Rapport de Projet IPF LDDIM2

Raphael LEONARDI, Baptiste PRAS

Section 1 : Fonctionnement du travail en groupe

- En Novembre / Décembre : (travail en présentiel) temps de travail presque exclusivement sur les temps de TP, à deux sur le même ordinateur pour se familiariser avec la structure et la construire (rien d'autre ne pouvait être fait tant que la structure n'était pas terminée). Fin décembre : constructeur de base et affichage simple terminés.
- Première semaine de janvier : (par messages uniquement) Raphael a terminé le constructeur pour qu'il vérifie que les murs intérieurs étaient bien placés (toutes les autres vérifications avaient déjà été faites). Il a également fait la génération random d'un labyrinthe.
- Deuxième et troisième semaines de janvier : (travail en présentiel), Raphael a travaillé sur l'affichage amélioré pendant que Baptiste a travaillé sur le solveur et les tests des 2 fonctions majeures qu'il fallait tester. Travail partagé à deux sur l'exécution à paramètres (random, solve, print, --help). Rédaction du rapport à deux une fois tout le code terminé et mise en page + commentaires du code par Baptiste.

Section 2 : Présentation générale du projet

- Pour la structure de donnée représentant le labyrinthe, nous avons opté pour le type labyrinthe suivant :

```
type labyrinthe = {
  largeur : int;
  hauteur : int;
  murs_largeur : int;
  murs_hauteur : int;
  murs : int array array;
  depart : int*int;
  arrivee : int*int
}
```

- Ce type contient 7 attributs :
    - largeur : un entier qui contient la largeur du fichier en entrée.
    - hauteur : un entier qui contient la hauteur du fichier en entrée.
    - murs_largeur : un entier qui correspond à 2*largeur+1, car une case contient 2*2 caractères, et on rajoute une colonne à la fin pour fermer le labyrinthe. C'est le nombre de caractères dans la largeur de la grille.
    - murs_hauteur : un entier qui correspond à 2*hauteur+1, car une case contient 2*2 caractères, et on rajoute une ligne à la fin pour fermer le labyrinthe. C'est le nombre de caractères dans la largeur de la grille.
    - murs : un array 2D de int contenant chaque caractère du labyrinthe (donc (murs_hauteur * murs_largeur). Les valeurs possibles sont 0 (mur), 1 (case vide), 2 (entrée), 3 (sortie), 4 (chemin résolu).
    - depart : une paire d'entiers définissant les coordonnées de l'entrée.
    - arrivee : une paire d'entiers définissant les coordonnées de la sortie.
- Les fonctions majeures sont :
    - constructeur file_name : construit un labyrinthe en lisant un fichier donné en paramètre. Vérifie qu'il est valable avant de la construire et de le renvoyer à l'utilisateur.
    - genereLabyrinthe hauteur largeur : génère un labyrinthe de manière aléatoire de taille (hauteur * largeur) (où hauteur et largeur sont le nombre de cases).
    - resolution laby : trouve le chemin entre l'entrée et la sortie dans le labyrinthe donné en entrée.
    - afficheLabyrinthe laby : affichage amélioré du labyrinthe.
    - read_parameters : lis les paramètres d'exécution du programme pour exécuter la bonne fonctionnalité du code (print, solve, random, --help).
- Les autres fonctions sont des sous-fonctions ou fonctions utilitaires réutilisées dans ces 5 grandes fonctions qui définissent le projet dans sa globalité.

Section 3 : Difficulté particulière rencontrée

Difficulté rencontrée avec le solveur :

Contexte :
A la base, cette fonction nous paraissait assez facile à faire car elle s'apparente à un basique problème de graphe, et Baptiste avait déjà travaillé sur les graphes de nombreuses fois dans d'autres langages de programmation (spécialement python). Il a donc été décidé de se voir en présentiel pour travailler parallèlement, Baptiste sur le solveur, Raphael sur l'affichage amélioré. Ainsi, il était facile de s'entraider ainsi que d'avancer vite en ayant chacun notre partie à faire, dont l'une se trouvait dans maze.ml et l'autre dans main.ml, ne créant aucun conflit de merge sur le dépôt git.

Mais après un premier jet pour la fonction solveur, cette dernière s'est avérée bien plus complexe qu'attendu, notamment car le paradigme fonctionnel, sans l'utilisation de boucles et avec une utilisation plus restreinte des listes et passages par valeur / variable qu'habituellement, notamment qu'en python.

