CERGY PARIS UNIVERSITÉ

Rapport de projet d'intégration

Sujet:

Implémentation des algorithmes du cours d'Automates

Présenté par:

- RIGHI Racim
- TRABELSI Lydia

Encadré par:

- BRIQUEL Irénée

2020/2021

CERGY PARIS UNIVERSITÉ

CERGY PARIS UNIVERSITÉ

INTRODUCTION

Ce rapport est composé de trois parties : une introduction pour décrire le projet et ses buts, une deuxième partie qui traite de la problématique du sujet, les solutions proposées ainsi que les résultats de ces derniers. Enfin, une dernière partie pour clore le sujet.

Nous allons donc procéder à la présentation de tous les détails de ce projet, en commençant par définir les représentations des différentes composantes du programme : les automates ainsi que les transitions qui composent ces derniers. Ensuite nous définirons la structure du projet. Puis, nous mettrons en avant les différents modes d'exécution que propose ce programme, qui donne la possibilité d'utiliser que certaines parties de ce dernier si cela est désiré.

Après cela, seront décrites et expliquées, toutes les fonctionnalités proposées par le programme. Nous détaillerons, alors, la partie graphique, de quelle façon sont générées les images des automates produits. Nous nous attarderons donc sur Graphviz, le définissant et expliquant son utilisation, ainsi que le format Json.

Une des parties les plus compliquées a été la génération automatique d'automates cohérents, nous avons donc réussi cette étape là et elle a donc été expliquée dans ce rapport. En dernier lieu, nous parlerons de PyLaTex, comment il nous a permis d'atteindre le but principal du projet qui est la génération de sujets d'examens et leurs corrigés.

Enfin, pour conclure ce projet, nous présenterons les résultats des tests de chacune des parties du programme, illustrant ceux-ci par différents automates.

1. Présentation du projet

1.1. Description

Ce projet représente un programme destiné aux utilisateurs voulant générer des sujets de contrôles ou d'examens pour le cours "Langages et Automates".

Pour ce faire, ce programme propose plusieurs solutions automatiques pour générer des automates à états finis, vérifier si une liste de mots est acceptée par l'automate ou non, générer un automate sans ε-transitions à partir d'un automate donné, déterminiser un automate, calculer un automate déterministe minimal, déterminer si deux automates sont équivalents et retourner un automate de Thompson calculant un langage donné suivant une expression régulière. Ces solutions permettent donc la génération de sujets sous format PDF avec Latex en utilisant les automates décrits auparavant ainsi que leurs solutions.

L'objectif de ce projet est de réaliser un prototype qui peut être repris par un enseignant, et l'adapter à ses besoins pour pouvoir proposer des sujets uniques aux étudiants. Toutes les fonctionnalités du projet sont utilisables indépendamment les unes des autres.

1.2. Techniques utilisées

Le projet nécessite des connaissances en programmation orientée objets en employant le langage Python. Ce langage a été choisi car en plus de sa polyvalence et sa simplicité, les nombreuses librairies disponibles ont été d'une grande aide, celles-ci seront citées et décrites plus tard dans ce rapport. Quant aux fichiers générés, il a été décidé d'utiliser Latex. Graphviz, quant à lui, a permis la réalisation de la partie graphique du projet. Plus de détails seront présentés plus tard dans le rapport.

Pour développer les différentes fonctionnalités du programme, on a principalement usé des définitions des AEFs (Automates à États Finis) pour définir les structures, les notions étudiés en théories des graphes et en algorithmique ont été très importantes vu l'utilisation des différentes structures de données, dont les graphes, les piles ainsi que les files et le développement des algorithmes s'inspirent des parcours des graphes qui traitent les opérations décrites plus haut.

CERGY PARIS UNIVERSITÉ

CONCEPTION

Dans cette partie, on va décortiquer les structures de données utilisées et les fonctionnalités principales du projet.

Dans le but de représenter les automates et tous leurs éléments, à savoir les états (initiaux, finaux et autres), les transitions et les alphabets, on a opté pour la programmation orientée objets, et ce pour utiliser des classes comme définitions de ces éléments.

1. Définitions

1.1. Représentation d'une transition

Une transition dans un automate est constituée de trois éléments: un état de départ, une lettre de l'alphabet à lire, et un état d'arrivée.

Elle est définie dans la classe ***Transition*** selon les éléments précédents par *(mFrom (str), mValue(str), mGoto(str))*

1.2. Représentation d'un automate

Un automate est défini par la classe ***Graph*** par les éléments suivant:

- **states[]: liste des états:** chaque état étant représenté par un numéro unique sous forme de chaîne de caractères *(str)*
- **finalStates[]: Liste des états finaux:** Chaque automate contient obligatoirement au minimum un état final, qui appartient à la liste des états.
- **initialState (str): Etat initial**
- **alphabet[]: liste des éléments de l'alphabet:** Un élément de l'alphabet est forcément un caractère (alphanumérique)
- **transitions: liste des Transitions:** Une transition est représentée par un objet ***Node*** décrit ci-dessus, donc la liste est constituée de plusieurs transitions.

2. Structure du projet

L'un des plus grands avantages de l'utilisation du langage Python est la présence des modules, de ce fait, on a essayé de découper le code au maximum pour faciliter le débogage et la lisibilité.

