Projet de Programmation :
RUSH HOUR / ÂNE ROUGE

IN400A1 - Groupe 6
Aliénor BRABANT
Abdoul DIALLO
Gérard LÉZÉ
Manon PHILIPPOT

20 Avril 2016

Table des matières

Première partie

Le projet

Chapitre 1

Que fait le projet ?

Dans de le cadre de l'UE "Environnement de Développement et Projet de Programmation 1" il nous a été donné comme projet la réalisation d'un jeu appelé *Rush-Hour* et de son solveur. Ce jeu est un casse tête. La grille, ou aire de jeu, est un carré de 6 cases sur 6. Les pièces placées sur le jeu sont des véhicules. On distingue :

— Les voitures, qui occupent 2 cases adjacentes.
— Les camions, qui occupent 3 cases adjacentes formant une ligne droite.
— La voiture rouge, qui est aussi une voiture, mais qui se distingue des autres par le fait que c'est celle-là qu'il faut essayer de sortir de l'aire de jeu.

Le seul endroit où la voiture rouge peut sortir se trouve à droite de la case située en 3ème ligne et en 6ème colonne. Un véhicule ne peut se déplacer que sur une seule dimension, verticalement ou horizontalement, selon qu'il a été placé, respectivement, verticalement ou horizontalement. Il n'est pas permis d'enlever (ou de placer) un véhicule en cours de jeu. Le but du jeu est donc de faire sortir la voiture rouge de l'aire de jeu, en n'y effectuant que des déplacements légaux : c'est à dire un déplacement sur des cases libres sans passer outre d'autre véhicule.

Au cours du projet le cahier des charges évolue pour permettre d'étendre le *Rush-Hour* et d'implémenter aussi le jeu de l'*Âne-Rouge*, similaire au *Rush-Hour* mais dont les pièces peuvent être de taille et de largeur toutes deux différentes de 1 et se déplacent à la fois horizontalement et verticalement. De plus les dimensions du plateaux sont de 4 cases sur 5. Le nouveau solveur devant résoudre les configurations de l'*Âne-Rouge*.

Le projet sert de prétexte pour nous faire appréhender de manière ludique les nombreux outils utiles et/ou nécessaires au développement. Les outils de compilations *Make* et *CMake*. Les logiciels de gestions de version *git* et *svn*. L'outil de débogage *gdb*. L'outil de recherche des fuites mémoires *valgrind*. L'outil de couverture de code *gcov*. Ainsi que l'importance des tests. Nous avons aussi pu nous confronté au mécanisme du travail en équipe et à l'organisation que cela demande.

Chapitre 2

Description des fonctions implémentées

2.1 Fichier piece.c

Le fichier piece.c implémente les fonctions nécéssaires à la création d'une pièce de jeu pouvant être utilisée pour les jeux de Rush Hour et d'Âne Rouge. Il est possible de réutiliser ce fichier source pour implémenter les pièces de jeux dont le fonctionnement est similaire.

2.1.1 Structure pièce

Une piece est modélisée par une structure C possédant six données membres :
— x : abscisse du coin en bas à gauche de la pièce.
— y : ordonnée du coin en bas à gauche de la pièce.
— *width* : largeur de la pièce.
— *height* : hauteur de la pièce.
— *move_x* : indique si la pièce est autorisée à bouger horizontalement.
— *move_y* : indique si la pièce est autorisée à bouger verticalement.

2.1.2 new_piece

La fonction *new_piece* crée une pièce en choisissant sa position sur le plateau de jeu, sa taille ainsi que ses capacités de mouvement. Elle consiste à allouer de la mémoire pour la structure *piece* et à initialiser chaque donnée membre de la structure par l'argument correspondant passé en paramètre. Elle retourne la pièce en question. Le principe de cette fonction est utilisé pour implémenter la fonction *new_piece_rh* permettant uniquement de créer une pièce de jeu pour Rush Hour.

