Travail Pratique #2 : Automates

LOG2810 : Structures discrètes

Rapport remis par :
Mariam Sarwat (1928777)
Stéphanie Mansour (1935595)
Yasmina Abou-Nakkoul (1897266)

Équipe: 22
Groupe : 01

École Polytechnique de Montréal
Date de remise (06-04-2019)

1. Introduction

Dans le cadre du cours Structures Discrètes, un programme à compléter nous a été donné. L'objectif de ce programme était d'appliquer les notions théoriques sur les automates et la théorie des langages en créant le jeu « Dont vous êtes le héros » . Ce jeu est divisé en deux parties principales. Premièrement, il permet à l'utilisateur de franchir un labyrinthe de portes jusqu'à ce qu'il atteigne le boss. Deuxièmement, une fois là-bas, il doit vaincre le boss pour remporter la partie. L'utilisateur est capable de conquérir le boss s'il a suivi exactement le même chemin trouvé dans Boss.txt, sinon le jeu est perdu. Chaque fichier de porte contient trois informations principales: les règles qui génèrent notre automate, des mots de passe (valide ou non), ainsi qu'une porte associer à chacun. À chaque porte traverser, l'agent indique à l'utilisateur le contenu du dernier événement (mot de passe, la porte associer ainsi que la validité du mot de passe). Dans ce rapport est expliqué la façon dont nous avons abordé les différents tâches données, la difficulté à les compléter et une conclusion résumant l'utilité de ce laboratoire.

2. Présentation des travaux

Avant tout, il est important de prendre en compte que nous avons changé la façon dont les informations dans les fichiers sont déclarées:

Version original	Notre version

{
S->aA, S->bA, S->c, A->aS, A->bS,
S->d,S->e
}
aaab Porte8
abab Porte10
c Porte2
d Porte15
bbbbb Porte18
e Porte9
	S aA S bA S c_ A aS A bS S d_ S e_
	; ; ;
	aaab Porte8
	abab Porte10
	c Porte2
	d Porte15
	bbbbb Porte18
	e Porte9

Tableau 1: exemple du format de nos fichiers avec Porte1

Pour réaliser ce laboratoire, nous avons utilisé une approche orientée objet en C++. Pour mieux comprendre la façon dont nous avons abordé cette tâche, ci-dessous se trouve les différentes classes créer expliquant leurs rôles et les méthodes relier.

Automate

La première classe créée est celle d'Automate qui possède plusieurs attributs:

- vector<string> porte_correspondante: stocke les portes correspondantes à chaque mot de passe dans un fichier. À chaque nouvelle lecture d'une porte, ce vecteur est effancé. Il contient les portes correspondantes d'une porte à la fois.
- vector<char> etat_terminaux: ce vecteur est en charge de conserver les lettres qui correspondent à des états terminaux. Par exemple: A b_, lors de la lecture du fichier, si on détecte le symbole '_', cela correspond a un état terminaux. Par conséquent, 'b' sera stocké dans ce vecteur.
- vector<string> mot_de_passe: garde en mémoire les mots de passe qui se trouvent dans une porte lors de la lecture d'un fichier. À chaque nouvelle lecture d'une porte, ce vecteur est effacé. En d'autres mot, il contient les mots de passe d'une seule porte à la fois.
- multimap<char, pair<char, char>> regles : un attribut très important, car il joue un role principale pour générer nos automate. Il stocke l'état courant dans le premier char (regles.first), ensuite, concernant le "pair", la lettre resortit (regles.second.firs), soit un 'a', 'b', 'c', 'd' ou 'e', et finalement l'état prochain (regles.second.second).
    - Si nous prenons comme exemple: S aB
        - 'S' sera pris comme l'état courant
        - 'A' sera la lettre ressortit
        - 'B' sera l'état prochain
- vector<string> mpValide et vector<string>pcValide: ceux-ci contiennent les mots de passe valides ainsi que les portes correspondantes valides, respectivement.