Ainsi, il a d'abord fallu comprendre pourquoi le programme ne marchait pas, et ensuite régler le problème, ce qui a pris de longues heures et de nombreux changements, voire même réécritures de la fonction.

Quelles erreurs se sont glissées dans le code :
Parmi les erreurs, la plus importante, et celle qui a pris le plus de temps, a été de réussir à faire remonter le bon chemin dans la fonction principale. L'idée à la base était d'avoir une fonction solve, et une sous-fonction récursive solveRec qui permettrait de calculer le chemin et de le renvoyer dans solve. Malheureusement, le chemin remonté dans solve était constamment vide, il a donc fallu comprendre pourquoi, et comment le remonter.

Deux autres erreurs étaient assez bêtes et surtout dues à une faute d'inattention. La première, qui a posé problème assez longtemps, a été d'utiliser la hauteur et la largeur en nombre de cases, plutôt qu'en nombre de caractères, ce qui posait problème pour vérifier qu'un caractère était bien dans le labyrinthe (hauteur, largeur étant plus petits que murs_hauteur, murs_largeur). La seconde était l'oubli de la condition qui permettait de vérifier que le passage d'une case (x, y) à (x', y') ne passait pas à travers un mur, résultant dans un chemin qui se prenait pour Spiderman et traversait les murs.

Comment a-t-on corrigé l'erreur :

La première étape a été de comprendre pourquoi, malgré le fait que la fonction renvoyait bien le chemin à un moment donné, ce dernier était constamment vide.

Après avoir épluché le code, qui était à ce moment une succession de if else qui permettait de commencer un appel récursif au nord, puis au sud, puis à l'ouest, puis à l'est, nous avons compris que s'il était possible de poursuivre le chemin vers le nord, alors lançait un appel récursif vers le nord et si la résolution arrivait dans un cul de sac, à cause des if else, elle n'allait pas explorer les autres directions.

Le problème était donc maintenant de trouver comment faire 4 appels récursifs pour chacune des directions et d'exécuter si nécessaire chacun de ces 4 appels, pour n'en garder que le bon résultat (et ce sans casser son ordinateur, car un ordinateur cassé est peu efficace pour programmer).

Piste 1 :

La première solution explorée était de faire des try with, où chaque try contiendrait un appel vers une direction, et si cet appel venait à ne pas trouver de chemin, il enverrait alors une exception qui serait récupéré dans le with, qui pourrait passer à la direction suivante. Cette solution, satisfaisante du point de vue du paradigme fonctionnel, s'est avéré compliqué à mettre en place à cause de notre trop faible compréhension et appréciation du système try with, qui, une fois implémentée, permettait bel et bien d'explorer toutes les directions cette fois-ci, mais ne remontait pas le chemin dans la fonction principale.

Piste 2 :

La seconde solution, qui est restée la solution finale, a été de se rapprocher légèrement de ce que l'on connaissait bien avec la programmation impérative. Désormais, on enregistre chaque appel récursif dans une variable sous la forme d'une paire (bool, (int*int list)) où bool vaut true si le chemin a été trouvé et false sinon, et (int*int) list est simplement la liste de chaque coordonnée du chemin.

Après avoir exécuté et stocké ces 4 appels, on vérifiait si l'un d'entre eux avait rendu true, si oui, on renvoyait son chemin, sinon, on renvoyait qu'il n'existait aucun chemin. Ainsi, on vérifie chaque direction et le programme remontait enfin le chemin nécessaire à l'affichage du labyrinthe (changement des 1 en 4 dans la structure labyrinthe.murs). Par la suite, il a été testé si un appel avait trouvé le chemin avant de lancer les autres appels, pour gagner en temps d'exécution (éviter des appels inutiles car le chemin est déjà trouvé).

Section 4 : Description des tests effectués

Au départ du projet, à chaque sous-fonction créé, notamment pour la lecture et la vérification du constructeur, la fonction était testée dans le terminal avec un simple appel à cette fonction et la vérification manuelle, dans tous les cas possibles, de la réponse de cette fonction et ainsi de son bon fonctionnement, avec l'aide de fonctions d'affichage utilitaires créées à cet effet, et toujours disponible dans main.ml.

