📄 Fonction `lire_automate`

🎯 Objectif de la fonction

La fonction `lire_automate(nom_fichier)' a pour but de lire un fichier texte décrivant un automate fini (déterministe ou non, avec ou sans transitions ε) et de construire une représentation mémoire claire et exploitable de cet automate.

🧠 Représentation interne de l'automate

L'automate est représenté en mémoire par un dictionnaire Python structuré ainsi :

```
{
 'alphabet': List[str],              # Ex: ['a', 'b']
 'etats': List[int],                 # Ex: [0, 1, 2, 3]
 'etats_initiaux': List[int],        # Ex: [0]
 'etats_terminaux': List[int],       # Ex: [2, 3]
 'transitions': Dict[(int, str), Set[int]]  # Ex: {(0, 'a'): {1, 2}}
}
```

📌 Remarques :

- `alphabet` : symboles de l'automate, générés automatiquement selon `nb_symboles`

- `transitions` : permet la représentation d'automates **non-déterministes** en associant un couple `(état, symbole)` à un ensemble d'états d'arrivée

⚙️ Étapes de la fonction

1. **Ouverture du fichier** : avec `open()` et lecture des lignes

2. **Extraction des métadonnées** : nombre de symboles, états, transitions...

3. **Conversion et nettoyage** : via `strip()`, `split()`, `map(int, ...)`

4. Validation des données

- États valides (entre 0 et `nb_etats - 1`)

- Symboles valides (parmi `'a'` à `'a' + n`)

- Nombre de lignes de transition cohérent

5. Construction du dictionnaire `transitions`

- Chaque transition est insérée dans un dictionnaire

- Si plusieurs transitions existent pour un même couple `(état, symbole)`, elles sont regroupées dans un `set`

📄 Fonction `afficher_automate`

🎯 Objectif de la fonction

La fonction `afficher_automate(automate)` a pour but d'afficher de manière lisible et structurée les composants d'un automate représenté en mémoire (sous forme de dictionnaire Python).

Elle est conçue comme outil de visualisation rapide de l'automate courant.

🧠 Rappel : Structure de l'automate en mémoire

L'automate est un dictionnaire avec la structure suivante :

```
{
 'alphabet': List[str],
 'etats': List[int],
 'etats_initiaux': List[int],
 'etats_terminaux': List[int],
 'transitions': Dict[(int, str), Set[int]]
}
```

📋 Fonctionnement détaillé

```
def afficher_automate(automate):
    print("Alphabet:", automate['alphabet'])
    print("États:", automate['etats'])
    print("États initiaux:", automate['etats_initiaux'])
    print("États terminaux:", automate['etats_terminaux'])
    print("Transitions:")
    for (depart, symbole), arrivees in automate['transitions'].items():
        print(f"{depart} --{symbole}--> {arrivees}")
```

🔍 Détail des étapes

1. Affiche l'alphabet :
   - Liste des symboles utilisables dans les transitions.

2. Affiche tous les états :
   - Généralement une liste d'entiers : `[0, 1, 2, ...]`

3. Affiche les états initiaux :
   - Peut contenir plusieurs éléments (AFN), ex: `[0]` ou `[0, 2]`

4. Affiche les états terminaux :
   - États d'acceptation de l'automate.

5. Affiche les transitions :
   - Pour chaque couple `(état, symbole)` :

```
0 --a--> {1, 2}
1 --b--> {3}
```

🔍 Remarque sur les `arrivees`

- Les `arrivees` sont souvent des `set () ` :

 - ` {1, 2} ` dans un AFN

- Mais dans certains cas (AFD), ce peut être un seul entier (` 3 `)

 - Le code les affiche de manière uniforme grâce à ` print()`.

📄 Fonctions de vérification d'un automate

🎯 Objectif global

Ces fonctions ont pour but de vérifier des propriétés fondamentales d'un automate fini représenté en mémoire. Elles permettent de savoir si l'automate est :

- Déterministe

- Standard

- Complet

Ces vérifications sont utiles avant d'appliquer des algorithmes comme la déterminisation, la standardisation ou la minimisation.