La classe Automate, autre que les méthodes d'accès (getters) et de modifications (setters) des attributs énumérés ci-dessus, possède les fonctions suivantes :

genererAutomate(string fichier, string provenance, string porte): cette fonction s'occupe de lire les fichiers textes qui contiennent chacune les informations sur un automates, les mots de passe et les portes correspondantes d'une porte spécifique. Elle prend en paramètre le fichier à lire (avec extension .txt), la provenance de l'appel de la fonction (si elle provient de l'appel à ouvrir une porte à l'aide de l'option 'b', de réinitialiser le parcours du labyrinthe avec l'option 'a', ou d'affronter le boss), et finalement, la porte dont on veut générer l'automate.

valideMotDePasse(vector <string> mpAValider, vector <string> pcAValider, multimap<char, pair<char, char>> reglesLangue): puisque la validation des mots de passe est

une des fonctionnalités principales de la création du jeux, nous avons jugé qu'il serait mieux de créer une fonction distincte. Elle prend en paramètre un vecteur de mot de passe à valider, les portes correspondantes à valider (chaque index des mots de passe à valider correspond au même index des portes correspondants) ainsi que le multimap contenant les règles de l'automate.

vector<string> ouvrirPorte(string fichier): cette fonction s'occupe de lire les fichiers textes pour créer l'automate et pour valider les mots de passe en faisant appel aux deux fonctions expliquées ci-dessus. Elle retourne un vecteur qui contient les mots de passe valide (chaque index des mots de passe valides correspond au même index des portes correspondants qu'on peut ouvrir).

Boss

La seconde classe est celle de Boss qui contient les attributs suivants:

- Automate automate: un objet de type Automate qui nous permet de faire appel à ses fonctions appropriés, tel *valideMotDePasse*.
- multimap<char, pair <char, char> > langageFinal: celui-ci est similaire au multimap expliqué pour la class Automate (état courant, lettre ressortit, état prochain), par contre, il correspond spécifiquement à l'automate obtenu pour affronter le boss.
- string mdpFinal: correspond à la concaténation des mots de passe utiliser pour affronter le boss.
- string mdpBossAffichage: permet de faire l'affichage des mots de passes concaténé sans prendre en considération les modification fait dans les fichiers. Soit l'ajout du mot "vide" lorsqu'un mot de passe existe pas. Ceci donne le mot de passe qui sera affiché lors de l'événement Boss.
- vector<string> cheminPris : contient l'information du chemin (les portes) parcouru pour arriver de la Porte1 au boss.
- vector<Automate> porte: contient toute l'information sur l'automate de toutes les portes (porte 1 à 20)
- bool theSame: ce boolean nous permet de determine si le chemin pris par l'agent est le même que celui qui se trouve dans Boss.

Aussi importante que la classe Automate, la classe Boss contient des méthodes principales pour jouer au jeux « Dont vous êtes le héros ». Autres que les méthodes d'accès et de modifications, nous pouvons y trouver:

videCheminPris(), videMdpBossAffichage(), videMdpFinale(): ces trois fonctions sont en charge de vider le chemin pris par l'agent, le mot de passe pour affronter le Boss et le mot de

passe final (la concaténation des mots de passe utilisé), respectivement. Ces fonctions sont appelées lorsque nous nous trouvons dans un gouffre et devons réinitialiser le labyrinthe à l'aide de l'option 'a' ou lorsqu'un débute le jeux.

fillPorte(): cette fonction est utilisée pour remplir l'attribut vector<Automate> porte expliqué ci-dessus.

afficherLeCheminParcouru(vector <string> porteChoisi): cette fonction affiche le chemin parcouru. Elle prend en paramètre un vecteur contenant les portes parcourues jusqu'à présent et affiche:

- Si on parle d'une porte:
  - Le choix d'une porte
  - Les mots de passe, la porte associée et la validité des mots de passe
  - Si la porte est un gouffre ou non
- Si on parle du boss:
  - Le choix du boss
  - La concaténation des mots de passe utilisés pour affronter le boss
  - Si le boss est vaincu ou non

affronterBoss(string fichier, string mdp): cette fonction s'occupe de lire le fichier texte Boss.txt qui contient le chemin de porte nécessaire pour vaincre le boss. Elle prend en charge la tâche de générer l'automate du chemin qui se trouve dans ce dernier fichier, soit la concaténation de tous les productions des portes qui forment le chemin. De plus, elle valide la concaténation des mots de passe utilisés pour affronter le boss.

Main

Dans le main on retrouve le menu principal du programme. Ce menu a été créé à l'aide d'un switch-case où chaque cas est une option du menu.

Cas 'a': permet à l'utilisateur d'entrer dans le labyrinthe et d'ouvrir la première porte: Porte1. Ce cas réinitialise toutes les structures de données, puisque si on s'y retrouve, soit c'est la première fois que l'utilisateur entame le labyrinthe, soit il a échoué la fois précédente et donc dois recommencer.

