TP2 : Automates

Par :
Jaafar Kaoussarani (1932805)
Mohamed Khairallah Gharbi (1837067)
Ibrahim Fradj (1890887)

Groupe 2

École Polytechnique de Montréal

LOG2810

2 avril 2019

1— Introduction :

Une des pistes de recherches les plus importantes des sciences informatiques est la conception des automates. Plusieurs entreprises ont pour mission de développer des programmes à apprentissage automatique. Dû à la prospérité de ce sous-secteur de l'informatique, la compétition entre ces compagnies est extrêmement élevée. Ceci fait en sorte que les automates ne cessent de croître en intelligence et en performance. Il s'ensuit que l'opportunité d'étudier les concepts de base des automates dès notre deuxième année au baccalauréat est très intéressante. Cette thématique est justement l'idée derrière ce laboratoire. Ce dernier consiste en l'écriture d'un logiciel étant capable de parcourir un labyrinthe en lisant les règles grammaticales d'un langage formel. Pour se faire, nous établissons que chaque porte du labyrinthe possède son propre fichier texte contenant les règles de grammaire associées ainsi qu'une liste de portes accessibles (et leurs mots de passe). Nous construisons par la suite un automate étant capable de vérifier s'il est possible de générer un mot de passe à partir de la grammaire donnée et donc de savoir si une porte est accessible ou non. Tout cela fera en sorte que l'utilisateur est capable de naviguer le labyrinthe.

Ce rapport présentera la conception du programme, l'implémentation de ses fonctionnalités, ainsi que les difficultés rencontrées et leurs solutions.

2— Travaux :

Contrairement au dernier TP, où nous avions décidé d'utiliser un mélange de la programmation orientée objet et séquentielle, nous utiliserons une approche purement séquentielle. Ceci nous permet d'avoir une meilleure idée de la structure du code et d'assurer que chaque développeur pourrait écrire du code indépendamment sans que le style de programmation change trop. De plus, nous avons décidé d'inclure toutes les variables et listes globales en tête de fichier afin que tout le monde puisse se rencontre de l'ajout d'une nouvelle. Ceci fait en sorte que l'on évite de la confusion lors de l'écriture et lors du débogage. Finalement, le fait que l'on utilise une série de listes pour contenir nos informations n'est pas très efficace en termes d'optimisation de mémoire et de temps d'exécution. Cependant, cela nous permet d'avoir un contrôle explicite des éléments en mémoires, ce qui peut être un grand avantage si notre communication est efficace.

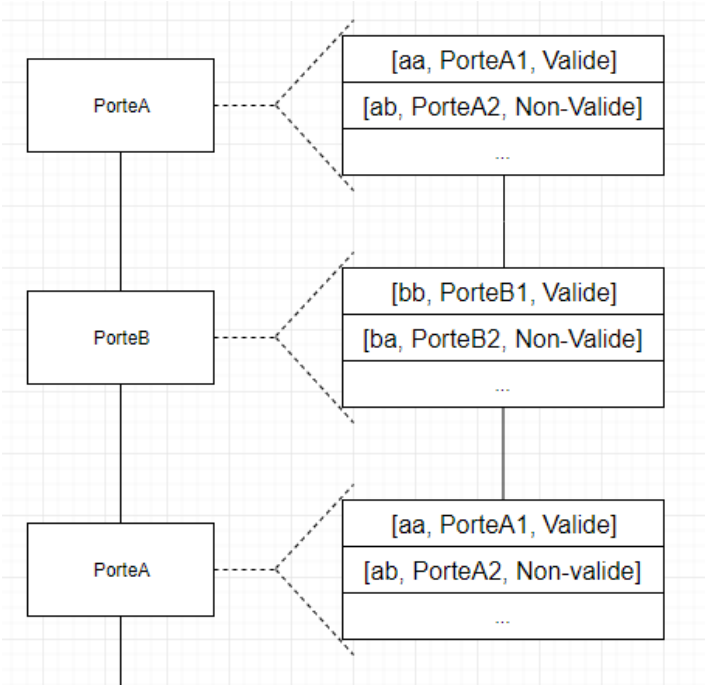
C1) La fonction « ouvrirPorte() » n'est appelé qu'une seule fois dans le main, soit lorsque l'utilisateur choisi l'option « a ». En effet, l'ouverture d'une porte dans l'option « b » ne se fait qu'après avoir affiché les portes disponibles et passer par la fonction « tryPorte () ». Cette dernière s'assure que la porte choisie par l'utilisateur est bel et

bien valide. Par la suite, nous entrons dans la fonction « ouvrirPorte() ». Elle met à jour la porte courante et lis le fichier associé à cette nouvelle porte avant d'appeler la fonction « genererAutomate() ».

C2) La fonction « affronterBoss() » appelle deux fonctions : « genererCodeBoss() » et « generLanguageBoss() ». La première lit le fichier « Boss.txt », remplit le tableau « arrayPorteBoss » contenant les portes écrites dans ce fichier. Puis, elle cherche les mots qui ont permis de passer d'une porte à la suivante en les mettant dans le tableau arrayCodeBoss. Cette fonction affiche la concaténation des mots de passe pour aller arriver au boss. La deuxième génère le langage du « Boss.txt » et les portes qui nous ont amènent au Boss. Cette fonction comporte un état « S » (initial) propre à chaque code pour toutes les portes. Un état final est écrit de la manière suivante « S-> ». Chaque transition qui mène à un état final doit aller à l'état « S ». Les états de transition sont présentés par des lettres de l'alphabet.