Le seul point d'entrée du projet est depuis le fichier ***index.py,*** qui permet de lancer un des modes d'exécution selon les arguments donnés en ligne de commande.

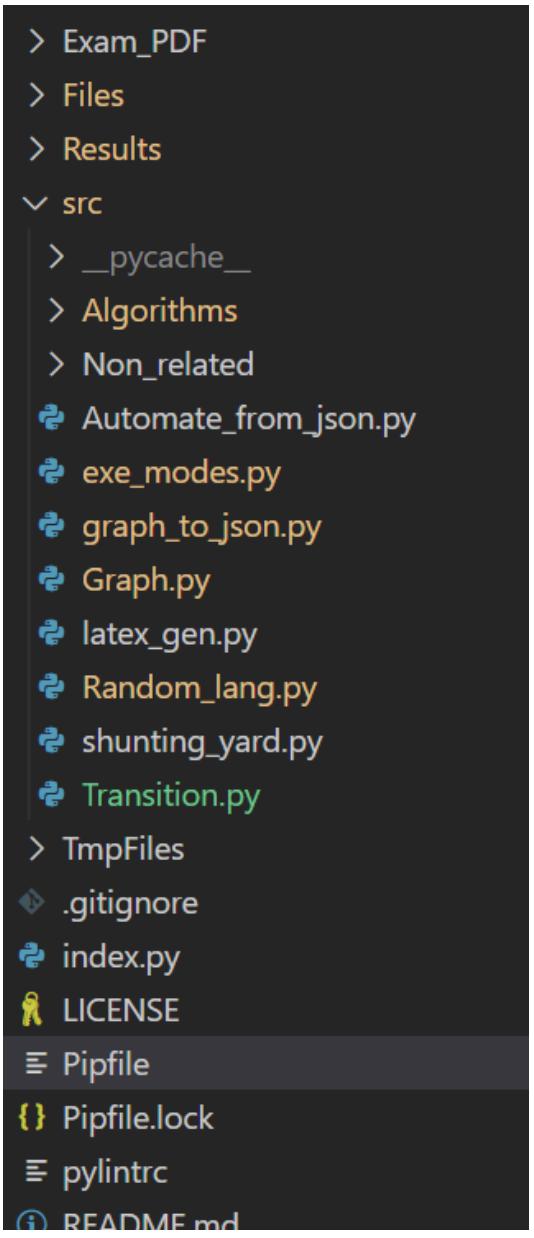


Figure 1: Arborescence de fichiers du projet

3. Cas d'utilisations

Le programme étant découpé en plusieurs modules réutilisables, il est possible d'appeler chaque fonction indépendamment des autres. Cependant il offre par défaut plusieurs modes d'exécution qui enveloppent les principales fonctionnalités, ces modes sont:

- Mode génération automatique de langages: en utilisant la commande

  $ python ./index.py -g <n>

  Ce mode permet de générer ***n*** langages automatiquement sous le format txt, générer les automates synchronisés, déterminés et minimales correspondants. Ensuite crée l'image de chacun de ces derniers

- Mode batch: En lisant les fichiers du dossier ***Files,*** créer des automates et exécuter leurs algorithmes pour les mettre dans le dossier ***Results*** sous format JSON et PNG.

  $ python ./index.py -f

- Mode génération d'automates de Thompson: cet algorithme nécessite des expressions régulières correctes et bien formatés, donc elles sont passées via l'invite de commande:

  $ python ./index.py -t "<exp>" # Exemple "a.(a+b)*.b"

- Mode génération de sujets d'examens: A partir des automates créés précédemment, générer des sujets d'examens avec leurs corrections.

  $ python ./index.py -e

CERGY PARIS UNIVERSITÉ

IMPLEMENTATION

---

1. Algorithmes d'automates

1.1. Acceptation

L'algorithme d'acceptation prend en paramètre un automate déterministe et une liste de mots à vérifier. Il renvoie un dictionnaire avec comme clés les mots donnés en paramètres, et pour chaque clé, une valeur booléenne *True* ou *False* selon si le mot appartient au langage ou pas.

1.2. Synchronisation

Le caractère epsilon et représenté par le caractère *unicode '\u03b5'*, donc l'algorithme de synchronisation parcours toutes les transitions de l'automate, et vérifie l'existence de ce caractère, et effectue les traitements nécessaires si il est trouvé, à savoir supprimer l'epsilon-transition, et ajouter les transitions depuis l'état destination de cette dernière, à l'état source. A la fin, on supprime tous les états et transitions "inutiles" (état puit, inaccessible, stérile, isolé ...etc).

1.3. Déterminisation

Cet algorithme prend la structure générale d'un parcours en largeur d'un graphe, en effet, il parcourt tous les états en utilisant une pile et une liste d'états visités. Pour chaque état, on parcourt les transitions possibles avec chaque lettre de l'alphabet.

A chaque étape, on vérifie d'abord si des transitions sont possibles, si oui on vérifie si plusieurs transitions sont possibles en lisant une seule lettre, ce qui signifie qu'on a trouvé le non-déterminisme dans l'automate, on effectue les traitements selon l'algorithme de déterminisation connu, en ajoutant les nouveaux états à l'automate et à la pile des états à traiter.