🔎 1. Fonction `est_déterministe(automate)`

📌 Objectif :

Vérifie si un automate est déterministe, c'est-à-dire :

- Il n'a qu'un seul état initial

- Il n'y a qu'une seule transition pour chaque couple `(état, symbole)`

🔍 Étapes :

- Vérifie l'existence de la clé `'transitions'`

- Vérifie que la liste `etats_initiaux` contient exactement 1 état

- Parcourt chaque transition :

 - Si le `set` des états d'arrivée a plus d'un élément → l'automate n'est pas déterministe

- Gère les cas où l'état d'arrivée est un entier ou une chaîne (compatibilité AFD)

✅ Résultat :

- `True` si toutes les conditions sont respectées

- `False` sinon, avec un message explicite

🔎 2. Fonction `est_standard(automate)`

📌 Objectif :

Vérifie si un automate est standard, c'est-à-dire :

- Il possède un unique état initial

- Aucune transition ne mène vers cet état initial

🔍 Étapes :

- Vérifie la présence et l'unicité de `etats_initiaux`

- Pour chaque transition, vérifie que l'état initial n'apparaît pas comme état d'arrivée

✅ Résultat :

- `True` si l'automate est standard

- `False` avec une explication sinon

🔎 3. Fonction `est_complet(automate)`

📌 Objectif :

Vérifie si l'automate est complet, c'est-à-dire :

- Pour chaque état et chaque symbole, une transition est définie

🔍 Étapes :

- Parcourt tous les couples `(état, symbole)`

- Identifie ceux absents dans `transitions`

- Les transitions manquantes sont listées

✅ Résultat :

- `True` si l'automate est complet

- `False` avec la liste des transitions manquantes

🔎 4. Fonction `verifier_automate(automate)`

📌 Objectif :

Affiche un résumé des trois propriétés ci-dessus en une seule commande.

🔍 Détail :

- Appelle successivement :

 - `est_deterministe`

 - `est_standard`

 - `est_complet`

- Affiche pour chacune : `True` ou `False` avec les messages explicatifs

✅ Utile pour :

- Valider un automate avant de le manipuler

- Identifier rapidement les actions à effectuer : standardiser, compléter, etc.

📄 Fonction `standardiser(automate)`

🎯 Objectif de la fonction

La fonction `standardiser(automate)` transforme un automate non standard en un automate standard, c'est-à-dire :

- Ayant un seul état initial

- Et aucune transition menant vers cet état initial

📘 Définition d'un automate standard

Un automate est dit standard s’il :

1. Possède un unique état initial

2. Cet état initial n’est jamais atteint via une transition

⚙️ Fonctionnement détaillé

1. Copie de l’automate

```python
auto = copy.deepcopy(automate)
```

- Utilisation de `deepcopy` pour ne pas modifier l'automate d'origine

2. Vérification

```python
if est_standard(auto):
  return automate
```

- Si l'automate est déjà standard → on le retourne tel quel

3. Création d’un nouvel état initial

```python
nouvel_etat = max(auto['etats']) + 1
auto['etats'].append(nouvel_etat)
auto['etats_initiaux'] = [nouvel_etat]
```

- Crée un état inédit (ex: `5` si les états vont jusqu'à `4`)

- Ce nouvel état devient l’unique état initial

4. Préservation des anciens états initiaux

- On conserve la connexion vers les anciens états initiaux depuis le nouveau :

```python
auto['transitions'][(nouvel_etat, symbole)] = ...
```

5. Propagation des transitions

- Toutes les transitions partant des anciens états initiaux sont copiées pour partir du nouvel état initial

- Cela garantit que le nouveau comportement simule le même langage

6. État terminal

```python
if ancien_initial_est_terminal:
    auto['etats_terminaux'].append(nouvel_etat)
```

