LOG2810

TP2 – Automates

Par

Gabriel Tagliabracci, 1935775

Roman Zhornytskiy, 1899786

Kevin Ciliento, 1933571

2 avril 2019

Polytechnique Montréal

**À faire :**

* Les classes
* Les difficultés
* ~~La conclusion~~ (à faire dernière révision + temps pour c2)

**Introduction**

(p-t ajouté dequoi de plus) Dans le but de se familiariser avec les automates, les graphes et la génération de langage, ce TP permet de créer une version modifiée du jeu « Dont vous êtes le héros » dans lequel le participant doit choisir une série de portes afin d’arriver à battre un « boss » avec les mots de passe accumulés depuis la première porte. Un agent est créé au début du labyrinthe et celui-ci génère un automate qui lit les mots de passe des portes suivantes afin de trouver les portes valides.

**Présentation des travaux**

Tout d’abord, pour ce TP, nous avons décidé de coder nos fichiers en C++, ce langage nous étant le plus familier. Nous avons ensuite séparé les différentes tâches à effectuer en plusieurs classes : agent.cpp, door.cpp, automate.cpp et nodeTree.cpp.

**agent.cpp**

Cette classe représente l’agent s’occupant de

**door.cpp**

La classe door.cpp représente les portes que l’agent doit parcourir dans le labyrinthe. Chaque porte consiste d’une série de règle de grammaire permettant de créer une série de mots de passe valides ainsi que les portes adjacentes avec leur propre mot de passe. Dans cette classe, il y une fonction canOpen() qui vérifie si la porte est présente dans la liste des portes pouvant être ouvertes. isValid() est la méthode vérifiant la validité d’une certaine porte par la variable « validity ». Cette variable est changée dans la fonction validate() qui vérifie si le mot de passe de la porte est valide.

Cette classe a aussi des fonctions pour lire les fichiers Porte#.txt comme readFile() et readRule() qui lise respectivement le fichier .txt et les règles de grammaires présentes dans ce même fichier. Finalement, la fonction readNextDoor() permet de trouver une porte adjacente et de l’insérer dans un map passMap\_ avec son mot de passe. Tous les map de portes adjacentes de la porte présente sont ensuite intégrés dans un multipmap doorMap\_.

**automate.cpp**

La classe automate.cpp contient les différentes méthodes pour parcourir le graphe NodeTree. Elle possède tout d’abord la fonction generateAutomate() qui prend les règles présentes dans la porte pour créer le graphe. Les états initiaux sont insérés en tant que noyaux et, selon l’état final de la règle, ceux-ci sont insérés en tant que « child » du noyau. Cela créer donc un graphe orienté dans lequel les mots de passe traversent pour être valider. La fonction validatePassword() commence à partir du début de l’arbre et le parcoure selon la valeur des différentes règles. Si le mot de passe se termine sur un noyau terminal (qui n’a pas de « child »), alors le mot de passe est validé avec la fonction « validate() » dans Door.cpp. Il est invalide s’il se termine sur n’importe quels autres noyaux ayant des « childs ».

La fonction validatePasswords() prend tous les mots de passe des portes adjacentes et utilise la fonction validatePassword() pour trouver les mots de passe valides. Finalement, la classe peut trouver la valeur d’un « Edge » avec la méthode trouverLettre().

**nodeTree.cpp**

La classe nodeTree.cpp permet de créer un graphe contenant les possibilités de mots de passe valides pouvant être créer par les règles de grammaire de la porte. La classe contient deux autres structures pour sa création : Node et Edge. « Node » représente un état dans les règles (soit initial ou final) et forme un noyau dans le graphe. Ces noyaux sont ensuite reliés entre eux par des « Edge » qui prennent deux « Node » comme noyau de départ et noyau d’arrivée afin de créer une branche orientée. Ces branches contiennent une valeur qui est la lettre du mot de passe dans les règles.

Cette classe contient la fonction find() qui cherche à travers le graphe pour trouver si la paire entrée en paramètre est présente. Elle contient aussi la méthode insert() qui ajoute un nouveau « Edge » dans le vecteur contenant les « Edge » ainsi qu’un nouveau noyau s’il n’est pas déjà présent dans le graphe. Ces deux structures sont ensuite intégrées dans une paire dans le graphe.