1.4. Minimisation

Il existe plusieurs algorithmes de minimisation d'automates, dans notre cas on a utilisé l'algorithme de *Moore,* le principe est le suivant: créer un ensemble initial, contenant deux ensemble, le premier avec les états finaux, et le deuxième avec les autres états.

On parcourt les ensembles un par un tant que des changements sont effectués. Un changement survient lorsqu'on deux états qui vont vers des ensembles différents en lisant un caractère de l'alphabet, dans ce cas, on les sépare en créant un nouvel ensemble, et en reprenant le parcours depuis le début du grand ensemble.

1.5. Equivalence

Pour vérifier l'équivalence de deux automates, il est possible de générer leurs automates minimaux et de les comparer.

Les automates générés peuvent avoir des noms d'états ou ordres différents, ce qui fait qu'il faut plutôt vérifier le type de ces états, en effet la logique suivie est la suivante:

- Si les automates ont le même alphabet et le même nombre d'états, on commence les traitements.
- On commence par les états initiaux des deux automates.
- Pour chaque paire d'états, récupérer les transitions possibles avec chaque lettre de l'alphabet.
- Si à un moment donné, on trouve deux transitions dans les deux automates qui mènent vers des états destinations de types différents (le type peut être initial, final ou autre)
- On arrête car les automates ne sont pas équivalents, sinon on continue.

1.6. Algorithme de Thompson

L'algorithme de Thompson implémenté prend en paramètre une expression régulière sous sa forme postfixe, et la transforme en automate suivant les règles de Thompson. Ci-dessous les étapes de passage d'une expression régulière à la forme postfixe, puis à un automate:

- **Transformation d'une expression régulière en forme postfixe:**

Note: L'expression régulière donnée doit absolument expliciter les concaténations pour qu'elle soit valide, le caractère choisi pour la concaténation est le point, mais peut être modifié facilement dans l'algorithme.

Exemple: *a(a+b)*b* ne sera pas validé et générera une erreur

a.(a+b).b* : Donnera un bon résultat (*aab+*.b.)*

L'algorithme de transformation en forme postfixe choisi et le ***shunting-yard-algorithm***. Le principe est le suivant:

On dispose une expression régulière, d'une pile de sortie et d'une pile d'opérateurs initialement vides.

On défini un ordre de priorité des opérateurs (* plus prioritaires que + et .)

- Si le caractère lu est de **l'alphabet**: l'ajouter à la pile de sortie
- Si le caractère est un **opérateur**:
  - Si un **opérateur** est au sommet de la pile d'opérateurs et sa priorité est **supérieure** ou **égale** à celle du caractère lu: dépiler le sommet de la pile et l'ajouter à la pile de sortie, continuer tant que la condition est vraie.
  - Si une **parenthèse ouvrante** est au sommet de la pile d'opérateurs: Empiler le caractère lu dans la pile d'opérateurs.
- Si le caractère lu est une **parenthèse ouvrante**: l'empiler dans la pile des opérateurs.
- Si le caractère lu est une **parenthèse fermante**: Dépiler tous les opérateurs depuis la pile, et les ajouter à la pile de sortie, jusqu'à arriver à une parenthèse ouvrante, ensuite supprimer les deux parenthèses.
- Si on a lu tous les caractères et la pile des opérateurs n'est **pas vide,** les dépiler et les ajouter à la pile de sortie

- **Passage de la forme postfixe à un automate:**

Pour passer de la forme postfixe à un automate, on a principalement besoin d'une pile qui va contenir l'évolution du graphe au fur et à mesure qu'on avance dans l'expression. A chaque caractère rencontré, on effectue un traitement comme suit:

- Caractère de l'alphabet: on génère un automate avec 2 états en lisant le caractère, et on l'empile.

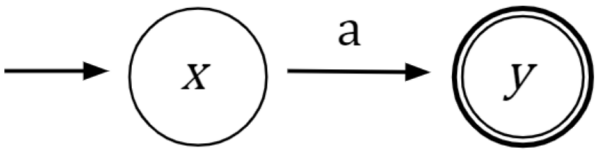


Figure 2: Transformation d'un caractère

- ***Union*** +: On dépile les deux derniers automates de la pile, on crée un nouvel état initial qui ira vers les états initiaux des automates dépilés en lisant epsilon, et un état final qui recevra une transition des anciens états finaux toujours avec epsilon. Puis on l'empile.