- Si un ancien état initial était terminal, le nouveau doit aussi l'être pour préserver le langage reconnu

📎 Exemple

Avant :

```python
etats_initiaux = [0, 3]
(3, 'a') → 2
```

Après :

```python
etats_initiaux = [5]
(5, 'a') → {2}   copie de (3, 'a')
```

📄 Déterminisation avec transitions ε (epsilon)

🎯 Objectif

Les trois fonctions `fermeture_epsilon`, `contient_epsilon_transitions`, et `determiniser_automate_epsilon` permettent de traiter un automate non déterministe avec ε-transitions et de le transformer en un automate déterministe équivalent (AFD).

🔹 Fonction `fermeture_epsilon(automate, etats)`

📌 But :

Calculer la fermeture ε d'un ensemble d'états :

→ Ensemble des états accessibles par une ou plusieurs transitions ε (`'€'`), à partir de `etats`.

🔍 Processus :

- Initialise `fermeture` avec les états de départ

- Utilise une pile pour explorer récursivement chaque transition ε

- Pour chaque état atteint, ajoute tous les nouveaux voisins via ε

✅ Résultat :

Un `set` contenant tous les états accessibles par ε depuis les états de départ.

🔹 Fonction `contient_epsilon_transitions(automate)`

📌 But :

Vérifie si l'automate contient au moins une transition ε

🔍 Détail :

- Parcourt toutes les transitions

- Retourne `True` dès qu'un symbole `'€'` est détecté

- Sinon, retourne `False`

🔹 Fonction `determiniser_automate_epsilon(automate)`

📌 But :

Transformer un automate non déterministe avec ε-transitions en un automate déterministe (AFD), en respectant le langage reconnu.

🔍 Étapes principales :

1. Vérification :

  - Si l'automate est déjà déterministe → retour immédiat

2. Initialisation :

  - Calcul de la fermeture ε de l'état initial

  - Nom symbolique de l'état initial : ex: `'0_2'` si `0` et `2` sont accessibles par ε

3. Boucle principale (exploration BFS) :

  - À chaque étape, calcule :

    - Tous les états atteignables par un symbole depuis un état courant

    - Puis leur fermeture ε

  - Crée un nouvel état (nommé par un tuple trié)

  - Enregistre une transition déterministe vers ce nouvel état

4. Construction de l'AFD :

  - `etats_afd` : associe à chaque ensemble d'états un nom unique

  - `transitions_afd` : dico de transitions entre noms d'états

  - `etats_terminaux_afd` : contient tous les états dont au moins un sous-état est terminal dans l'automate original

📘 Remarques importantes

- Chaque état de l'AFD représente un ensemble d'états de l'AFN

- Les ε-transitions sont uniquement utilisées pour atteindre des états, jamais dans les transitions finales

- La fermeture ε est utilisée à deux moments :

  - Pour initialiser l'état de départ

  - Pour étendre les transitions accessibles

📎 Résultat de `determiniser_automate_epsilon`

```python
{
 'alphabet': ['a', 'b'],
 'etats': {'0_2', '1_3'},
 'etats_initiaux': {'0_2'},
 'etats_terminaux': {'1_3'},
 'transitions': {
  ('0_2', 'a'): '1_3',
  ...
 }
}
```

📄 Déterminisation et Complétion d'un Automate

🎯 Objectif global

Ces fonctions permettent de transformer un automate non déterministe (AFN) en un automate déterministe et complet (AFD complet), prêt à être utilisé pour la reconnaissance de mots, la complémentarisation ou la minimisation.

🔹 Fonction `determiniser_automate(automate)`

📌 But :

Transformer un AFN (sans ε) en un AFD, en créant de nouveaux états représentant des ensembles d'états du non-déterministe.