2.1.3 delete_piece

La fonction *delete_piece* supprime une pièce. Elle consiste à libérer la mémoire allouée à la pièce passée en paramètre en effectuant un *free*. Elle ne retourne rien.

2.1.4 copy_piece

La fonction *copy_piece* copie la pièce passée en premier paramètre dans la pièce passée en second paramètre. On considère que la pièce copiée a déjà été initialisée avant l'appel à la fonction *copy_piece*. Elle consiste donc à uniquement copier les données membres de la pièce originale dans les données membres de la pièce copiée. Elle ne retourne rien.

2.1.5 move_piece

La fonction *move_piece* bouge la pièce p passée en premier paramètre, dans la direction *dir* passée en second paramètre, d'une distance *dist* passée en troisième paramètre. Si *dir* vaut *RIGHT*, alors l'abscisse x de p est incrémentée de *dist*. Si *dir* vaut *LEFT*, alors l'abscisse x de p est décrémentée de *dist*. Si *dir* vaut *UP*, alors l'ordonnée y de p est incrémentée de *dist*. Si *dir* vaut *DOWN*, alors l'ordonnée y de p est décrémentée de *dist*. Cette fonction ne retourne rien. Elle se contente uniquement de bouger la pièce, sans prendre en compte les mouvements invalides (mouvement en dehors du plateau, intersection de pièces, direction invalide). C'est l'implémentation de la fonction *play_move* du fichier source game.c qui s'en charge (cf. sous-section 2.2.8).

2.1.6 intersect

La fonction *intersect* teste si les deux pièces passées en paramètre s'intersectent. Elle consiste à comparer les coordonnées de chaque cellule de la première pièce avec les coordonnées de chaque cellule de la seconde pièce. Elle retourne un bouléen, qui vaut *true* si les deux pièces ont au moins une cellule en commun, et *false* dans le cas contraire.

2.1.7 Getteurs

Les fonctions *get_x*, *get_y*, *get_height*, *get_width*, *can_move_x* et *can_move_y* sont des getteurs qui retournent respectivement l'abscisse x, l'ordonnée y, la hauteur *height*, la largeur *width*, le booléen *move_x* indiquant la capacité de mouvement horizontal et le booléen *move_y* indiquant la capacité de mouvement verticale de la pièce. Il existe également une fonction *is_horizontal* dont l'implémentation est similaire à celle de la fonction *can_move_x*.

2.2 Fichier game.c

Le fichier game.c implémente les fonctions nécéssaires à la création d'un plateau de jeu, et à la gestion d'un jeu de Rush Hour et d'Ane Rouge. Il est possible de réutiliser ce fichier source pour implémenter des jeux dont le fonctionnement est similaire.

2.2.1 Structure game

Un jeu est modélisé par une structure C possédant cinq données membres :
— *width* : largeur du plateau de jeu.
— *height* : hauteur du plateau de jeu.
— *nb_pieces* : nombre de pièces sur le plateau de jeu.
— *nb_moves* : nombre mouvement depuis le début du jeu.
— *pieces* : tableau de pièces.

2.2.2 new_game

La fonction *new_game* crée un jeu en choisissant ses dimensions ainsi que la configuration initiale de celui-ci par l'intermédiaire d'un set de pièces. Elle consiste à allouer de la mémoire pour la structure *game* et à initialiser chaque donnée membre de la structure par l'argument correspondant passé en paramètre. L'initialisation du tableau de pièces de la structure est assez particulière. Il faut d'abord allouer l'espace mémoire pour le tableau de pièces de la structure, puis copier chaque pièce du tableau fourni en paramètre dans le tableau *pieces* de la structure par l'intermédiaire de la fonction *copy_piece* expliquée à la sous-section 2.1.4. Cette fonction retourne le jeu en question. Le principe de cette fonction est utilisé pour implémenter la fonction *new_game_hr* permettant uniquement de créer un jeu de Rush Hour.