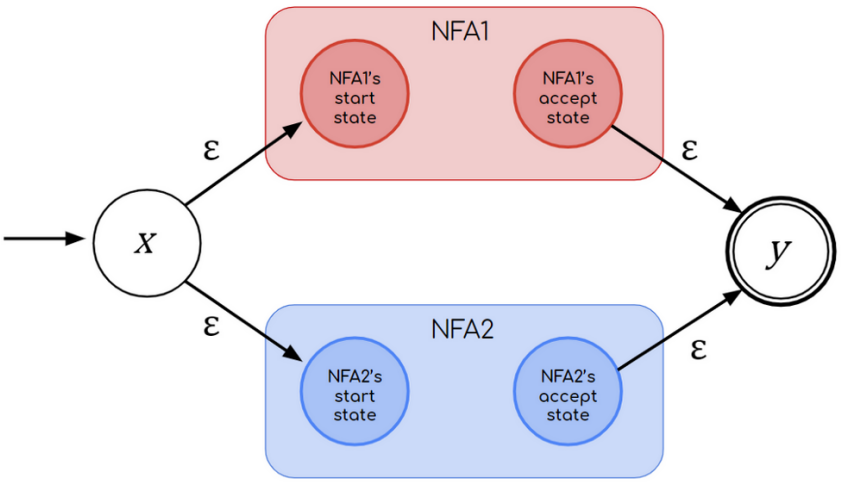


Figure 3: Transformation de l'union

- ***Concaténation .***: On dépile les deux derniers automates de la pile et on fusionne l'état final du premier avec l'état initial du deuxième, pour avoir un nouvel automate et l'empiler.

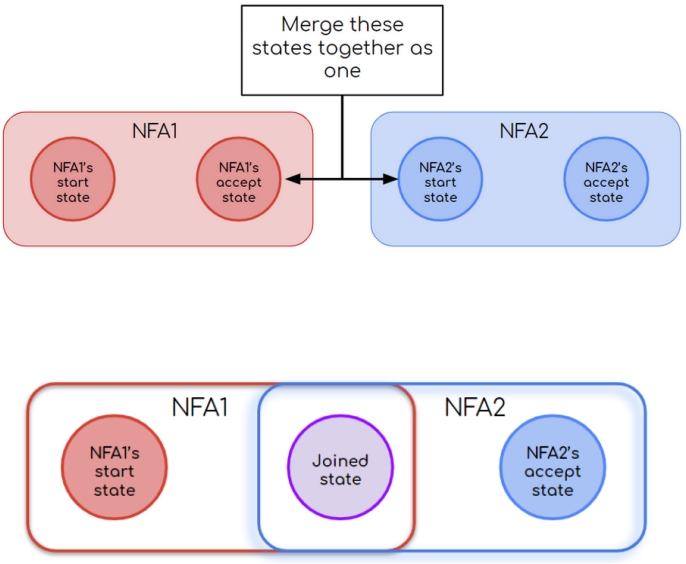


Figure 4: Transformation de la concaténation

- ***Fermeture *:*** Dépiler le sommet de la pile, ajouter une epsilon-transition entre l'état final et l'état initial, puis les remplacer par un nouvel état initial relié à un nouvel état final par une epsilon-transition. Empiler le nouvel automate.

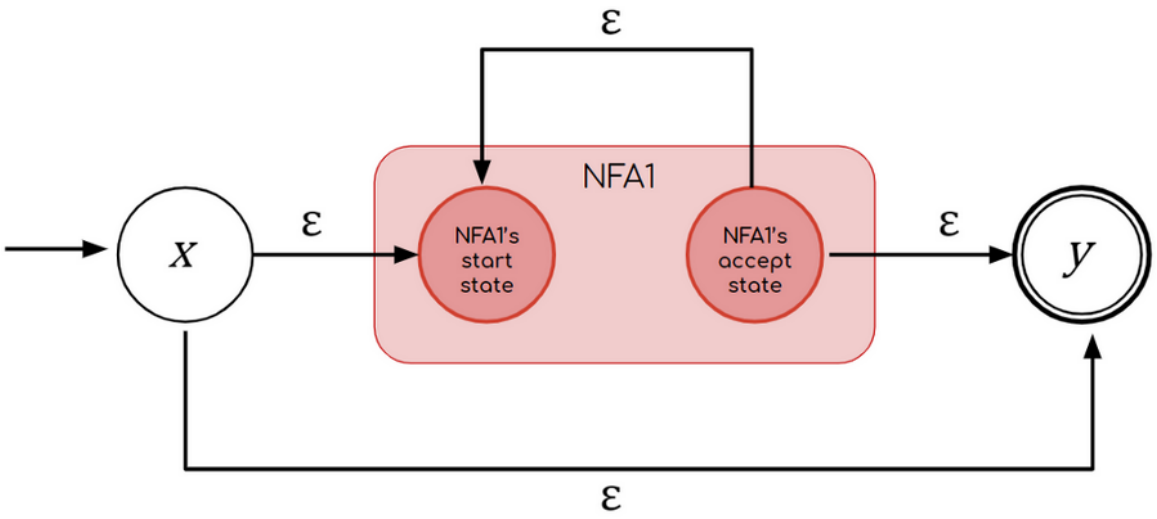


Figure 5: Transformation d'une étoile de Kleene

2. Représentation d'un automate via une image

L'objectif de cette partie est de pouvoir passer d'un automate sous la forme d'un objet de la classe *Graph* à une représentation graphique.

2.1. Graphviz

Graphviz est un logiciel open source de visualisation de graphes. Ces fonctions prennent des descriptions de graphes dans un langage texte simple, et réalisent des diagrammes dans des formats utiles, tels que des images (PNG, JPG), et SVG pour les pages web, PDF ou Postscript pour l'inclusion dans d'autres documents ; ou l'affichage dans un navigateur de graphiques interactif.