🔍 Étapes :

1. Vérifie si l'automate est déjà déterministe

2. Initialise l'état de départ comme un tuple d'états initiaux

3. Utilise une file FIFO (BFS) pour explorer les transitions

4. Pour chaque symbole :

- Calcule les états atteignables

- Crée un nouvel état composite

- Enregistre la transition

✅ Résultat :

Un automate déterministe avec :

- Des états nommés par concaténation (ex: `'0_2'`)

- Des transitions uniques pour chaque couple `(état, symbole)`

🔹 Fonction `completer_automate(automate, debug=False)`

📌 But :

S'assurer que pour chaque état et chaque symbole, une transition existe

🔍 Étapes :

1. Vérifie si l'automate est déjà complet

2. Si non, crée un état puits (numéroté ou nommé `"puits"`)

3. Pour chaque couple `(état, symbole)` manquant :

- Ajoute une transition vers le puits

4. Le puits boucle sur lui-même pour tous les symboles

✅ Résultat :

Un automate complet, prêt pour :

- La reconnaissance sûre de tous les mots

- La complémentarisation

🔹 Fonction `determiniser_et_completer(automate)`

📌 But :

Chainer les deux opérations précédentes automatiquement :

1. Déterminisation (avec ou sans ε)

2. Complétion

🔍 Étapes :

1. Détecte s'il y a des transitions ε

  - Si oui → `determiniser_automate_epsilon`

  - Sinon → `determiniser_automate`

2. Applique `completer_automate`

3. Affiche les deux versions (intermédiaire et finale)

🧠 Pourquoi ces étapes sont cruciales ?

| Étape | Utilité |

|---|---|

| Déterminisation | Représenter l'automate sous forme AFD, unifiée |

| Complétion | Garantir un comportement défini pour tout mot (utile pour complément) |

🕯️ Exemple de transformation

- AFN :

- États : `0, 1`

- Transitions : `0 --a--> {0, 1}`

- AFD :

- États : `'0'`, `'0_1'`

- Transitions : `'0' --a--> '0_1'`

- Complétion :

- Si `'0_1' --b-->` est manquant → ajoute `'0_1' --b--> puits`

📄 Fonction `creer_automate_complementaire(automate)`

🎯 Objectif

Cette fonction transforme un automate déterministe et complet en son automate complémentaire :

→ Il reconnaît tous les mots que l'automate original ne reconnaît pas.

🔍 Définition : automate complémentaire

Pour un automate déterministe et complet, le complémentaire est obtenu en :

1. Gardant la structure (états, transitions)

2. Inversant les états terminaux :

- Tous les états non terminaux deviennent terminaux

- Les anciens terminaux ne le sont plus

⚙️ Fonctionnement détaillé

1. 🧪 Vérification de base

```python
if not automate['etats']:
```

raise ValueError("L'automate est vide.")

```

- Lève une erreur si aucun état n'est défini

2. ⚙️ Préparation de l’automate

```python

automate = determiniser_automate(automate)

automate = completer_automate(automate)

```

- L’automate est rendu déterministe et complet

- Cela est indispensable pour que la complémentarisation soit correcte

3. 📋 Duplication de l’automate

```python

automate_complementaire = copy.deepcopy(automate)

```

- Permet de ne pas modifier l’automate d’origine

4. 🔁 Inversion des états terminaux

```python

automate_complementaire['etats_terminaux'] = set(automate['etats']) - set(anciens_terminaux)

```

- Tous les états qui n’étaient pas terminaux le deviennent

- Cette opération est basée sur un `set` pour éviter les doublons

✨ Résultat

Un automate avec :

- Les mêmes états et transitions

- Un **ensemble terminal inversé**
```

📘 Exemple

Automate original :

- États : `0, 1, 2`

- Terminaux : `{2}`

Automate complémentaire :

- Terminaux : `{0, 1}`

📄 Fonction `reconnaitre_mot (automate, mot)`

🎯 Objectif

Cette fonction permet de vérifier si un mot donné est accepté ou rejeté par un automate déterministe (AFD ou AFD complet). Elle simule le comportement de l'automate sur ce mot.

