Rapport Projet Programmation

Groupe A : Ali Cherifi, Valentin Gaillard, Julien Pilleux

23/04/2016

1 Les objectifs

1.1 Objectifs atteints

Au sein de ce projet, plusieurs objectifs nous ont été fixés et nous avons essayé de tous les remplir. Les objectifs accomplis sont donc les suivants :

— La version 1 de RushHour a été rendue en temps et en heure et pleinement fonctionnelle. Certains bugs étaient encore présent mais aucun de ses bugs n'étaient pénalisant pour jouer à la première version du jeu.

— La remise de la version 2 s'est elle aussi effectuée sans aucun bug majeur. De plus, les bugs présent dans la première version ont pu être corrigés. Notre code étant fait de manière à être facilement maintenable, nous n'avions que très peu de changement à opérer afin d'inclure le puzzle de l'âne rouge dans le projet. Rendre un code maintenable étant un élément important au sein d'un projet, nous pouvons dire que nous l'avons accompli.

— Un des autres objectifs accompli a été de rendre un code commenté et lisible par autrui. En effet, la relecture croisée nous a permis de nous rendre compte que note code était suffisamment commenté pour permettre une lecture plus aisé au groupe qui nous a évalué. Cela nous a également permis d'apprendre à rédiger des commentaires dans une forme correcte.

— Un des autres gros objectifs principaux à accomplir a été le solveur. En ayant pensé, écrit et conçu l'algorithme de résolution, nous avons réussi à implémenter un algorithme "naïf" permettant la résolution de jeu simples de RushHour et de l'Ane rouge. Nous avons donc réussi à partir de zéro pour implémenter un solveur fonctionnel.

— Un des autres objectifs accompli a été l'apprentissage et l'utilisation d'une librairie externe ç celles proposées de base par C, la SDL. Nous avons pu créer une interface graphique simple mais fonctionelle pour notre jeu nous montrant notre progression depuis le début du projet.

— Au travers de ce projet nous avons également développé nos compétences dans l'utilisation des ressources et outils offert en programmation. Nous sommes désormais capables d'utiliser les outils de débuggage en terminal ou dans un IDE, de nous servir des programmes de gestion de contrôle de version (GIT/SVN), et des programmes de contrôle de la mémoire (valgrind).

1.2 Objectifs non atteints

— Tout d'abord, nous avons les nombreuses fuites mémoires contenues dans le solveur. Celles ci sont importantes et nous ne sommes pas parvenues à les éliminer malgré nos recherches et l'aide des outils proposés.

— Ensuite, nous avons le fait que le solveur ne peut pas résoudre des jeux trop compliqués. Le solveur sature vite, fait beaucoup trop d'allocation mémoire, et met énormément de temps à résoudre des jeux complexes. Il nous a également été possible de voir grâce à la compétition de solveur qu'il ne retournait pas forcément une valeur correcte. Il faudrait donc revoir son implémentation ainsi que peut être celle du tas et de la file.

— Dans les objectifs non accomplis nous avons également la non documentation via Doxygen. Cela peut devenir très embêtant si nous avions dû donner une librairie contenant par exemple la pile et le tas. Le programmeur en utilisant cette librairie n'aurait eu aucun moyen de connaître les services offerts par cette librairie dans disposer des fichiers d'en-tête. Nous ne sommes donc pas en mesure de fournir une librairie complète.

— Au sein de l'interface graphique, il est impossible de déplacer un véhicule sur plusieurs cases. tout les déplacements se font case par case, ce qui peut être embêtant pour le joueur.

2 Problème technique

2.1 L'affichage dans le terminal

Au moment d'afficher notre jeu dans le terminal, nous avons cherché à avoir une grille et des instructions les plus propres possibles. Nous avons déjà eu recours au printf classique, mais l'affichage se faisais en dessous et s'empilait ainsi dans le terminal. Cela ne nous convenait pas alors nous avons entendu parlé d'une méthode nous permettant de ré-afficher par dessus ce que nous avons déjà inscrit. Il fallait toujours utiliser printf mais avec des paramètres supplémentaires au début de celle-ci.