Graphviz dispose de nombreuses fonctionnalités utiles pour les diagrammes, telles que des options pour les couleurs, les polices, les styles de lignes, les hyperliens et les formes personnalisées.

Il utilise par défaut le format ***dot (Graph Description Language)*** pour représenter les graphes sous forme textuelle, en particulier si c'est un graphe orienté.

Graphviz est utilisable avec Python en utilisant une interface simple disponible à l'installation via ***pip***.

2.2. Graphe sous format JSON

Python permet de convertir les dictionnaire vers des fichiers json et vice versa via le module ***json***. Ce format est connu pour être simple et intuitif à utiliser, ce qui

nous permet de faire la conversion d'un automate en image qu'à la fin de tous les traitements nécessaires en utilisant l'interface de graphviz.

2.3. Utilisation de graphviz

Après avoir généré ou lu un automate, et l'avoir écrit dans un fichier JSON dans le dossier */Results/*. Il suffit de lire ce fichier, et de récupérer les éléments de l'automate pour les transformer via la classe ***Diagraph*** de graphviz.

Son constructeur prend en paramètre le type d'automate à implémenter, le chemin de sortie, le nom du fichier et le format. Le type étant dans notre cas ***"finite state machine",*** et le reste des paramètres étant optionnels.

Malheureusement, cette interface ne permet pas de représenter explicitement un état initial, donc on s'est contenté pour l'instant de le présenter sous forme de flèche (***shape='rarrow')***, différente des autres états.

Les états finaux sont représentés par deux cercles (***shape='doublecircle'),*** et les autres états par un cercle (***shape='circle')***

Les transitions sont ajoutées en utilisant la fonction ***edge,*** qui prend en paramètre dans l'ordre: l'état de départ, l'état d'arrivée, la lettre à lire.

Enfin, on enregistre et affiche l'image de l'automate avec la fonction ***view.***

3. Génération automatique d'automates

L'automatisation des tâches est toujours un sujet complexe en informatique, en particulier lorsqu'il s'agit de générer automatiquement des automates. Le but final est de pouvoir avec ou sans l'aide humaine, générer des langages qui sont pertinents, et pas trop chargés, et qui peuvent l'objet d'exercices d'examens.

La démarche qu'on a suivi est simple, on utilise le module ***random*** pour générer un nombre d'états et de transitions aléatoires compris dans des intervalles choisies manuellement, puis lister toutes les combinaisons de transitions possibles et les filtrer en en supprimant un nombre donné, ce dernier aussi sera généré aléatoirement dans une intervalle précise.

Cependant, faire ça sans vérification génère des automates qui ne peuvent subir d'autres traitements, et par conséquent sont inutiles. Donc on continue de générer des automates tant que l'une des conditions suivantes est réalisée:
L'automate contient un des états suivants:

- Etat initial sans transition sortante, sans compter les boucles, donc pas de possibilité d'avancement.
- État final sans transition entrante, sans compter les boucles, donc pas de possibilité d'arrêt.
- Etat sans transition sortante, sans compter les boucles, mais n'est pas final. Donc c'est un état puit.
- Etat sans transition entrante ni sortante (état isolé)
- Etat sans transition entrante, avec des transitions sortantes, mais n'est pas initial, donc cet état serait inaccessible.

Cette liste de conditions n'est pas exhaustive, on pourrait détecter d'autres cas d'états inutiles à considérer, surtout par la possibilité d'effectuer des traitements dessus. Mais l'algorithme génère tout de même des résultats acceptables.

4. Génération de sujets d'examen

Cette partie étant optionnelle, nous étions quand même très intéressés par le principe, en effet, ça nous a permis de redécouvrir la syntaxe ***LaTeX*** et de voir à peu près comment nos enseignants arrivent à générer des sujets d'examens pour toute la section dans les autres UE.

4.1. PyLaTeX

PyLaTeX est une bibliothèque Python pour la création et la compilation de fichiers LaTeX. Le but de cette bibliothèque est d'être une interface facile, mais extensible, entre Python et LaTeX.

PyLaTeX a deux utilisations très différentes : la génération de pdfs complets et la génération de snippets LaTeX. La génération de pdfs complets est surtout utile lorsque tout le texte que le pdf doit contenir est généré par python, par exemple l'exportation des données d'une base de données. Les snippets sont utiles lorsqu'une partie du texte doit encore être écrite à la main, mais que certaines choses peuvent être générées automatiquement, par exemple écrire un rapport avec quelques graphiques *matplotlib*, ou comme dans notre cas, générer des questions différentes pour chaque étudiant.

4.2. Utilisation

Le fichier ***latex_gen.py*** contient la logique pour générer des sujets d'examens et leurs corrections, bien que ce soit qu'un prototype pour le moment, les parties les plus essentielles sont bien présentes et sont:

- Création de l'en-tête contenant les informations principales sur l'examen
- Création d'un exercice simple d'automates, en lisant un des automates du dossier */Results/*. Les exercices doivent suivre la convention de nommage: "**ExN-M.png",** où **N** est le numéro de l'exercice, et **M** le numéro de la question. Les corrections doivent être suivies par le nom de l'opération effectuée de la manière suivante:
  "ExN-M-K.png" où **K** = **'-det'** ou **'-min'** ou **'-thompson'** ou **'-eps'**
- Création de la correction de l'exercice, suivant les traitements de l'image sélectionnée précédemment.