⚠️ Prérequis

L'automate utilisé doit :

- Être déterministe

- Avoir un alphabet bien défini

- Utiliser une structure de transitions ` (état, symbole) → état ou set(d'états)`

⚙️ Fonctionnement détaillé

1. 🔐 Vérification de l'état initial

```python
if len(automate['etats_initiaux']) != 1:
    raise ValueError("L'automate doit avoir un seul état initial.")
```

2. 🧭 Lecture du mot

- Récupère l'état initial

- Parcourt chaque symbole du mot :

  - Vérifie que le symbole est dans l'alphabet

  - Vérifie qu'il existe une transition définie

  - Suit la transition (prend l'unique état d'arrivée)

3. 🔁 Simulation état par état

```python
if (etat_courant, symbole) in automate['transitions']:
    arrivees = automate['transitions'][(etat_courant, symbole)]
    if isinstance(arrivees, set):
        etat_courant = next(iter(arrivees))
    else:
        etat_courant = arrivees
```

- Avance dans l'automate

- Gère les cas où les transitions sont :

  - Un set (automates déterministes mal formatés)

  - Un entier ou une chaîne (état unique)

4. 🍼 Fin de mot : vérification terminale

```python
if etat_courant in automate['etats_terminaux']:
    return True
```

- Le mot est accepté si l'état atteint à la fin est terminal

📘 Exemple

Automate :

- Alphabet : `['a', 'b']`

- Transitions :

 - `0 --a--> 1`

 - `1 --b--> 2`

- État initial : `0`

- État terminal : `2`

Mot :

- `"ab"` → accepté ✅

- `"aa"` → rejeté ❌ (transition absente)

📄 Fonction `minimiser_automate(automate)`

🎯 Objectif

Transformer un automate déterministe complet (AFD) en un automate minimal équivalent, qui reconnaît le même langage mais avec le nombre minimal d'états.

🔎 Fonction `reconnait_uniquement_mot_vide(automate)`

📌 But :

Détecter si l'automate reconnaît uniquement le mot vide ε

✅ Conditions :

1. Un seul état initial

2. Cet état est terminal

3. Aucune transition ne part de cet état

➡️ Si toutes ces conditions sont remplies, on retourne un automate minimal à 1 seul état.

PS : nous avons écrit cette fonction à cause de l'automate 1

🔧 Fonction `minimiser_automate(automate)`

📌 Étapes principales :

1. 🔍 Cas particulier : mot vide

- Si l'automate reconnaît uniquement le mot vide :
  - Retour d'un automate minimal :
    - Un seul état
    - Terminal
    - Aucune transition

2. 🖌️ Préparation

- Si l'automate n'est pas déterministe→ déterminisation
- Si l'automate n'est pas complet → complétion

3. 🧠 Algorithme de minimisation (partition de Moore)

a. Initialisation des partitions

- 2 groupes :
  - États terminaux
  - États non terminaux

b. Raffinement

- Pour chaque symbole :
  - On divise chaque groupe de la partition en sous-groupes
    selon les transitions qui mènent à un groupe donné
- Répéter jusqu'à stabilisation

4. 🧱 Construction de l'automate minimisé

- Chaque groupe devient un **nouvel état**
- Les transitions sont redirigées :
  - En fonction du **représentant** de chaque groupe
- On reconstruit :

- Nouveaux états initiaux (si un état initial est dans un groupe)

- Nouveaux états terminaux

📘 Exemple :

Automate original :

- États : `0, 1, 2`

- Transitions : `0 --a--> 1`, `1 --a--> 2`, `2 --a--> 1`

- Terminaux : `{1}`

Après minimisation :

- États minimaux : `q0`, `q1`

- Transitions reconstituées

- Même langage reconnu

📌 Résultat

```python
{
 'alphabet': ['a', 'b'],
 'etats': ['q0', 'q1'],
 'etats_initiaux': ['q0'],
 'etats_terminaux': ['q1'],
 'transitions': {
  ('q0', 'a'): {'q1'},
  ...
 }
}
```