Rapport Final projet EDD

Pinero, Borde, Bonnet

20 avril 2015

Table des matières

1 Le projet

1.1 Présentation

Le projet ce compose de plusieurs parties de développement sur un sujet commun, le jeu du 2048. Ce jeu conciste à additionner des tuiles de même puissance de 2, sur un plateau de 16 tuiles, jusqu'à ce que le plateau soit plien et qu'aucune addition ne soit possible. Le but du jeu et d'arriver á construire une brique de valeur la plus grande possible (2048, 4096, 8192, ...)

POURQUOI PAS AJOUTER UNE PHOTO EXEMPLE ?!

1.2 Le sujet

Dans le projet nous avons eu à réaliser dans un premier temps l'ensemble du moteur du jeu et quelques tests qui nous garantisent un maximum la véracité pendant le développement. Dans un deuxième temps, nous devions faire la « critique » du developpement de la première partie faite par d'autres groupes basé sur une grille d'evaluation qui nous a été fourni. Ensuite nous avons réaliser un ensemble de stratégie de jeu qui sont capable de jouer de manière autonome une partie entière, jusqu'á la fin de la partie. Ces stratégies doivent à chaque tours calculer le meilleur prochain coups pour atteindre le plus grand score possible. La dernière partie du projet conciste à réalisé une véritable interface graphique pour jouer au jeu.

2 Développement du 2048 et tests

Pour le développement du 2048, le langage de programmation était imposé, nous devions utilisé le langage C. Chaque groupe ont dû fournir les sources permettant de générer une bibliothèque libgrid.a (implémentant les fonctions d'écrites dans grid.h en annexe **??**) ainsi qu'un exécutable permettant de jouer au 2048 dans la console. En plus de ça, nous avons réalisé des tests pour vérifier toutes les fonctions implémentés.

Avant de commencer le travail de développement, nous avons utilisé un outil de gestion de versions décentralisé pour que chaque membre du groupe puisse avoir accés à chaque instant à la dernière version du projet. Pour ça nous avons choisit le logiciel GIT, et la plateforme GitHub. Les sources du projet ont été disponible tout au long du développement ici.

En ce qui concerne l'arborescence de notre projet, nous avons décidé d'avoir une organisation bien précise pour quelle puisse être le plus possible générique, c'est á dire quelle s'adapte aux évolutions du projet. Mais surtout pour quelle soit le plus clair possible.

```
EDD_projet/        # Racine du projet
    bin/           # Fichiers exécutables.
    build/         # Fichiers de compilation du projet
    include/       # Fichiers .h des librairies générés
    lib/           # Libraires
    src/           # Fichiers tous les fichiers sources
        2048/      # Fichiers sources de l'interface graphique
        grid/      # Fichiers du moteur de jeu
            tests/ # Fichiers qui testent le moteur de jeu
```

Tout au long du développement du projet, tous les membres du groupe se sont mis d'accord sur le fait d'utiliser l'outils CMake pour compiler le projet, générer les Makefiles qui servent eux-mêmes à générer les exécutables, les libraires et exécuter les tests.

2.1 Le developpement de grid.c

En ce qui concerne le développement de grid.c, qui est le moteur du jeu, c'été relativement simple. Les fonctions qui ont mérité un réflexion sérieuse

sont « do_move() » et « cam_move() ». Pour ces fonctions nous devions limité au mieux la duplication de code et la compléxité de l'algorithme. Chose que nous n'avions au final pas très bien reussi, mais nous veront dans la pratie ??.

2.1.1 Implementation de « can_move »

Pour cette fonction, nous avons choisi l'algorithme suivant :

```
Si la direction donnée est «  UP » ou «  DOWN » alors
    Pour chaque colonne faire :
        Pour chaque tuile de la colonne faire :
            Si la tuille précédente est égale à la courante
                retourner vrai

            Si la tuille courante et non vide
                   et qu'il y a une tuille vide dans la colonne
                retourner vrai
Si la direction donnée est «  RIGHT » ou «  LEFT » alors
    Pour chaque ligne faire
        Pour chaque tuile de la ligne faire
            Si la tuille précédente est égale à la courante
                retourner vrai

            Si la tuille courante et non vide alors
                   et qu'il y a une tuille vide dans la ligne
                retourner vrai
retourner faux
```

Cette fonction est utilisée á plusieurs reprises dans d'autres fonction comme par exemple « game_over » ou « do_move ». d'où l'importance qu'elle n'ai pas une trop grande compèxité de calcul.

