Minesweeper

Projet de programmation fonctionnelle

Calandra Joséphine
Parpaite Thibault
Poncet Clémence
Sarrabayrouse Alexis

Encadrant : M. Hett Kilian

ENSEIRB-MATMECA
Filière informatique
Semestre 6
12 mai 2017

Table des matières

1 Introduction

Contexte

Ce document a pour but de décrire le déroulement de notre projet de programmation fonctionnelle réalisé en Scheme au cours de notre semestre 6 de cursus informatique à l'ENSEIRB-MATMECA. L'objectif de ce dernier est d'implémenter un jeu de réflexion, **le démineur**, dont la version la plus connue est celle distribuée par Microsoft sur le système Windows. Le deuxième point crucial est la mise en place d'une intelligence artificielle capable de jouer au jeu en adoptant différents types de stratégies.

Plan

Dans un premier temps, nous étudierons le **domaine** du projet en explicitant de manière formelle les règles du jeu tout en réalisant une analyse approfondie de l'existant.

Ensuite nous nous intéresserons à la phase de **conception** en rédigeant le cahier des charges décrivant de manière détaillée les besoins et les limites du projet. Cette partie détaillera également l'architecture du code.

Après cela, nous présenterons plus en détails le **logiciel** ainsi que les choix d'**implémentation** pris tout au long du développement.

Enfin nous allons mettre en évidence la véracité et performance de notre travail à l'aide de l'ensemble des jeux de **tests** mis en place.

Méthodes de travail

Nous avons mis en place l'outil CMake pour permettre une compilation dynamique, portable et simplifiée de notre projet.

Est aussi jointe à ce dernier, une documentation au format HTML Doxygen de l'ensemble des fonctions, macros et autres structures de données implémentées tout au long du développement.

Pour la gestion des versions, nous avons utilisé un dépôt *SVN* sur la forge de l'ENSEIRB-MATMECA. Les éditeurs de texte que nous avons privilégiés sont *Emacs* et *Atom*.

Pour suivre l'avancée du projet de manière ergonomique, nous avons mis en place un tableau Trello visible sur la figure **??**. Cela a permis d'évaluer de manière efficace la charge de travail, répartir les tâches et tenir les délais. Le rapport a été rédigé à l'aide de l'éditeur en ligne collaboratif ShareLatex.

Enfin, nous avons choisi de réaliser ce projet en suivant la méthodologie *SCRUM*. En réusmé, *SCRUM* est un processus agile qui procède par itération successives en se focalisant dans un premier temps sur les fonctionnalités essentielles. L'objectif est qu'à la fin de chaque phase de *sprint* (environ deux semaines) un produit **fonctionnel** soit livré. Le livrable implémente seulement une partie des besoins définis dans la cahier des charges (choisis par ordre de priorité), les autres fonctionnalités étant traitées durant des phases de sprint ultérieures.

2 Démineur - Règles du Jeu

Le jeu du démineur suit le principe suivant : le joueur dispose d'une grille (de taille variable) sous les cases de laquelle se trouve un certain nombre de mines (là encore, le nombre peut varier) dont le joueur ne connaît pas l'emplacement.
Son but est donc de déterminer l'emplacement de toutes ces mines sans exception, et de "découvrir" toutes les cases restantes. Pour l'aider, chaque case découverte permet d'obtenir des informations sur les mines du voisinage.
En effet, sur chaque case découverte se trouve un nombre, qui correspond au nombre de mines présentes dans les cases adjacentes à celle-ci. De cette manière, il est possible de déterminer l'emplacement de certaines mines, ou de découvrir sans risque une case, car on sait qu'elle ne peut pas abriter de mine.
Si le joueur se trompe, et choisit de découvrir une case qui s'avère minée, il a perdu.