Par la suite, nous avons créé dans test.ml des tests plus lourds qui ont permis de tester les fonctions majeures (décrites dans la section 2 de ce projet), et par la même occasion donc les sous-fonctions testées manuellement, qui ont, grâce à différents tests à vérification automatique (grâce à une première vérification manuelle sur une petite grille 3x2) étaient vérifiées en profondeur dans toutes les grandes familles de cas possibles.

Ainsi le constructeur d'un labyrinthe est testé dans 4 cas différents :

- Un premier cas, où l'on vérifie que la structure renvoyée est bien la bonne pour tous les attributs, notamment la grille.
- Un deuxième cas où l'on vérifie que l'on renvoie bien une erreur si le fichier passé en lecture est invalide à cause d'un caractère inconnu.
- Un troisième cas où l'on vérifie que l'on renvoie bien une erreur si le fichier passé en lecture est invalide à cause d'un mur mal placé.
- Un quatrième cas où l'on vérifie que l'on renvoie bien une erreur si le fichier passé en lecture est invalide à cause de la présence de plusieurs départs.

Ensuite, le solveur a été testé en vérifiant, toujours sur le labyrinthe 3x2, que la grille renvoyée, avec les 4 inclus représentant les caractères du chemin, était correcte. Cette fonction a aussi été testée manuellement maintes et maintes fois avec des labyrinthes aléatoires de toutes tailles (jusqu'à 100x100).

Ces 5 tests automatiques sont visibles dans test.ml, inclus dans le répertoire git, et sont exécutable en changeant le mot main par test dans le dune build. Nous n'avons pas réussi à faire un fichier dune qui permettait de faire en même temps le main et le test, car visiblement le fait d'inclure maze.ml dans les deux en même temps dérangeait le compilateur.

Section 5 : Affichage amélioré

Nous avons décidé de choisir l'affichage du labyrinthe avec des caractères ASCII, et des caractères d'échappement. Raphael a fait l'affichage amélioré des murs et du chemin tel qu'il est dans sa forme finale.

L'affichage des murs se décompose en 3 parties distinctes : d'abord l'affichage des murs de la ligne du haut, puis des murs intérieurs et enfin ceux de la ligne du bas.

L'affichage se fait grâce à la banque de caractères ASCII trouvée sur Wikipedia. Voici pour les murs choisis :
‖ ╔ ╗ ╚ ╝ ╠ ╣ ╦ ╩ ╬ ═

Pour les murs extérieurs du labyrinthe, on regarde les cases autour de ce dernier afin de déterminer la forme du mur, car ce dernier doit être connecté à un potentiel mur présent à sa droite, gauche, en dessous ou au-dessus.

C'est pour cela que l'on doit distinguer plusieurs cas :

- Mur du haut : on se contente de regarder la case du dessous. Si cette case est un mur on affiche le caractère ╦, sinon on affiche ═.
- Mur du bas : on se contente de regarder la case du dessus. Si cette case est un mur on affiche le caractère ╩, sinon on affiche ═.
- Mur de gauche : on se contente de regarder la case de droite. Si cette case est un mur on affiche le caractère ╠, sinon on affiche ‖.
- Mur de droite : on se contente de regarder la case de gauche. Si cette case est un mur on affiche le caractère ╣, sinon on affiche ‖.

