            coupleTmp[0] = self.__indexes(\
                self.__automate.getTransitionsA()[premierEtat])
            coupleTmp[1] = self.__indexes(\
                self.__automate.getTransitionsA()[secondEtat])
            couplesSortie.extend(self.__toutesCombinaisons(coupleTmp))
        except:
            pass  # Declenche si absence de transition pour ce symbole pour l'etat
        coupleTmp = [None,None]
        try:
            coupleTmp[0] = self.__indexes(\
                self.__automate.getTransitionsB()[premierEtat])
            coupleTmp[1] = self.__indexes(\
                self.__automate.getTransitionsB()[secondEtat])
            couplesSortie.extend(self.__toutesCombinaisons(coupleTmp))
        except:
            pass  # Declenche si absence de transition pour ce symbole pour l'etat
        return couplesSortie
```

Pour récuperer les couples manquants, le programme génère tous les couples possibles puis il supprime un par un tous ceux qui existe dans notre file. Les couples qui restes sont ceux qui n'existent pas en file donc ceux indistingables. Le programme vient effectuer l'opération ensembliste suivantes :

soit :
A ={Ensemble des couples d'états distingués que possède la file}
$\bar{A}$ ={Ensemble des couples d'états indistingués}
Ω ={Ensemble des couples d'états possibles}

Nous cherchons à avoir $\bar{A}$.

$\bar{A} = \Omega - A$
Pour récuperer les états indistingables, il a suffit de prendre les états existant au moins une fois dans les couples de $\bar{A}$.

2.3 Programme et execution

Dans cette partie, nous allons détailler quelques méthodes du programme permettant le bon fonctionnement de l'algorithme décrit ci-avant puis allons mettre en pratique avec un exemple.

Notre programme tient essentiellement sur des matrices contenant des transitions d'état. Pour pouvoir récuperer depuis un état les états qui pointent vers lui nous avons développé la methode *indexes()*. Cette méthode recupère les numéros des colonnes de la matrice pour une ligne donnée.

```
    ################################
    # Fonction qui recupere la position
    # de tous les "1" d'un liste
    #
    # ENTREE: liste transition de l'etat de depart
    # SORTIE: liste entier des etats arrives
    ################################
    def __indexes(self, etatPointe: list[int]) -> list[int]:
        sortie :list[int] = []
        for transition in range(self.__automate.getNbEtat()):
            if etatPointe[transition]==1:
                sortie.append(transition)
        if(len(sortie)==0):
            return None
        return sortie
```

Avant de pouvoir ajouter en file tous les états distingués engendré par un couple d'état, il faut pouvoir trouver toutes les combinaisons de couples d'états. Pour cela la méthode *toutesCombinaisons()* permet d'associer les états entre-eux.

```
    ################################
    # Fonction qui realise toutes les
    # combinaisons possible entre deux listes
    # ENTREE: Un tuple de listes
    # SORTIE: liste entier
    ################################
    def __toutesCombinaisons(self, l_couple: list[list[int]])->list[list[int]]:
        sortie :list[list[int]] = []
        for i in range(len(l_couple[0])):
            for j in range (len(l_couple[1])):
                sortie.append([l_couple[0][i],l_couple[1][j]])
        return sortie
```

Cette méthode se contente uniquement de faire toutes les combinaisons possibles, c'est la méthode appelante qui va gérer les doublons par exemple.

Exemple :
Concidérons l'automate suivant :

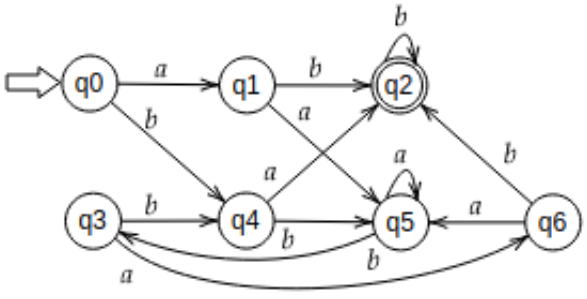

FIGURE 2 – Automate du premier exemple

La représentation de notre automate pour le programme sera la suivante :

```
7
2
0 1 0 0 0 0 0
0 0 0 0 0 1 0
0 0 0 0 0 0 0
0 0 0 0 0 0 1
0 0 1 0 0 0 0
0 0 0 0 0 1 0
0 0 0 0 0 1 0
0 0 0 0 1 0 0
0 0 1 0 0 0 0
0 0 1 0 0 0 0
0 0 0 0 1 0 0
0 0 0 0 0 1 0
0 0 0 1 0 0 0
0 0 1 0 0 0 0
```

Grace au fichier l'automate est chargé correctement dans le programme.

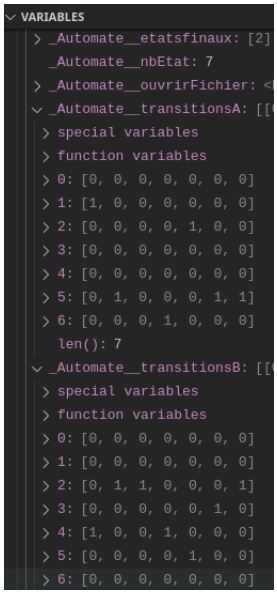

FIGURE 3 – Automate chargé en mémoire du programme

Nous pouvons constater que toutes les informations sont chargées en mémoire et les transitions sont les transposées des matrices du fichier. Par ailleurs, nous pouvons observer la file durant l'execution et voir qu'elle contient les couples de l'initialisation et ceux trouvés lors du déroulement

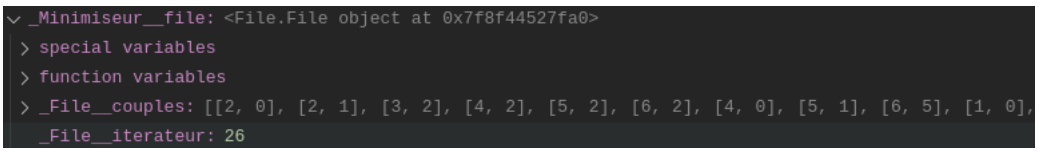

FIGURE 4 – Une partie de la file à la fin de l'execution

Après execution de notre programme, nous obtenons la sortie suivante qui correspond à ce que nous avions trouvé lors de notre recherche manuel en amont.

```
[jeremod@jeremod-desktop Minimisation]$ python main.py
Les etats identiques de l'automate sont: [3, 0, 6, 1]
L'automate minimiser fait 3
```

FIGURE 5 – Affichage du programme pour l'automate de l'exemple