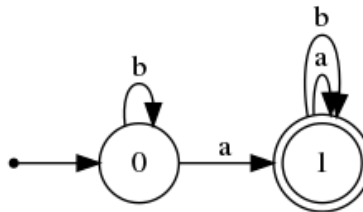


Dans ce TD nous allons introduire les automates, un modèle de calcul permettant de déterminer si une séquence d'information est valide ou non, selon une règle déterminée. Dans un premier temps nous allons construire des automates simples, et ensuite les implémenter en Python et résoudre des problèmes de complexité croissante.

Qu'est-ce qu'un automate ?

Un automate est une séquence d'actions, pré-déterminées sous forme d'états et de transitions. Un exemple d'automate est un feu tricolore, où les états sont la couleur du feu (rouge, vert, orange) et les transitions les changements possibles de couleurs (du rouge au vert, du vert au orange, etc.). Les automates permettent donc d'encoder le fonctionnement d'un système plus ou moins complexe et de détecter des erreurs (en reprenant l'exemple du feu tricolore, passer du vert au rouge directement est une erreur). Les applications des automates sont nombreuses et offrent souvent un code plus facile à écrire et lire.



Un automate peut être représenté graphiquement (comme ci-dessus) et possède une structure de données similaire à un graphe orienté, où chaque noeud est un état, et un arc une transition possible d'un état à l'autre. Ce graphe est ensuite parcouru à partir de **mots** qui sont une suite de symboles (ex. lettres) permettant de passer d'un état à l'autre. Le premier état est indiqué avec une flèche entrante, et le dernier état (il peut y en avoir plusieurs) avec un double cercle. Les symboles **a** et **b** consistent l'alphabet de l'automate (il est possible d'utiliser un autre alphabet comme les transitions d'un feu tricolore).

Dans l'exemple ci-dessus, si le mot à lire est **ab**, l'automate commence à lire le mot depuis l'état 0 et réalise ensuite une transition vers un autre état à partir du premier symbole du mot, la lettre **a**. La transition vers l'état 1 est réalisée. L'automate se retrouve alors dans l'état 1 et lit le symbole suivant, la lettre **b**. L'automate reste sur l'état 1 (il s'agit d'une boucle).

Les automates servent à valider un comportement que l'on appelle un *motif* (ou langage), et dans l'exemple ci-dessus l'automate valide tous les mots contenant au moins un symbole **a** (que l'on note plus communément ***a***, avec ***** indiquant que tout symbole peut être utilisé).

Pour résumer, un automate est défini comme $A = (\Sigma, S, s_0, \delta, F)$, avec :

- Σ , un ensemble fini, non vide de symboles qui est l'alphabet d'entrée
- S , un ensemble fini, non vide d'états
- s_0 , l'état initial, élément de S

- δ , la fonction de transition d'états: $\delta : S \times \Sigma \rightarrow S$
- F est l'ensemble des états terminaux, un sous-ensemble (éventuellement vide) de S

1 Automates simples (45min, sur papier)

Vous pouvez également utiliser l'outil en ligne ¹ pour générer un automate au format dot (le format est introduit dans la question 2.4).

Exercice 1.1 – Proposez un automate qui contient un nombre paire de fois la lettre **a** (pour un alphabet contenant les lettres ***a*** et ***b***).

Exercice 1.2 – Proposez un automate qui valide toutes les paires de **ab** ou **ba**. Faire une représentation graphique de cet automate. Testez le avec **ababab** ou **babababa**.

Exercice 1.3 – Proposez un automate qui valide le motif ***aaba***. Donnez une représentation graphique et proposez des mots à tester.

Exercice 1.4 – Proposez enfin un automate qui valide une adresse email, comme suit : la première partie accepte toute lettre, mais uniquement des chiffres à la fin de cette partie; ensuite le **@**, un nom de domaine (sous forme de lettres et chiffre, avec un seul point **.** pour simplifier). Vous pouvez voir la RFC 822² pour une définition plus précise.