RÉSULTATS ET PERSPECTIVES

1. Test des algorithmes et de la visualisation

Pour tester les algorithmes de traitement d'automates, nous allons prendre des automates traités à la main, et les comparer aux résultats de notre projet.

- ***Exemple 1: Déterminisation et minimisation***

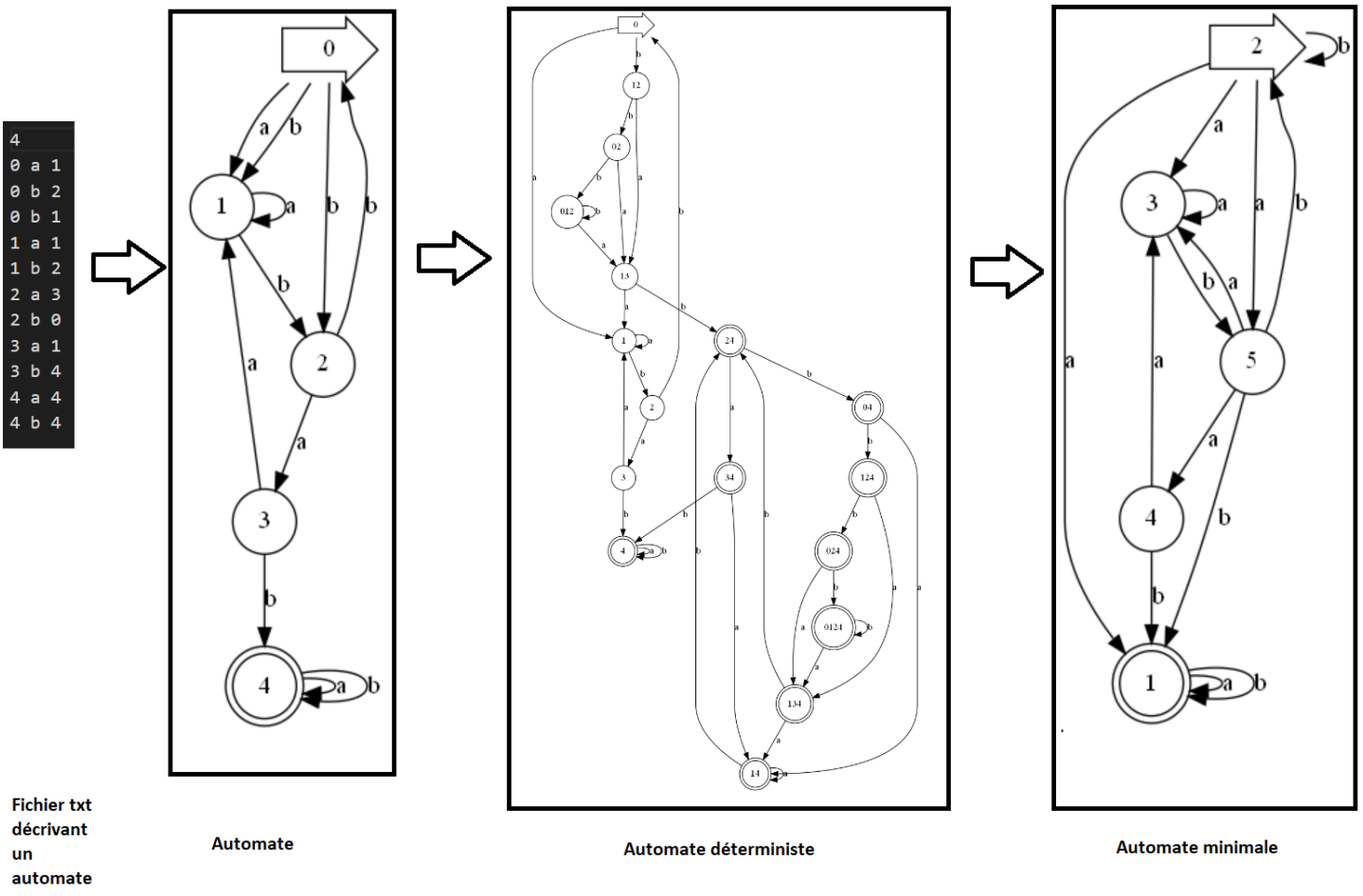


Figure 6: Exemple de déterminisation et minimisation

- ***Exemple 2: Automate de Thompson puis synchronisation***