2.2 Les paramètres de la fonction printf

```
printf ("\033[%d;%dH%s\n", INSTRUCTION_LINE, 0, "Chose a piece : Type the number of the piece.
```

Ce paramètre additionnel nous a permis de pouvoir choisir, directement dans le terminal, où écrire. Le premier paramètre entier, nous permet de choisir quelle est la ligne du terminal nous allons écrire, la ligne est choisie à partir de la vu courante, elle est donc comptée depuis la première ligne affichée du terminal. Ici la variable instruction line correspond à la ligne en dessous de la grille. Un

autre problème rencontré est que à chaque affichage de la grille, la vue courante du terminal changeait, donc pour avoir toujours la même ligne à chaque tour de boucle, il a juste fallut ajouter une fonction permettant de clean l'affichage du terminal (System ("clear")). Le second paramètre entier correspond tout simplement au nombre de caractère qu'il faut laisser entre le bord gauche du terminal et le premier caractère de la chaîne de caractère que l'on veut afficher. Ici, le fait de mettre 0 permet de coller la chaîne au bord du terminal. Finalement, le dernier paramètre, la chaîne de caractère est tout simplement la chaîne à afficher aux coordonnées précédemment choisies. Avec ceci, nous obtenons un affichage propre.

2.3 Le retour à la ligne du prompt

Même avec ce printf, il nous restait le soucis du prompt qui revenait à la ligne, et les options saisies ne s'affichait pas sur la même ligne. En effet, une fois une option saisie, le curseur se positionnait automatiquement sur la ligne d'en dessous. Pour palier à ça, il a fallut créer une fonction reset cursor qui replaçait le prompt sur la bonne ligne.

```
Void reset_cursor (void) {
  printf("\033[%d;%dH%s\n", PROMPT_LINE, 0, "
  printf("\033[%d;%dH%s\n", INSTRUCTION_LINE, PROMPT_RESET, "");
}
```

Cette fonction utilise le printf précédemment vu. Le premier printf sert à réinitialiser la ligne de saisie afin qu'elle apparaisse vide, pour que la prochaine saisie puisse être entrée sans chevauchement de caractères. Puis, la seconde instruction nous sert à imprimer une chaîne de caractère vide sur la ligne au dessus de la ligne de la saisie, pour que le curseur se positionne automatiquement sur cette dernière.

3 Étude de la complexité

Ici, nous détaillerons la complexité des différentes fonctions du solveur afin d'établir une estimation de la complexité du solveur.

3.1 Lecture de fichier

```
game game_from_file(FILE * f){


    int width_game, height_game, nb_pieces;
    int x,y, width_p, height_p;
    int can_move_y;
    int can_move_x;

    fscanf(f,"%d %d", &width_game, &height_game);
    fscanf(f,"%d",&nb_pieces);

    piece *pieces = malloc (sizeof(*pieces) * nb_pieces);


    for (int i = 0; i < nb_pieces; i++){
        fscanf(f,"%d %d %d %d %d %d", &x, &y, &width_p, &height_p, &can_move_x, &can_move_y);
        pieces[i] = new_piece(x,y,width_p,height_p,can_move_x,can_move_y);
    }

    game g = new_game(width_game,height_game,nb_pieces,pieces);

    for (int i = 0; i < nb_pieces; ++i){
        free(pieces[i]);
    }
    free(pieces);
    return g;


}
```

La lecture de fichier ne comporte qu'une seule boucle. On peut donc ajouter à la complexité autre des fonctions fscanf et la boucle dépendant du nombre de pièces que comporte le jeu ainsi que celle de new game. new_game a pour complexité celle des mallocs ainsi que celle de la boucle remplissant le tableau de pièce. La complexité de cette fonction est donc si n est le nombre de pièces : 2n ajouté a la complexité des autres fonctions.

3.2 Création du jeu et des pièces

On a ici une complexité dépendant pour la création d'un jeu de la boucle for permettant d'allouer le tableau donc n. Pour la fonction new_piece, la complexité ne dépends donc que du malloc puisque tout le reste est en temps constant.