C3) La fonction « genererAutomate() » est responsable de la ségrégation des portes accessibles (valides vs Non valides) à partir de la grammaire de la porte courante. Pour se faire, on appelle une méthode « fillTables (array) » qui ajoute les cas terminaux à une liste (incluant la chaîne vide). Par la suite, nous utilisons une deuxième méthode nommée « validMotDePasse (array) » afin de confirmer qu'un mot de passe peut être généré.

C4) Pour la fonction « afficherLeCheminParcouru() », nous avons décidé de suivre le format présenter à l'annexe 2. Pour se faire, deux listes globales, la première contenant les portes visitées et la deuxième contenant des sous-listes de trois chaînes de caractères, sont utilisées. Ces trois chaînes sont, respectivement, le mot de passe associé à une porte accessible, la porte accessible et si le mot de passe d'une. Nous utilisons cette structure afin de simuler une sorte de dictionnaire supportant des « clés » multiples cela est représenté dans la figure suivante (une « clé » dupliquée serait tout simplement associée à une copie de la partie de droite de la précédente).



Par la suite, nous avons tout simplement itéré sur cette structure et affiché selon le format demandé. En ce qui concerne l'affichage du Boss (-- —)

C5) Pour l'affichage du menu, nous avons encore décidé de simuler un main. Pour ce faire, on écrit une fonction que nous avons décidé d'appeler « main () » et on utilise l'attribut «__name__» pour faire en sorte que le programme commence au main plutôt que vers le haut de la page. Par la suite, nous utilisons une boucle « while True » afin de nous assurer que le menu est réaffiché à chaque fois que l'option choisie par l'utilisateur a fini de s'exécuter.

3— Difficultées:

Le plus grand défi de ce TP a été l'établissement d'une stratégie de développement efficace. Cette difficulté provient du fait que, à première vue, les requis et les objectifs à atteindre semblent complexes. Notre équipe a dû discuter des attentes du travail à plusieurs reprises, et nous avons mal compris certains concepts à plusieurs reprises. De plus, il arrivait parfois que deux coéquipiers pensent avoir la même idée, alors qu'en effet ils pensaient à des choses différentes. Les restructurations ont donc été initialement simples, mais plus fréquentes que l'on aurait voulu. La solution employée est très simple : poser des questions et faire en sorte que chaque personne explique son idée à fond. Après l'avoir expliquée, les deux autres tentent de la réexpliquer. Ainsi, on peut être certains que l'on se comprend.

Une autre difficulté a été de savoir comment faire pour détecter les mots de passe valides à partir de la grammaire. En effet, nous avons d'abord essayé de déterminer comment générer une liste de tous les mots possiblent. Nous avons réalisé rapidement que cela serait extrêmement complexe et inefficace. Par la suite, nous avons décidé de générer les mots de passe en vérifiant s'ils sont valides avant de les insérer dans la liste de mots valides. Cela est plus efficace, mais toujours beaucoup trop complexe pour nos objectifs. Finalement, nous avons réalisé que la solution repose dans le fait que nous devons axer notre raisonnement en fonction des terminaux. En effet, plutôt que de déterminer les mots de passe valides en partant des non-terminaux, il est beaucoup plus efficace de tout simplement comparer les terminaux aux derniers caractères de chaque mot de passe. Nous comparons de la fin vers le début. Ainsi, nous pouvons rejeter les mots de passe non valides rapidement et avec une structure de code qui est beaucoup plus claire.

Une dernière difficulté parmi tant d'autres se trouve dans la gestion des listes. En effet, au début, il a été difficile de déterminer quelles listes utiliser et surtout, quand appeler la méthode « clear » de chacune. De plus, le cas exceptionnel ou nous avions une grammaire présentant une situation de type « S->, S-> aS » (soit le cas où une règle comporte un non-terminal pouvait faire référence à une chaîne vide dans un autre règle) nous a causé quelques problèmes. Notamment, on avait dû tenir compte de cette possibilité lors de la division des règles de grammaire en une liste (puisque nous avons utiliser la méthode split). Finalement, nous avons trouvé que la meilleure façon de gérer tout cela a été d'exécuter le programme en mode débogage et essayer de bien comprendre comment Python gère sa mémoire tout en tenant compte de la durée de vie nécessaire de toutes nos listes. Cette solution est loin d'être idéale, mais elle fonctionne bien à nos fins et ne demande pas trop de restructuration.

4 – Conclusion :

Pour conclure, ce travail nous a permis d'approfondir nos connaissances en Python, de constater de la flexibilité des automates lors de la résolution d'un problème et d'adapter nos habiletés en programmation et en gestion de projets envers la conception d'un logiciel respectant les exigences fournies. Finalement, tout cela nous a permis d'accumuler de l'expérience qui nous sera utile non seulement dans notre parcours à la Polytechnique, mais aussi pour le reste de notre carrière.