3 Analyse de l'existant

Il existe déjà plusieurs implémentations bien connues du jeu de démineur, dont la plus connue est la version de Microsoft Windows.
Parmi elles, de nombreuses interfaces graphiques, mais peu de différences de fonctionnement, puisqu'elles restent toutes fidèles aux règles décrites précédemment.
En effet, toutes les versions suivent le même principe : le joueur doit cliquer sur une case pour la découvrir, et le nombre de voisins minés de celle-ci apparaît alors -si elle n'est pas elle-même minée -. Si elle ne possède aucun voisin miné, une révélation récursive des cases voisines se met alors en place, jusqu'à ce que les extrémités de la zone découverte possèdent chacune au moins un voisin miné. Enfin, si la case abritait une mine, alors celle-ci "explose", ainsi que toutes les autres présentes sur le plateau de jeu, et toutes les cases sont révélées : le joueur a perdu. Il existe peu de différences fondamentales entre les diverses implémentations de ce jeu, si ce n'est peut-être que toutes les versions ne proposent pas de drapeaux à placer sur les cases, ou encore que certaines proposent en outre un système de drapeau pour indiquer l'incertitude quant au statut de la case concernée.
En effet, il n'est pas rare qu'un joueur se trouve à court de coups sûrs, et qu'il se voit obligé d'émettre des hypothèses quant aux statuts des cases encore recouvertes, et donc à se réduire à des coups non déterministes, ou même à cliquer au hasard. Cette situation étant coutumière pour le jeu du démineur, elle représente une part importante de notre implémentation.
Quant aux *flags* présents dans la plupart des implémentations, ils semblent une bonne manière de faciliter la détermination des coups sûrs.

4 Cahier des charges

4.1 Besoins fonctionnels

Notre application doit être adaptable à différents types de pavages : pavages uniformes (rectangles, triangles...) dans un premier temps, puis pavages archimédiens plus globalement. Afin de faciliter les débugs de notre programme, nous avons visualisé nos grilles de démineur par des graphes grâce au langage graphviz.
Dans un second temps, on cherchera également à modéliser des parties automatiques, afin de procéder à des calculs statistiques, si le temps nous le permet.

4.2 Besoins non fonctionnels

Le jeu doit être pertinent : il doit pouvoir s'arrêter lorsque l'on tombe sur une bombe.
Le jeu doit être facile de compréhension : En effet, pour permettre une application du jeu agréable, nous avons choisi dans un second temps d'implémenter une interface visuelle compréhensible pour le joueur.

4.3 Contraintes et limites

Il s'agit ici d'un projet de programmation fonctionnelle sur du racket non typé.
On ne peut pas utiliser d'effets de bords en fonctionnel, il est donc nécessaire de s'adapter en renvoyant un nouveau plateau à chaque appel de fonction.

5 Architecture

Notre projet s'organise autour de plusieurs fichiers : board.scm, contenant nos structures et implémentant notre plateau, les fichiers de stratégies strategy.scm et strategy_limited_search.scm - qui correspondent respectivement aux coups sûrs et aux coups non déterministes - ainsi que l'implémentation de l'interface graphique gui.scm .

Nous avons par ailleurs choisi de représenter notre grille de démineur par un graphe. Ce graphe est représenté par une liste d'adjacence, contenant pour chaque élément la liste des voisins à ce sommet.

6 Logiciel

Pour faciliter l'implémentation des structures permettant de représenter notre grille de jeu, ainsi que les fonctions la manipulant, nous avons exporté notre graphe au format Graphviz [1].

L'exportation se fait par le biais de la fonction *board-to-gv* qui se charge également de générer le png et de l'ouvrir dans une fenêtre.

Le lancement des tests unitaires consiste à exécuter le fichier tests.scm.

1. Graphviz est un logiciel de visualisation de graphe `http://www.graphviz.org/`

7 Algorithmes et structures de données

Après avoir pris connaissance du projet, il nous a fallu considérer l'implémentation que nous souhaitions pour les structure représentant le plateau de jeu - *Board* - et les cases de celui-ci - *Cell* -.
Dans le premier cas, nous nous sommes basés sur les informations données dans le graphe en entrée permettant la création d'un plateau de jeu. Aussi, la structure *Board* contient des informations sur la taille de celui-ci, ainsi qu'un tableau *tab* de *Cell*, mais également le nombre de mines du plateau et le nombre de cases encore recouvertes, dans le but de savoir en temps constant si la partie a été gagnée, grâce à la fonction *game-won*.
Dans le cas de la strcuture *Cell*, les informations présentes dans le graphe en entrée concernant les voisins respectifs de chaque case y sont stockées sous la forme d'une liste d'entier (*list-neighbors*), ce qui permet d'en faire aisément varier la taille, et donc d'ajouter des voisins en temps constant. Chaque *Cell* est représentée par un entier *id* qui correspond à sa place dans le tableau *tab* du *Board*. Enfin, chaque *Cell* est marquée comme étant ou non découverte, minée ou possédant un drapeau indiquant si l'utilisateur la considère minée ou non, et à chacune est associée le nombre de mines de ses voisins, qui dévient accessible au joueur dès lors que la case est découverte.
Tous ces choix d'implémentation ont été faits dans l'esprit de faciliter l'accès à ces informations au regard des fonctions que nous avons implémentées, dans le but de réduire leur complexité.