2 Structure de données d'automate en Python (45min)

Exercice 2.1 – Implémentez en Python une structure de données d'automate. Vous pouvez suivre les étapes suivantes :

1. Une méthode **init** qui initialise l'automate avec les symboles du motif (ici **a** et **b**) et les variables d'état interne.
2. Une méthode **ajout_etat** qui rajoute un nouvel état et s'assure que l'état n'existe pas déjà
3. Une méthode **ajout_transition** qui rajoute un nouvel état et s'assure que l'état n'existe pas déjà.
4. Une méthode **recherche_etat** qui étant donné un état source et un symbole, renvoie l'état correspondant (via une transition rajouté).
5. Une fonction **run** qui valide un mot donnée et renvoie **True** si l'état final est atteint et **False**.

Voici l'initialisation et le test de l'automate qui implémente le motif ***a*** :

```
a = automate("abc")
```

¹<https://pypi.org/project/graphviz/>

²<https://www.w3.org/Protocols/rfc822/>

```
a.ajout_state("0")
a.ajout_state("1", True)

a.ajout_transition("0", "b", "0")
a.ajout_transition("0", "a", "1")
a.ajout_transition("1", "a", "1")
a.ajout_transition("1", "b", "1")

print(a)

assert a.run("abaaaaa") == True
assert a.run("bbb") == False
```

Exercice 2.2 – Utilisez votre structure de données pour implémenter les automates de la partie précédente.

Exercice 2.3 (bonus) – Implémentez une méthode `__str__` afin que la commande `print(a)` affiche les états internes à l'automate :

```
automate :
- alphabet : 'abc'
- init : 0
- final : ['1']
- etats (2) :
  - (0)automate :
- alphabet : 'abc'
- init : 0
- final : ['1']
- etats (2) :
  - (0):
    --(b)--> (0)
    --(a)--> (1)
  - (1):
    --(a)--> (1)
    --(b)--> (1)
```

Exercice 2.4 (bonus) – Implémenter une méthode `export_dot` afin d'exporter l'automate en utilisant le format dot <https://graphviz.org/doc/info/lang.html> de la bibliothèque Graphviz <https://pypi.org/project/graphviz/>

Un exemple [charger en ligne](#).

```
digraph auto {
  rankdir="LR";
```

```
// Etats (12)
node [shape = point ];      __Qi__ // Etat initial invisible
node [shape=circle]; Q_0 [label=0];
node [shape=doublecircle]; Q_1 [label=1]; // Etat final

// Transitions
__Qi__ -> Q_0; // Etat initial fleche
Q_0 -> Q_0 [label=b];
Q_0 -> Q_1 [label=a];
Q_1 -> Q_1 [label=a];
Q_1 -> Q_1 [label=b];
}
```

3 Analyse de texte avec un automate (30min)

Nous allons maintenant développer un programme qui utilise votre structure de données d'automate implémentée en Python dans la section précédente. L'objectif de ce programme sera le suivant : proposer de compléter un mot, à partir d'une séquence de lettres partielle donnée. Par exemple votre programme prend en entrée la séquence **bon**, en retour vous devez proposer une séquence de lettres pertinentes afin de compléter ce mot comme **bonjour** ou **bonsoir**.

Vous êtes libres de proposer la stratégie de recommandation de lettres que vous souhaitez. Nous vous proposons de vous baser sur les listes de mots les plus fréquents en Français <http://www.pallier.org/extra/liste.de.mots.francais.frgut.txt> (fourni dans le fichier `code/mots.txt`) par exemple. Ces mots permettent de réaliser des statistiques de co-occurrences. Par exemple, étant donnée les mots suivants :

```
abaissa
abaissable
abaissables
abaissai
abaissaient
abaissais
abaissait
abaissâmes
```

Votre programme recommande les lettres suivantes ordonnées par ordre de probabilité de transition pour compléter le mot (basé sur l'analyse du fichier `code/mots-10.txt` qui contient les mots ci-dessus):

```
i (4)
b (2)
m (1)
```

Conseils :

1. Utilisez les fichiers de listes de mots (mots.txt, ..) en analysant leur fréquence de co-occurrences
2. Construisez un automate qui encode les transitions possibles entre lettres
3. Proposez une méthode de recommandation de transition à partir de quelques lettres fournies en entrée