2.1.2 Implementation de « do_move »

Pour cette fonction, nous avons choisi l'algorithme suivant :

```
Si la direction donnée ne peut pas être jouer
    Ne rien faire


Si la direction donné est «  UP » ou «  DOWN » alors
    Pour chaque colonne faire :
        Pour chaque tuile de la colonne faire :
            Si la tuille précédente est égale à la courante
                On fusionne les deux tuilles
                On modifie le score de la partie

            Si la tuille courante est vide alors
             On change sa place avec la première tuille non vide

Si la direction donnée est «  RIGHT » ou «  LEFT » alors
    Pour chaque ligne faire
        Pour chaque tuile de la ligne faire :
            Si la tuille précédente est égale à la courante
                On fusionne les deux tuilles
                On modifie le score de la partie

            Si la tuille courante est vide alors
             On change sa place avec la première tuille non vide
```

2.2 L'afficage de la grille dans la console

Pour l'affichage de la grille dans la console, nous avons eu plusieurs version et design. La toute première version était une simple grille dans la console où était affiché le score et les valeurs des tuilles. Dans cette version nous devions jouer avec les lettres Z, Q, S et D du clavier car dans le language C, la fonction « scanf » de la librairie « stdio » qui permet de lire l'entrée standard, ne prend pas en compte les flèches du clavier. En annexe vous touverai une capture d'ecran de cette version.

Dans une deuximème version, nous avions décidé de mettre des couleurs dans les tuilles en fonction de leurs valeurs. (à voir en annexe **??**).

Enfin pour répondre correctement au sujet, nous avons utilisé la librairie

« NCurses ». L'utilisation de cette librairie nous a permis de se libérer de deux de nos difficultés. La première était celle que nous devions jouer avec des lettres et non les flèches du clavier, la seconde était que nous n'avions d'affichage dynamique de la grille à chaque coup. Pour être plus clair, lorsqu'on joué un coups avec les deux premières version de l'affichage, la grille du coup précedent était toujours présent dans la console. Désormais, la console n'affiche plus qu'une grille et elle met à jour après chaque coups.

2.3 Les tests

Les Tests sont écrit pour confronter une réalisation à sa spécification. Le test définit un critère d'arrêt (état ou sorties à l'issue de l'exécution) et permet de statuer sur le succès ou sur l'échec d'une vérification. Grâce à la spécification, on est en mesure de faire correspondre un état d'entrée donné à un résultat ou à une sortie. Le test permet de vérifier que la relation d'entrée / sortie donnée par la spécification est bel et bien réalisée. [?]

Sur ce principe nous avons réalisé un test par fonction décrite dans le fichier « grid.h » et un test sur la réalisation de plusieurs parties qui s'exécutent avec une stratégie basique qui consiste à jouer a gauche si on le peut sinon à droite, sinon en haut enfin en bas. Ainsi lorsqu'on fait des modifications de l'une de ces fonctions, nous nous assurons que cela n'a en rien changé à la véracité des résultats. L'eécution des tests s'éffectue avec la commande « make test ». L'utilisation de la technologie « CMake » nous permet d'avoir un affichage complet à la fin des tests sur leurs résultats pour savoir s'ils sont passés avec succès ou non (exemple en annexe **??**).

Comme dis précédement, chaque test est spécifique à une fonction.

— La création d'une grille et sont initialisation correcte.
— L'homogénéité de la fonction add_tile, c'est à dire que la probabilité que la tuile généré soit bien de $\frac{1}{le\ nombre\ de\ tuille}$ avec un marge de 0.05%.
— La véracité des fonctions « game_over » et « can_move »
— La bonne exécution des mouvements dans chaque direction.
— Enfin l'exécution sans erreur de 1000 parties.

3 Relectures critiques

À l'issue de la première phase de travail, chaque groupe ont dû relire le code produit par trois autres groupes. Pour chaque groupe à relire, on a rempli un formulaire d'éaluation. Ce formulaire d'éaluation nous a été fourni par notre chargé de TD.

Cette feuille d'évaluation était composée de 13 questions orientées sur trois axes. Le point de vue utilisateur, qui pousse à vérifier principalement la propeté de l'archive, que la compilation ne génère pas d'erreur et qu'il y a un fichier « README » pour expliquer l'utilisation du programme si ce n'est pas évident. Dans un deuxième temps, d'un point de vu fonctionnel, nous devions vérifier si le programme ne présente pas de bogue ou de fuite mémoire. Enfin, d'un point de vue programmation, nous devions porter notre attention sur les patrons de conception (ou *design patterns* en anglais).

3.1 Les relectures que nous avons réalisé

Nous avons eu à faire la relecture des groupes A, D et G. Comme nous sommes trois, nous nous sommes divisé le travail et avons évalué un groupe chacun. Nous nous sommes ensuite concerté pour harmoniser nos commentaires. Cette relecture fût très intéressente car elle nous a permis de mettre au premier plan les points importants sur lesquels il faut insister lors du développement d'un projet. En plus de cela, comme chaque groupe à eu le choix sur la façon d'implémenter son programme, nous avons vu les différentes possibilités d'implementation parfois meilleure et plus éfficace ou pas. Le but n'était pas d'être trop sévère ou au contraire laxiste sur nos commentaires car ceux que nous avons donné sur chaque projet n'ont pas été noté.

