Rapport du Projet de Programmation
Rush-Hour

Groupe A2_E - BESSE Jonathan, CAUBET Christophe, CHENE Emmanuel

27 avril 2016

Table des matières

Introduction

Dans le cadre d'un Projet pour le cours de *Projet de Programmation 1 et Environnement de Développement*, nous avons eu pour objectif de développer une version textuelle d'un jeu de Rush-Hour puis également d'Âne-Rouge, ainsi que de créer un solveur capable de donner le nombre minimal de coups nécessaires pour résoudre une partie quelconque de chacun de ces deux jeux.

1 Les programmes

1.1 Le Rush-Hour

Pour la première partie nous ne devions implémenter qu'une version de **Rush-Hour** textuelle jouable dans la console Linux. Pour cela, à partir des fichiers `piece.h` et `game.h` (et avec le `test_piece.c`), nous avons développé les parties `piece.c`, `game.c` ainsi qu'un `main.c` et le `test_game.c` permettant de tester `game.c`.

1.2 L'Âne-Rouge

Pour cette nouvelle partie, il a fallu adapter le code précédemment crée afin de pouvoir jouer, au choix, au **Rush-Hour** ou à l'**Âne Rouge**. Pour cela nous avons simplement adapté la plupart des fonctions pour les rendre fonctionnelles peu importe le jeu joué, et avons ajouté un `gameover.c` et un `gameover.h` pour différencier les cas de Game Over entre les deux jeux. Nous avons également adapté les tests présents dans `test_piece.c` et `test_game.c` pour les rendre efficaces sur les nouvelles versions du code.

1.3 Le Solveur

Ici, nous avons réfléchi à différentes manières d'implémenter un solveur efficace. Au final, nous avons décidé d'implémenter un solveur qui effectue tous les mouvements à déplacement de 1 à partir d'une configuration du jeu (en excluant les configurations déjà rencontrées), et tant qu'il ne trouve pas de Game Over, il poursuit en effectuant 1 déplacement depuis chacune de ces configurations. A la fin, donc dès que l'on rencontre un Game Over, on retourne le nombre de mouvements effectués pour atteindre ce Game Over. Si plus aucun mouvement n'est effectuable et qu'aucun Game Over n'a été trouvé, alors on retourne -1.

1.4 L'Interface Graphique

Afin de créer une interface graphique, nous avons décidé d'utiliser la `SDL 2.0`. L'interface graphique est assez simpliste, et consiste en des carrés de couleurs différentes(selon les voitures/pièces), déplaçables à la souris sur l'interface.

2 Objectifs, Problèmes, Optimisation

2.1 Objectifs

Notre objectif principal étant, il faut le dire, l'état fonctionnel des programmes, cet objectif est majoritairement atteint, tant concernant le fonctionnement de la version textuelle du **Rush-Hour** et de l'**Âne Rouge** et des programmes de tests, que le fonctionnement, bien que lent et avec du retard, du **Solveur**.

2.2 Problèmes rencontrés

La plupart des problèmes rencontrés ont été réglés assez vite, mis à part pour deux :

— Dans le **Solveur**, nous avions commencé par implémenter une certaine version, vérifiant chemin par chemin lequel était le plus rapide. Finalement, cette méthode a apporté divers bugs, et s'est avérée très peu efficace, alors nous avons recommencé la partie "supérieure" du **Solveur** de la manière décrite plus haut, et avons pu obtenir un programme fonctionnel et plus optimisé ;

— Dans l'**Interface Graphique**, nous avons rencontré divers problèmes, notamment liés à la librairie graphique SDL2, des problèmes liés pour certains au rendu graphique, mais pour d'autre à l'interaction entre l'utilisateur et le jeu.

2.3 Optimisation et Complexité

Nous avons fait de nombreux ajustements au code pour le rendre plus optimisé, visuellement mais aussi efficacement parlant. Visuellement, nous avons principalement factorisé certaines parties de fonctions pour les rendre moins redondantes ; Efficacement, nous avons tenté de rendre certaines parties du code plus "courtes" dans leur exécution, en réduisant le nombre de calculs et d'appels nécessaires. Malheureusement, certaines fonctions n'en restent pas moins "lourdes", surtout celles implémentées dans le solveur, dont les principales :

— `find_shortest_path`, dont la complexité est, dans le pire des cas, du nombre de mouvements effectuables sur le plateau à partir de la

configuration initiale. A chaque itération de cette boucle, elle appelle `shortest_determine_fils` ;

— `shortest_determine_fils`, dont la complexité est, dans le pire des cas, de nb_pieces multiplié par le nombre de mouvements déjà effectués, nb_pieces étant le nombre de pièces présentes sur le plateau de jeu, sachant que pour chaque itéraction de cette boucle, on appelle d'autres fonctions(notamment `games_are_equals` ainsi que `c_played_add_c`, l'une faisant elle-même des boucles, l'autre pouvant appeler un realloc).

Au final, soit nM le nombre de mouvements effectuables, on obtient, rien que pour ces deux fonctions, une complexité, dans le pire des cas, de `nM*log(nM)`, ce qui est très conséquent sachant que pour 6 pièces sur un plateau, il y a en moyenne environ 10 mouvements effectuables à partir de chaque configuration, la complexité prend rapidement des tournures drastiques au fur et à mesure ainsi qu'en fonction du nombre de pièces, et du nombre de mouvements nécessaires pour résoudre un niveau.

3 Conclusion

Au cours de ce projet, nous avons pu mettre en pratique les différents outils et méthodes vus en cours mais aussi apprendre de nouveaux. Dans un premier temps, **GitHub** et **Git** nous ont permis de mieux gérer notre projet et ses différentes versions, la gestion du travail est alors devenue plus facile. Nous avons pu réutiliser **GDB** qui nous a permis de mieux comprendre et corriger les différents problèmes que nous avons rencontrer dans notre code, de même pour Valgrind. Dans la dernière partie, nous avons appris à intégrer une bibliothèque graphique à notre projet (**SDL**) qui nous permet de le faire tourner autrement qu'en ligne de commande.
Pour conclure, la réalisation de ce projet nous a permis d'apprendre de nouvelles méthodes pour la réalisation de notre le travail de groupe afin d'être plus efficace grâce à de nouveaux outils et d'autres que nous avons pu apprendre à mieux maitriser.