3.3 Destructions et copies

La destruction de jeu dépend du nombre de pièces c'est-à-dire n. La destruction d'une pièce est en temps constant. Les copies quant à elle sont plus

coûteuses.Dans le pire des cas, le tableau de destination est plus grand que celui de la source. Il faut donc libérer tout le tableau, donc faire une boucle de taille n puis le réallouer de taille "nsource". On a donc : $n + nsource$.

3.4 Mouvement d'une pièce

```
bool test_move (game g, int piece_num, dir d, int distance){ //Create a copy of a piece to calculate later if the move don't intersect and so don't
change the real piece.

    piece move_copy = new_piece_rh(0, 0, true,true);
    copy_piece(game_piece(g, piece_num), move_copy);

    for (int test_distance = 0; test_distance < distance; ++test_distance) {    //This loop test the move 1 by 1 to be sure the piece don't transperc
and exceed another piece.
        move_piece(move_copy, d, 1);
        for (int i = 0; i < piece_num; ++i) {    //Double loop to skip the intersect test with the piece itself.
            if (intersect(move_copy, g->pieces[i])){
                free(move_copy);
                return false;
            }
        }
        for (int i = piece_num + 1; i < game_nb_pieces(g); i++) {
            if (intersect(move_copy, g->pieces[i])){
                free(move_copy);
                return false;
            }
        }
    }
    free(move_copy);
    return true;
```

Cette fonction fait uniquement des appels ver test_move elle voit donc sa complexité déterminée par celle-ci.Tout d'abord la copie effectuée influe sur la complexité. Pour ne pas avoir à bouger la pièce de base et ne pas modifier le jeu on copie le jeu, on a donc une complexité en n, puis ensuite la boucle parcourant toutes les autres pièces afin de tester les intersections est aussi en n. On a donc une complexité de 2n.

3.5 Obtention de l'indice d'une piece en fonction des coordonnées

La fonction game_square_piece se contente d'une simple boucle for parcourant toutes les pièces du jeu. Nous avons donc dans le pire des cas une complexité ne dépendant que du nombre de pièces c'est-à-dire n.

3.6 Strcutures de données : file et tas

Commençons d'abord par la file. La plupart des fonctions font des allocations mémoires ou des affections. Les seules fonctions dont la complexité est élevée sont donc la fonction de ré-arrangement de la file et de libération de la file. La fonction de ré-arrangement parcours tous les éléments contenus dans la file moins le premier qui est écrasé par le suivant. On a donc si n est le nombre d'éléments une complexité en n-1.

```c
void rearrange_queue(queue q){
    int n = q->index_top;

    // Shift all games stocked in the array.
    for (int i = 0; i < n; ++i)
        q->game_array[i] = q->game_array[i + 1];

    q->index_top -= 1;
}
```

Quant à la suppression, cela prends en compte tout les éléments contenus dans le tableau c'est-à-dire n.

Le tas quant à lui est un plus complexe. La fonction de création de tas doit affecté toutes la valeurs à NULL. On a donc pour n éléments une complexité en n. L'augmentation de la taille du taille prends également en compte un boucle dépendant du nombre d'éléments à rajouter soit encore ici n. La recherche dans le tas doit comparer pour chaque jeu si les pièces sont égales. Dans le pire des cas, le jeu n'est pas dans le tas. Il faut donc parcourir les n jeux déjà présent et regarder si les k pièces sont différentes on a donc une complexité en $n * k$.

```c
bool heap_game_search (heap h, game g) {

    for (int i = 1; i < h->nb_elements; ++i) {
        if (games_equality(g, h->games[i]) == true)
            return true;
    }
    return false;
}
```

```c
bool pieces_equality(cpiece p, cpiece p2){
    return get_x(p) != get_x(p2) || get_y(p) != get_y (p2);
}

bool games_equality(game g, game g2){
    for (int i = 0 ; i < game_nb_pieces(g); ++i){
        if (pieces_equality(game_piece(g, i), game_piece(g2, i)) == true){
            return false;
        }
    }
    return true;
}
```

La suppression du tas quant à elle parcourt tout le tas contenant n éléments et supprime pour chaque jeux les k pièces présentes. On a donc un complexité en $k * n$

4 Redondance de code

Durant l'implémentation du projet nous nous sommes aperçus que certaines parties du code étaient dupliquées, nous avons donc créé des fonctions afin de factoriser le code et d'améliorer la lisibilité du projet. Les deux factorisations

de code notables de notre projet sont celle pour la fonction play move dans le fichier game.c et celle pour le solver.