Cas 'b': En sélectionnant cette option, cela permet à l'utilisateur d'ouvrir une porte. L'agent choisit une porte aléatoire à ouvrir et ensuite fait appel a *vector<string> ouvrirPorte(string fichier)*. À chaque fois que l'utilisateur choisit l'option 'b', l'agent affiche le contenu du dernier événement à l'aide de la fonction *afficherLeCheminParcouru(vector <string> porteChoisi)*. Il est important de noter que cette option ne peut pas être sélectionné

avant l'option 'a', sinon un message indiquera à l'utilisateur l'invalidité de cette option. De plus, si l'utilisateur tombe dans un gauffre, il devra réinitialiser le labyrinthe en sélectionnant l'option 'a'.

Cas 'c': permet d'afficher les choix de porte effectués par l'agent en faisant appel à la fonction *afficherLeCheminParcouru(vector <string> porteChoisi)*.

Cas 'd': cette option est affichée pour rendre possible à l'adversaire de quitter le programme.

Le programme réaffiche le menu tant que l'option 'd' n'a pas été choisie. De plus, le program affiche le menu de nouveau si l'utilisateur entre un index invalide.

Diagramme de classes

Ci-dessous se trouve le diagramme de classe présentant les relations entre les différentes classes de notre programme:

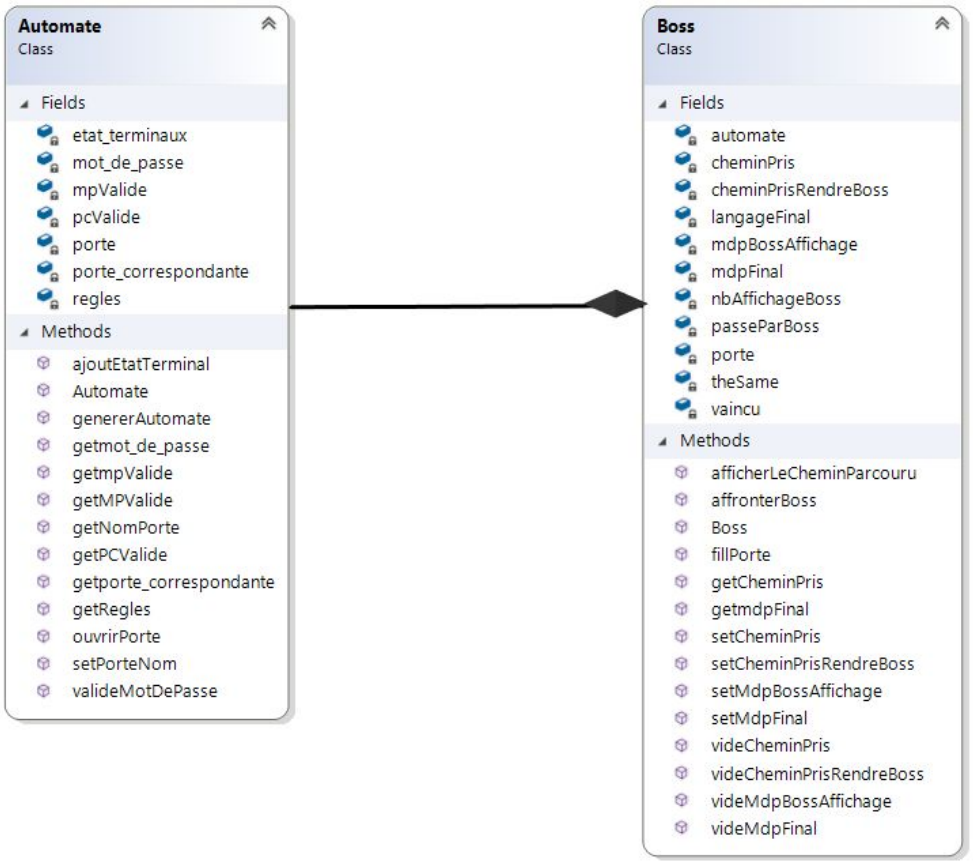

Figure 1 : Diagramme de classes

3. Difficultés rencontrées

Une des difficultés que nous avons rencontrés a été la conception du programme. En effet, le vrai difficulté se trouvait dans la génération de l'automate. Nous avons toutes eu des idées différentes, une aussi valide que l'autre. Suite à cette confusion, nous avons décidé de visualiser chaque option sur papier et d'établir quelle était la meilleure option.

4. Conclusion

Ce laboratoire nous a été utile pour appliquer la théorie du cours, tel les notions sur les automates et la théorie des langages. Contrairement au premier laboratoire, la densité de celui-ci était d'un niveau approprié. En résumé, même s'il s'agissait d'une situation complexe à décortiquer, cela nous a vraiment aidés à voir comment appliquer ces notions dans notre carrière d'ingénieur.