3.2 Les reslectures réalisé par les autres groupes

En ce qui concerne les commentaires qui ont été faits par les autres groupes sur notre projet, nous les avons trouvé justifié. Memê si, il faut l'avouer, recevoir des « critiques » sur le travail que nous avons réalisé pendant près de trois ou quatre semaines fait un peu grincer des dents. Nous avons pris ces critiques en compte et en avons corrigé un maximum avant le rendu final.

Nous avons principalement reçu des commentaires négatifs sur ma duplication de code dans l'implementation du moteur de jeu. Certaines fonctions

dans utilisé dans les fonctions « can_move » et « do_move » se rensemble fortement. Un autre point à été remarqué, le nomage des variables et les commentaires dans le code source étaient certaines fois en français et d'autre en anglais alors que les patrons de conceptions nous imposé de les mettre qu'en anglais.

4 Les stratégies

Dans cette partie, chacun des groupes ont du mettre au points au minimum deux stratégies. Une stratégie rapide (capable de jouer une partie en moins de 10s) et une stratégie plus gourmande (capable de jouer une partie en moins de 2min). Une stratégie est une structure qui implémente l'interface décrite dans strategy.h. Chaque stratégie sont de la forme d'une bibliothèque dynamique. Nous avons implementer quatre stratégies pour l'instant, deux plutot triviales qui sont basées sur la stratégie du coin, et deux autres sur la stratégies « expected max ».

4.1 Stratégie du coin

La stratégie fonctionnelle choisie pour le rapport est la stratégie du coin. Pour cela, nous avions fait un premier algorithme qui se contente de jouer vers la gauche tant qu'on le peut, sinon il joue vers le bas encore une fois tant qu'on le peut, de même vers la droite puis vers le haut. Avec cet stratégie, sur 48 parties nous avons un score moyen de 2475, avec 6x64, 20x128, 20x256 et 2x512.

Dans un seconds temps, nous avons fait un second algorithme qui est basé sur le premier mais qui un coup sur deux change les mouvements favoris. Les premiers mouvements favoris sont dans cet ordre, gauche, bas, droite et haut, et les seconds mouvements sont dans cet ordre, bas, gauche, droite et haut. Avec cette stratégie, sur 48 parties nous avons un score moyen de 2259, avec 1x32, 3x64, 21x128 et 23x256.

Les deux stratégies se valent à peu près, mais c'est bien la première qui est la meilleure.

4.2 Implémentation d'expected max

« Expected Max » est un algorithme qui consiste à retourner la direction optimale à jouer. Elle est basée sur l'algorithme « minimax ». Le principe est de donner une valeur à la grille en fonction de plusieurs critères (cf. « **??** Evaluation de la grille ») ainsi nous jouerons le coup avec le maximum de chances de gagner combiné au coup avec le minimum de chances de perdre.

L'une des limites de cette méthode est qu'il faille jouer à deux joueurs, or au 2048 on joue tout seul. C'est pour cette raison qu'on utilisera pas le théorème du « minimax » mais celui d'« expected Max ». En effet ici on

concidèrera le second joueur comme l'ordinateur qui remplira de façon aléatoire une des tuilles vides après la direction choisie. Cela engage de calculer, pour chaque grille et pour toutes les possibilités de remplissage d'une tile vide, la valeur de cette grille. Il faudra ensuite faire une moyenne de toutes les valeurs calculées pour chaque direction afin de déterminer laquel est la plus intéressante. Pour un algorithme encore plus performant il serait avantageux de pouvoir étendre notre arbre à une profondeur N choisie au début du programme. Il parait évident qu'une telle technique va avoir un coût en mémoire et calcule important. Hors dans la mesure où des limitations en temps d'exécution nous sont imposés il faudra trouver un équilibre entre résultats et performances.

Pour la stratégie courte nous n'allons qu'a une profondeur de 2 (nous avons deux sous arbres) et calculons avec une probabilité de 100 % d'avoir une tuile de valeur 2 qui apparait. Pour la stratégie plus éfficiente, nous allons à une profondeur de trois et nous prenons en compte la probabilité de 10 % qu'il y est une tuille de valeur 4 qui soit généré.

4.3 Evaluation de la grille

Pour évaluer la grille, nous avons pris en compte quatre paramètres, et à chacun de ces paramètres nous leurs avons donné un coefficient pour leur donner plus ou moins d'importance. [**?**]

4.3.1 Le nombre de tiles vides

On compte le nombre de tiles vides que l'on a après avoir joué.