8 Stratégies

8.1 Les coups sûrs

Un des aspects importants de ce projet était d'implémenter un algorithme déterminant l'ensemble des coups sûrs, c'est-à-dire l'ensemble des cases dont l'on est certain qu'elles ne sont pas minées.
Dans cette optique, on a choisi de réfléchir à ce problème en adoptant un point de vue différent. En effet, nous avons décidé de déterminer tout d'abord les cases qui étaient minées, afin d'en déduire ensuite les cases qui ne l'étaient pas.
Ce choix s'explique par le côté plus intuitif de la détermination des cases minées, puisque les jeux de démineurs existant proposent pour la plupart un système de drapeaux qui permet de repérer les cases qui sont, à notre avis, minées.
De la même manière, nous avons donc mis en place un système de *flag*, mais en élargissant cette notion, puisque notre implémentation permet non seulement de marquer une case comme minée, mais également de les marquer comme "coup sûr".
Aussi, il s'agit de mettre en place une détermination des cases minées sur le plateau de jeu. Pour cela, on choisit de parcourir celui-ci dans son ensemble - à l'exception des cases déjà découvertes - et de considérer chaque case à l'aide de l'ensemble de ses voisines, découvertes ou non.
Concrètement, il s'agit notamment de repérer toutes les cases découvertes dont le nombre de cases adjacentes non découvertes est égal au nombre de ses mines voisines. On en déduit naturellement que tous les voisins de cette case sont minés, et on les marque donc comme tels.
Un autre cas de figure consiste à repérer les cases c_1 ayant seulement 1 mine voisine, et de s'intéresser aux voisins c_next de celles-ci. En effet, si on note *n_mines_vois(c)* le nombre de voisins minés d'une case c, s'il existe une c_next qui a *n_mines_vois(c_l) + 1* voisins non découverts, alors toute carte voisine de c_next qui n'est pas voisine de c_l est forcément minée, et donc marquée comme telle.
A la fin de ce premier parcours, on a donc marqué un certain nombre de case marquées comme minées, et l'on peut donc se concentrer sur le réel objectif de l'algorithme, qui est de déterminer des cases **non minées**. Par manque de temps, nous n'avons malheureusement pas eu le temps de terminer l'implémentation de cet algorithme.

8.2 Les coups non déterministes

Afin d'effectuer une action même s'il n'existe aucun coup sûr, nous avons choisi d'implémenter un algorithme suivant le paradigme min/max (heuristique probabiliste).

On définit tout d'abord la *zone of interest* d'une cellule inconnue x comme l'ensemble des cellules déjà découvertes adjacentes à x (ensemble 1) auxquelles on ajoute l'ensemble des cellules découvertes partageant une cellule inconnue avec l'ensemble 1. Cette zone permet de limiter l'étendue des recherches que l'on fera lors de la phase de backtracking.

FIGURE 8.1 – *Zone of interest* de la cellule x

Une version simplifiée de l'algorithme [1] utilisé est décrite ci-dessous. L'idée est de supposer qu'une cellule est minée (ou non) sur plusieurs niveaux et de voir si cela mène à des incohérences. Lorsqu'un coup sûr n'est pas possible on établit une *safe_list* qui fait la moyenne des cas où la supposition a mené à une finalité sans contradiction.

1. Inspiré de l'article suivant :
`www.minesweeper.info/articles/ComplexityMinesweeperStrategiesForGamePlaying.pdf`

Algorithm 1: Algorithme de stratégie (avec probabilité) : strat_prob(b)

```
b : board
safe_list ← {}
while not lost do
    for Chaque cellule x non découverte de board do
        m ←search(b, O, #t)
        if m == 0 then
            Dévoiler x
        else
            f ← search(b, O, #f)
            if f == 0 then
                Marquer x comme miné
            else
                add_safe_list(safe_list, m/(m + f))
            end
        end
    end
    Jouer le coup le plus sur dans la safe_list
end
```

Algorithm 2: Algorithme de backtracking : search(b, O, bool)

```
1  b : board, O : l'ensemble des cellules à traiter
2  bool : indiquant si on doit supposer qu'il y a une mine sur la première cellule de O
3  if O est vide then
4  |  retourner 0
5  else
6  |  p ← O_1
7  |  O ← O − {p}
8  |  if bool == 0 then
9  |  |  b ← On suppose qu'il y a une mine à la position p dans b
10 |  else
11 |  |  b ← On suppose qu'il y n'y a pas de mine à la position p dans b
12 |  end
13 |  if Une contradiction est apparue suite à la supposition précédent then
14 |  |  retourner 1
15 |  else
16 |  |  retourner search(b, O, #t) + search(b, O, #f)
17 |  end
18 end
```