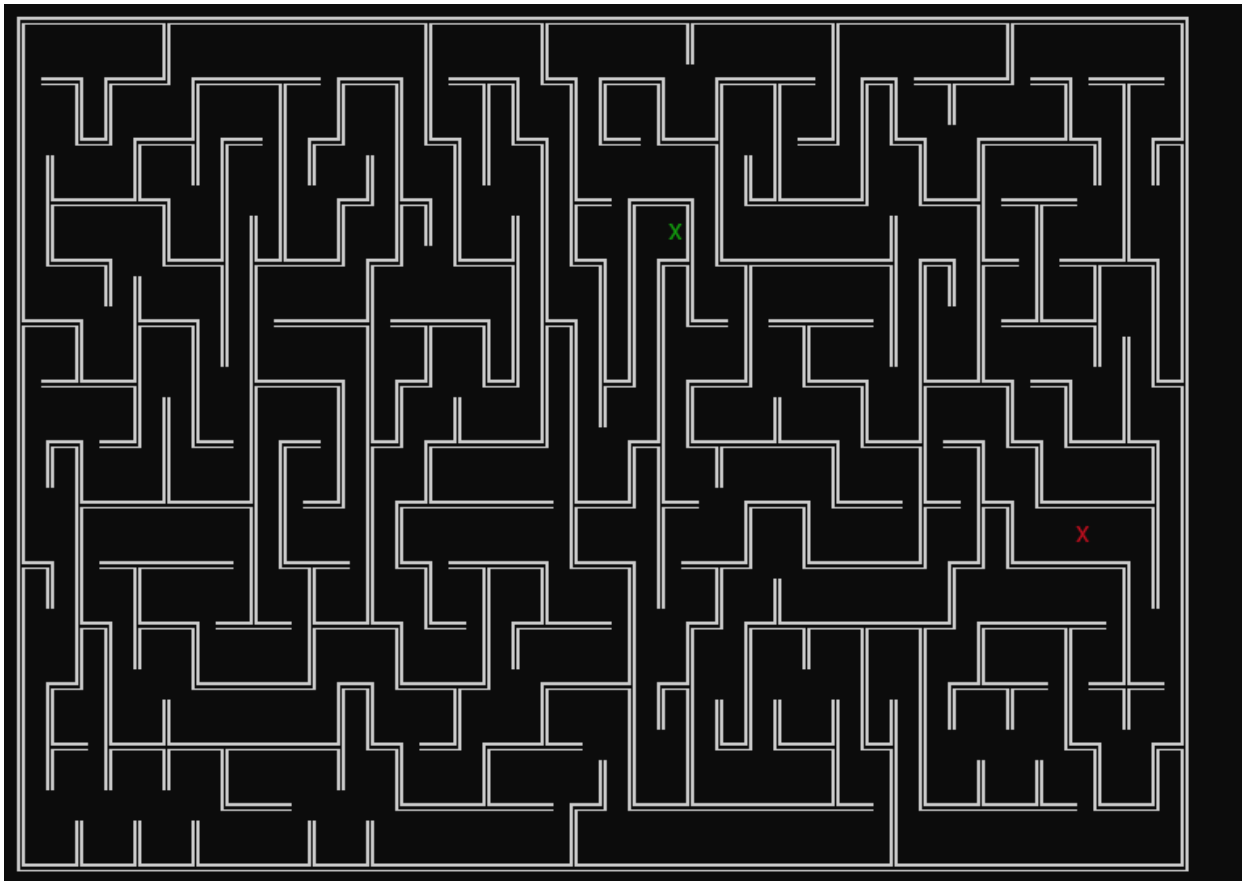
Entre la ligne du haut et celle du bas, il faut faire l'affichage des lignes intermédiaires. Lorsque l'on est du début d'une ligne, on fait appel à la fonction prettyGauche, qui regarde si la case à droite est un mur, et agit en conséquence. Si la case est à la fin de la ligne, on utilise la fonction prettyDroite qui regarde si la case à gauche est un mur, et agit en conséquence.

Lorsque l'on ne se trouve pas sur les murs extérieurs, on regarde la valeur du int à l'intérieur de l'array 2D d'entiers qui définit la grille. Si c'est une case vide, on se contente d'afficher un espace.

Si c'est un mur, on doit alors regarder les valeurs des cases adjacentes. On stocke quatre booléens pour chacune des cases ajdacentes, qui vaut true si la case est un mur et false sinon. On effectue un match sur ces booléens et on affiche un caractère selon ses valeurs, afin de connecter le mur de la case courante aux potentiels murs des cases adjacentes et de ne pas avoir d'artefact dans l'affichage.

Enfin si c'est le départ, on affiche un X en vert, et si c'est l'arrivée, on affiche un X en rouge.

Voilà un exemple, avec un labyrinthe généré aléatoirement de taille (14*40) :



Lorsque l'on utilise solve, l'affichage des chemins fonctionne de manière analogue à l'affichage, simplement avec des caractères différents, que l'on affichera qui plus est en jaune.
On utilise la fonction prettyChemin qui regarde les cases autour de la case courante. On fait un match sur les booléens qui valent true si la case est un chemin, l'arrivée ou le départ. Il y a toujours exactement 2 cases adjacentes qui sont un chemin. Cela permet, comme pour murs, de lier la case courante au reste du chemin et d'éviter les artefacts visuels.

Avec l’affichage amélioré, notre labyrinthe (14*40) résolu devient :