4.3.2 La plus grande tile

Est la valeur de la tile avec la plus grande valeur.

4.3.3 La grille la plus progressive possible

Une grille est progressive lorsque la valeur des cases augmente ou descend quelle que soit la direction. Ainsi, 2 - 4 - 8 - 16 est acceptable, tout comme 32 - 8 - 4 - 2. Mais 2 - 8 - 2 - 16 ne l'est pas (à cause de la 3eme case). À chaque mouvement on doit vérifier si le mouvement d'après rendra la grille plus progressive. Si c'est le cas, le mouvement doit être effectué. Si non, trouver un autre mouvement.

4.3.4 La grille la plus régulière possible

Pour pouvoir fusionner, les cases doivent comporter des valeurs identiques. Ainsi une suite 2 - 16 - 64 - 256 respecte la règle de progressivitè) mais aboutira à un échec car on ne pourra fusionner aucune case avec sa voisine. Il faut donc respecter un autre critère : la régularité. Et faire en sorte de ne pas avoir de cassure dans les séries. Entre créer une série 2 - 4 - 8 - 16 et créer une série 16 - 64 - 256 - 1024 on préférera donc la première, même si la possibilité d'obtenir un nombre fort comme 1024 peut être tentante a priori.

5 Une vrai interface graphique ? !

6 En bref

Bien que passionnant, le projet fu très long mais très enrichissant. Nous avons appris à travailler en groupe, utiliser des technologies nouvelles comme GIT, SDL,

Références

[1]

7 Annexes

Table des figures

FIGURE 1 – Debut du fichier grid.h

grid.h

```
1  #ifndef _GRILLE_H_
2  #define _GRILLE_H_
3  /**
4   * \file grid.h
5   * \brief Contains structures and functions needed to play 2048 game.
6   **/
7
8  #include <stdbool.h>
9
10 /** Grid dimension */
11 #define GRID_SIDE 4
12
13 /**
14  * \brief Contain game's status: tiles and current score.
15  *
16  *              X
17  *       0  1    ... GRID_SIDE-1
18  *      +--+--+- ... -+--+
19  *      |  |  |  ...  +  | 0
20  *      +--+--+- ... -+--+
21  *      |  |  |  ...  |  | 1
22  *      +--+--+- ... -+--+          Y
23  *      ...      ...       ...
24  *      +--+--+- ... -+--+
25  *      |  |  |  ...  |  | GRID_SIDE-1
26  *      +--+--+- ... -+--+
27  */
28 typedef struct grid_s* grid;
29
30 /**
31  * \brief Log_2-encoded tile : 0 is empty, i is 2**i.
32  */
33 typedef unsigned int tile;
34
35 /**
36  * \brief List of accepted movement in the game
37  */
```

FIGURE 2 – Deuxième version d'affichage de la grille

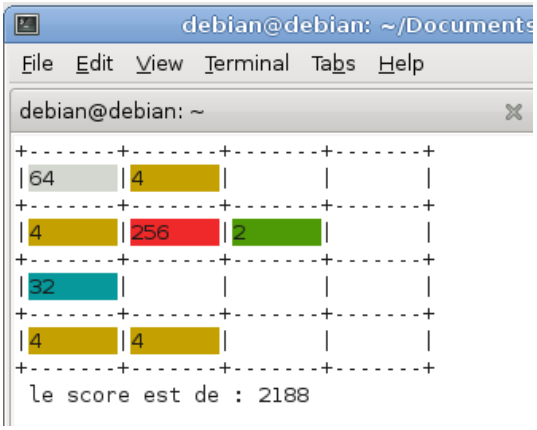

FIGURE 3 – Résultat exécution des tests

```
debian@debian:~/Documents/Cours/EDD/EDD_project/build$ make test
Running tests...
Test project /home/debian/Documents/Cours/EDD/EDD_project/build
    Start 1: test_new_grid
1/7 Test #1: test_new_grid ....................   Passed    0.01 sec
    Start 2: test_homogeneite_add_tile
2/7 Test #2: test_homogeneite_add_tile ........   Passed    0.02 sec
    Start 3: test_copy_grid
3/7 Test #3: test_copy_grid ...................   Passed    0.01 sec
    Start 4: test_can_move
4/7 Test #4: test_can_move ....................   Passed    0.02 sec
    Start 5: test_game_over
5/7 Test #5: test_game_over ...................   Passed    0.00 sec
    Start 6: test_do_move
6/7 Test #6: test_do_move .....................   Passed    0.01 sec
    Start 7: test_basic_strategy
7/7 Test #7: test_basic_strategy ..............   Passed    1.00 sec

100% tests passed, 0 tests failed out of 7

Total Test time (real) =   1.12 sec
debian@debian:~/Documents/Cours/EDD/EDD_project/build$
```