**main.cpp**

Finalement, le main sert principalement à afficher l’interface du jeu donnant plusieurs options au joueur : entrer dans le labyrinthe, ouvrir une porte, afficher le chemin parcouru ou quitter. L’option « a » permet de placer l’agent à la porte 1 afin de commencer le labyrinthe. Cette option peut être sélectionnée durant le jeu ou quand le joueur arrive dans un gouffre. Le chemin déjà parcouru est gardé en mémoire même si le joueur recommence au début. L’option « b » utilise la méthode openDoor() pour ouvrir une porte sélectionnée par le joueur. Cette option ne permet d’ouvrir que les portes dont le mot de passe a été validé. L’option « c » affiche le chemin de porte parcouru par le joueur depuis le début de sa partie. Elle permet aussi d’afficher le chemin en mémoire après un recommencement de partie. L’option « d » permet de quitter le jeu.

Figure 1. Diagramme de classe du logiciel.

**Difficultés rencontrées lors de l’élaboration du TP**

Durant la conception du code, plusieurs problèmes ont été rencontrés à différentes étapes. Tout d’abord, en choisissant l’architecture, nous avions prévu intégrer toutes nos méthodes dans une seule grosse classe, mais nous nous sommes rendu compte que cela ne ferait aucun sens. Pour régler cela, nous avons décidé de répartir les fonctions en plusieurs classes plus petites et d’appliquer l’encapsulation. Cela a aussi permis d’utiliser des fonctions clés qui permettaient aux fonctionnalités de travailler en arrière-plan.

Nous avons aussi eu de la difficulté à savoir si on allait « cloner » les individus, mais nous avons trouvé comme problème que cela aurait pu représenter une grosse quantité de données à gérer. C’est pour cela que nous avons préféré utiliser l’agrégation et le partage de pointeur ainsi que d’ajouter de nouveaux attributs aux individus.

Il a aussi été difficile de trouver exactement les individus choisis, parce que certains individus avaient les mêmes caractéristiques. Nous avons dû compromettre sur ce point-là et laisser l’agent choisir au hasard parmi les individus similaires. Cela fait du sens, car même le joueur pouvait ne pas deviner l’individu précisément si les suspects avaient tous les mêmes caractéristiques. Cela est dû au fait que le joueur avait l’option de répondre par « oui pour un », ce qui laissait quelquefois le choix à la chance pour le joueur.

Un autre problème fut de gérer tous les cas créés par la pose de question. En effet, pour trouver certains individus, l’agent devait poser jusqu’à 18 questions afin d’arriver à sa réponse et donc, il était nécessaire de créer des cas pour toutes les réponses à ces 18 questions. Ce problème fût réglé avec plusieurs switch case, une solution longue, mais efficace.

Bien évidemment, nous avons rencontré quelques problèmes avec l’implémentation de l’algorithme de chemin le plus court, mais c’était normal, ce problème étant une partie de l’énoncé. Nous avons utilisé l’algorithme de Dijkstra, car cela nous semblait être un algorithme efficace pour parcourir le réseau social et découvrir le chemin le plus court entre les individus. Réussir à changer le pseudo-code de cet algorithme que nous avons vu en vrai code.

Finalement, un autre problème mineur fut le fait que nous étions plusieurs à travailler sur un même projet contenant peu de fichiers. Cela a créé plusieurs problèmes « git diff », mais l’utilisation de git a été très bénéfique pour la réduction de ce genre d’erreurs qui aurait bien plus fréquent sans git.

**Conclusion**

En conclusion, en créant un jeu « Dont vous êtes le héros » qui utilise des automates ainsi que la théorie du langage pour créer des mots de passe nous a permis de se familiariser avec le principe d’utilisation de graphe pour la validation de mots de passe ainsi que les règles de création de langage.

Au total, nous avons mis environ 24 heures pour réaliser ce travail. Nous avons mis 2-3 heures pour créer la première fonction C1 (ouvrirPorte()) et C2 (afficherReseauSocial()), 5-6 heures pour C3 (genererAutomate()), 2-3 heures pour C4 (afficherLeCheminParcouru()), 1-2 heures pour C5 (menu), 3 heures pour la rédaction du rapport et 2 heures pour la révision du code et du rapport, ce qui fait que nous avons mis plus de temps pour ce travail que pour nos examens.