9 Tests

9.1 Tests unitaires

Dans un premier temps, il a fallu effectuer une série de tests unitaires afin de vérifier le bon fonctionnement des fonctions que nous avions implémentées dans *board.scm*. Dans cette optique, nous avons défini plusieurs *board* à l'aide de fichiers décrivant un plateau carré de 4 cases, un plateau "en escalier" de 8 cases, ainsi qu'un plateau vide.
Pour commencer, on a testé la bonne implémentation de *board-create* en vérifiant que les divers plateaux créés par la fonction étaient de la taille spécifiée, et que chaque case de ceux-ci aient les voisins désirés, et pas d'autres.
Pour ce qui est du test de *cell-set-mine*, il s'est agi de vérifier en premier lieu que les plateaux considérés n'étaient pas minés, puis d'ajouter une ou plusieurs mines sur différentes cases à l'aide de la fonction, avant de vérifier que ces cases, et seulement elles, avaient été minées. Enfin, on vérifie que le nombre de mines voisines de chaque case ait été correctement mis à jour.
Dans le même esprit, on vérifie *cell-set-uncovered* découvre bien les cases demandées -et seulement elles. Le fonctionnement de *cell-set-flag* est testé de manière similaire.
Afin d'établir le bon fonctionnement de *cell-add-neighbor*, on ajoute un voisin à une case qui en possède déjà n, et on accède ensuite au n+1ème élément de sa liste de voisins afin de le comparer avec celui qu'on a souhaité ajouter.
S'agissant de la fonction *board-get-n-mines-neighbor* qui, étant donné un plateau et un entier, est censée renvoyer le nombre de voisins minés de la cellule identifiée par l'entier donné - et ce seulement si la cellule est découverte, sinon elle renvoie -1 - le test consiste à vérifier pour chaque case que la valeur retournée est celle qui correspond à la réalité du plateau.
Dans le cas de *board-generate-mines*, puisqu'il s'agit d'un placement aléatoires de mines, on se sert d'un affichage avant et après l'appel de la fonction afin de vérifier que celle-ci génère le bon nombre de mines sur le plateau considéré, et met correctement à jour le nombre de voisins minés de chaque case de celui-ci.
Certaines fonctions auxiliaires n'auront pas été testées, car les tests réussis des fonctions dans lesquelles elles interviennent suffisent à justifier que notre implémentation est correcte.

10 Conclusion

Nous sommes parvenus à implémenter toutes les fonctionnalités nécessaires au jeu du démineur, mais n'avons malheureusement pas réussi à lancer une partie. Grâce à nos tests, nous avons pu voir que toutes les fonctions permettaient bel et bien de réaliser une partie de démineur, seule la boucle de jeu principale n'aura pas été implémentée : seul une bonne gestion du temps nous aura donc fait défaut pour parvenir à réaliser cet objectif.
En outre, une ébauche d'interface graphique est disponible dans le fichier *gui.scm*. Nous avons réussi à afficher un matrice et à récupérer les coordonnées des cliques de la souris mais sans parvenir à récupérer l'objet sur le quel nous cliquions.

Une solutions envisagée aurait été de faire correspondre ces coordonnées à une case de la grille mais cette solutions parait peu viable. Une autre piste trouvé malheureusement trop tard est la bibliothèque *htdp/gui* qui apparement fait correspondre les cliques directement à des objets et non uniquement à des coordonnées sur la fenêtre.

Dans l'ensemble, nous avons atteint un certain nombre des objectifs proposés, et n'avons cependant pas atteint le cinquième, n'ayant pas été en mesure de terminer l'implémentation de l'algorithme concernant les coups sûrs, pas plus que de réaliser des statistiques sur plusieurs parties. Cependant, nous avons beaucoup réfléchi sur les différents algorithmes que nous ne sommes pas en mesure de délivrer fonctionnels, et serions donc capables de les implémenter.

Ce projet nous aura permis de manipuler un langage de programmation, le Scheme, et de nous l'approprier un peu plus, donc de le maîtriser un peu plus.
Le fait de programmer en langage fonctionnel impliquait d'adopter un nouveau point de vue, notamment concernant l'absence d'effet de bord qui caractérise le fonctionnel. En effet, il nous a fallu penser différemment que lors de la programmation en C, ou en python notamment, et nous nous sommes peu à peu faits à cette nouvelle manière de réfléchir, et de programmer.