4.1 game.c

La fonction play move du fichier game.c permet de tester si le déplacement d'une pièce est possible, si c'est le cas on utilise la fonction move piece du piece.c. Il faut donc tester pour chaque direction possible, une grande partie du code se répétant donc 4 fois nous avons créé une fonction test move permettant la factorisation du code :

```
bool test_move (game g, int piece_num, dir d, int distance){ //Create a copy of a piece to calculate later if the move

    piece move_copy = new_piece_rh(0, 0, true,true);
    copy_piece(game_piece(g, piece_num), move_copy);

    for (int test_distance = 0; test_distance < distance; ++test_distance) {    //This loop test the move 1 by 1 to be
        move_piece(move_copy, d, 1);
        for (int i = 0; i < piece_num; ++i) {    //Double loop to skip the intersect test with the piece itself.
            if (intersect(move_copy, g->pieces[i])){
                free(move_copy);
                return false;
            }
        }
        for (int i = piece_num + 1; i < game_nb_pieces(g); i++) {
            if (intersect(move_copy, g->pieces[i])){
                free(move_copy);
                return false;
            }
        }
    }
    free(move_copy);
    return true;

}
```

On crée une copie de la pièce que l'on souhaite déplacer, on utilise intersect pour savoir si le déplacement est possible, puis on supprime la copie et on retourne le résultat.
On obtient donc un code plus concis pour la fonction play move :

```
 if (can_move_x((piece) game_piece(g, piece_num))) {    //Check if the piece is horizontal

     if (d == LEFT && get_x((piece) game_piece(g, piece_num)) - distance >=0) {    // Piece is horizontal and check if
         if (test_move(g,piece_num,d,distance)){
             move_piece((piece) game_piece(g, piece_num), d, distance); //Move the real piece, increase the number of
             g->nb_moves += distance;
             return true;
         }
     }
```

4.2 solver.c

Pour le solveur nous avons eu besoin de factoriser le code contenu dans le while du main, nous avons utilisé deux fonctions, new configuration horizontal

et vertical :

```
void new_configuration_horizontal(game g, int i, heap game_heap, queue q){
    game copy = new_game_hr(0,NULL);
    copy_game(g,copy);
    if (play_move(copy,i,RIGHT, 1)){
        if(heap_game_search(game_heap, copy) == false){
            push(q,copy);
            game heap_copy = new_game_hr(0,NULL); //Copie le jeu car passage de réference et donc destruction de la re
            copy_game(copy,heap_copy);
            heap_add(game_heap,heap_copy);
        }
    }
    else
        delete_game(copy);
    game copy2 = new_game_hr(0,NULL);
    copy_game(g,copy2);
    if (play_move(copy2,i,LEFT, 1)){
        if(heap_game_search(game_heap, copy2) == false){
            push(q,copy2);
            game heap_copy = new_game_hr(0,NULL); //Copie le jeu car passage de réference et donc destruction de la re
            copy_game(copy2,heap_copy);
            heap_add(game_heap,heap_copy);
        }

    }
    else
        delete_game(copy2);

}
```

Dans ces deux fonctions on essaie de faire bouger une pièce dans une direction, si c'est possible alors on cherche si le jeu existe dans le tas, si ce n'est pas le cas on push le jeu dans la file puis dans le tas.
Ces deux fonctions permettent donc de simplifier le code à l'intérieur du while :

```
   while(! (*gameover_tab[gameover_function])(g)){
       game t = pop(q);  //Storing the reference from the queue
       copy_game(t, g);  //Copying into the current game
       rearrange_queue(q);  //Sort the queue
       delete_game(t); //Deleting the old reference
       for (int i = 0; i < game_nb_pieces(g); i++){
           if (can_move_x(game_piece(g,i))){
               new_configuration_horizontal(g, i, game_heap, q);
           }
           if (can_move_y(game_piece(g,i))){
               new_configuration_vertical(g, i, game_heap, q);
           }
       }
   }
```