En utilisant l'expression "**a.(a+b)*.b" :**

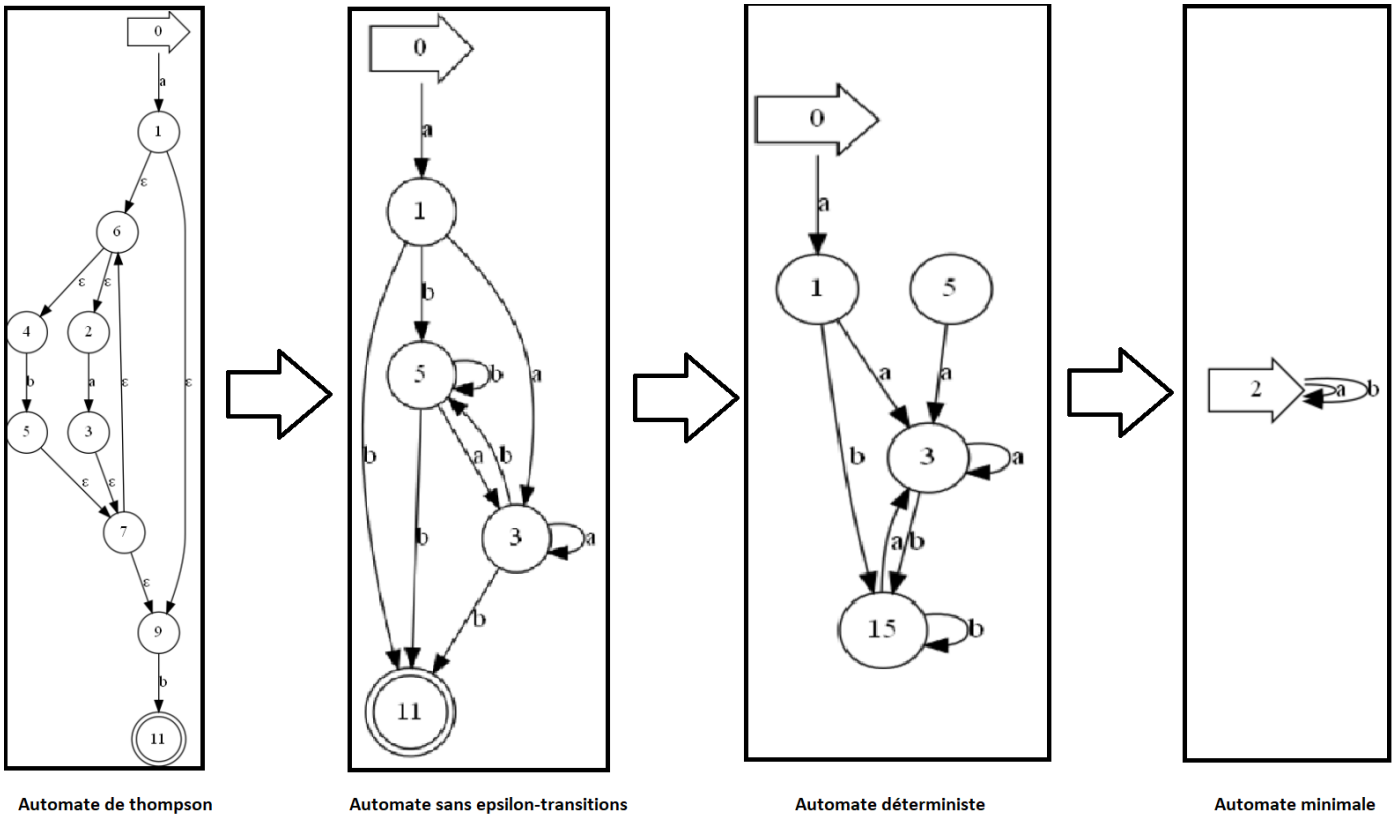


Figure 7: Exemple d'automate de Thompson

2. Génération automatique de langages

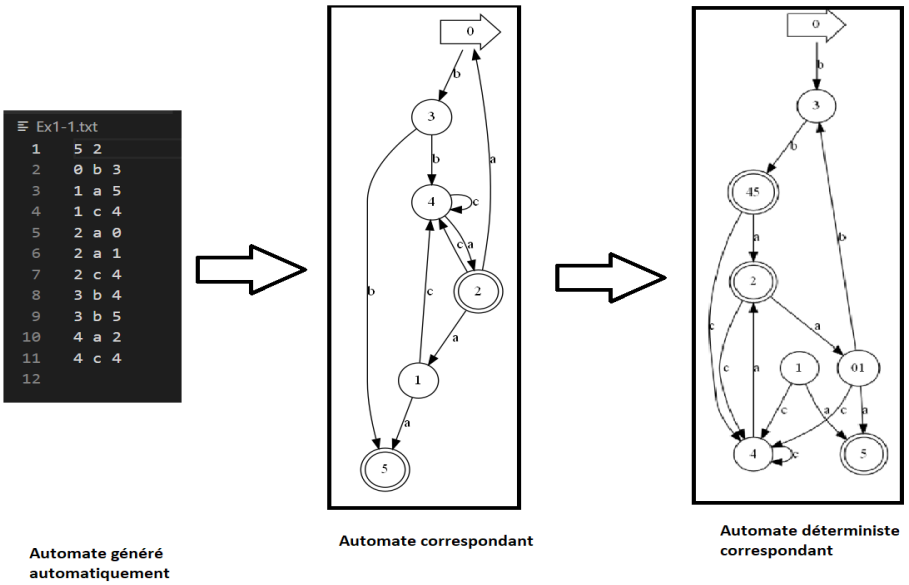


Figure 8: Exemple de langage généré automatiquement

3. Génération d'exercices d'examens

- ***Exemple d'un sujet d'examen généré automatiquement, ainsi que sa correction***

Universite Cergy Pontoise
irenee.briquel@gmail.com

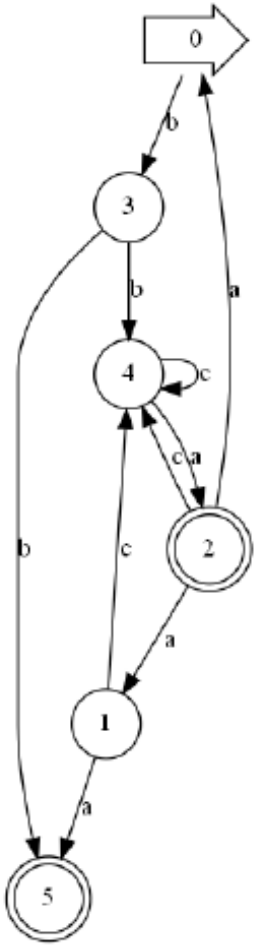
Annee 2021

Automates et langages reguliers
L2

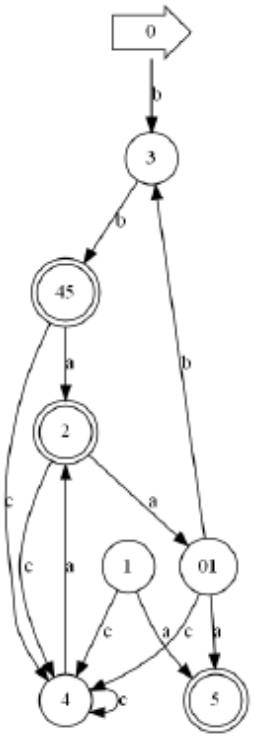
Examen de deuxieme session - Langages et automates
Mardi 16 avril 2021

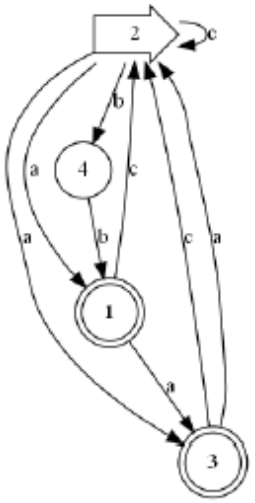
1 Exercice 1

1.1 Soit l'automate suivant:



- Trouver l'automate deterministe.
- Trouver l'automate minimal.
- Trouver l'automate de thompson de l'expression: a(a+b)*b


0
b
3
b
45
a
b
2
c
a
c
a
1
01
c
a
c
a
4
c
5


2
c
b
a
4
c
a
b
c
a
1
a
3

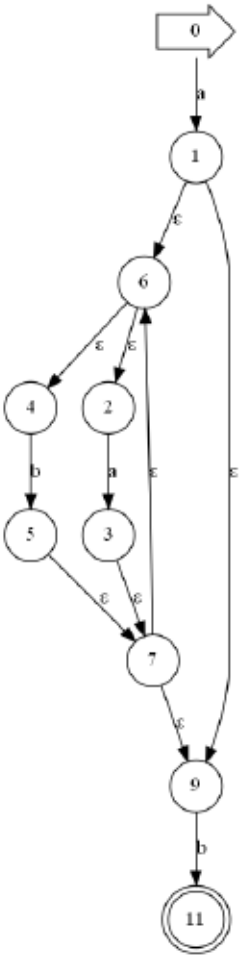


Figure 9: Exemple d'examen et sa correction

CONCLUSION

En conclusion, ce projet rassemble plusieurs notions intéressantes et a permis la réalisation de tous les points demandés.

Ce dernier a été très enrichissante, il nous a permis de redécouvrir la syntaxe de Python et d'utiliser des librairies externes créées par la communauté, il nous a aussi permis de revoir les notions d'automates ainsi que leurs traitements et de passer à la pratique en réalisant quelque chose de concret et technique en les utilisant.

Nous avons conscience des axes d'amélioration possibles et pensons vraiment que ce projet peut aboutir à donner des résultats solides et fiables, mais nous pensons cependant que ses résultats sont satisfaisants surtout pour ce qu'il a pu nous apporter en tant qu'étudiants.

CERGY PARIS UNIVERSITÉ

RÉFÉRENCES

- https://medium.com/swlh/automata-theory-in-python-part-1-deterministic-finite-automata-95d7c4a711f5
- https://github.com/iistrate/DFA
- https://automatetheboringstuff.com/
- https://realpython.com/working-with-files-in-python/#pythons-with-open-as-pattern
- https://www.momirandum.com/automates-finis/Automatefini1.html
- Groups, graphs, languages, automata, games and second-order monadic logic Tullio Ceccherini-Silbersteina, Michel Coornaertb, Francesca Fiorenzic, Paul E.Schuppd
- Frédérique Bassino, Cyril Nicaud. Enumeration and random generation of accessible automata. Theoretical Computer Science, Elsevier, 2007, 381, pp.86-104. hal-004597122
- http://www.desmontils.net/emiage/Module209EMiage/c5/Ch5_9.html
- https://jeltef.github.io/PyLaTeX/current/index.html
- https://graphviz.readthedocs.io/en/stable/
- https://medium.com/swlh/visualizing-thompsons-construction-algorithm-for-nfas-step-by-step-f92ef378581b