2.2.3 delete_game

La fonction *delete_game* supprime un jeu. Elle consiste à libérer la mémoire allouée au jeu passé en paramètre en supprimant dans un premier temps toutes les pièces du tableau de pièces par l'intermédiaire de la fonction *delete_piece* expliquée à la sous-section 2.1.3, puis en effectuant un *free* sur le tableau et un *free* sur le jeu en question. Elle ne retourne rien.

2.2.4 copy_game

La fonction *copy_piece* copie le jeu passé en premier paramètre dans le jeu passé en second paramètre. On considère que le jeu copié a déjà été initialisé avant l'appel à la fonction *copy_game*. Elle consiste donc à uniquement copier les données membres du jeu original dans les données membres du jeu copié. La copie du tableau de pièces s'effectue au moyen d'une boucle *for* parcourant le tableau et de la fonction *copy_piece* expliquée à la sous-section 2.1.4. Elle ne retourne rien.

2.2.5 game_piece

La fonction *game_piece* retourne la pièce portant l'indice passé en paramètre, du jeu également passé en paramètre. Elle consiste à renvoyer le contenu du tableau *pieces* à l'indice donné. Si cet indice est supérieur ou égal au nombre de pièces, alors la fonction retourne *NULL*.

2.2.6 game_square_piece

La fonction *game_square_piece* retourne le numéro de la pièce située aux coordonnées x et y passées en paramètre, du jeu également passé en paramètre. Elle consiste à parcourir toutes les cellules constitutives des pièces du jeu en comparant leurs coordonnées à celles données en paramètre. S'il y a égalité, alors le numéro de la pièce en question est renvoyé, sinon la valeur -1 est renvoyée.

2.2.7 game_over_hr

La fonction *game_over_hr* teste si le jeu passé en paramètre est terminé, c'est-à-dire si la pièce 0 à atteint la sortie. Elle consiste à comparer les coordonnées de la pièce 0 avec les coordonnées de la sortie. Elle renvoit un booléen. La fonction *game_over* de l'Âne Rouge est implémentée dans le fichier source main.c expliquée à la section 2.3.1.

2.2.8 play_move

La fonction *play_move* tente de bouger dans le jeu passé en premier paramètre, la pièce dont l'indice est passé en deuxième paramètre, dans une direction passée en troisième paramètre et d'une distance passée en quatrième paramètre. Si le mouvement est valide la pièce est bougée et la fonction retourne *true*. S'il n'est pas valide (la pièce sort du plateau, la direction est incompatible avec le type de pièce, la pièce croise une autre pièce), la pièce n'est pas bougée et la fonction retourne *false*.

2.2.9 game_nb_moves

La fonction *get_nb_moves* retourne le nombre de mouvements depuis le début de la partie. Sa valeur est incrémentée à chaque coup joué.

2.2.10 Getteurs

Les fonctions *game_nb_pieces*, *game_width* et *game_height* sont des getteurs qui retournent respectivement le nombre de pièces du jeu, la largeur et la hauteur du plateau de jeu.

2.3 Fichier main.c

Maintenant que nous pouvons créer un plateau et les pièces qu'il contient, il reste à créer le jeu. C'est à dire les mécanismes qui vont permettre au joueur de communiquer avec l'ordinateur afin de jouer. Nous avons donc créer un fichier "main.c" qui constitue le jeu et se sert des modules précédemment vu.

2.3.1 La création d'un visuel du jeu

Dans un premier temps pour permettre au joueur de jouer, il lui faut une manière de se représenter le jeu. Nous avons donc choisi d'afficher sur le terminal une version texte de la grille

de plateau. Pour cela le jeu fait appel à une fonction : print_game qui ne retourne rien mais affiche l'état du jeu ou variable "game", passé en paramètre. Le principe est le suivant : on affiche ligne par ligne, case par case, la grille de jeu affichant une case vide si cette dernière ne contient pas de pièce ou bien le numéro de la pièce qui s'y trouve.

Dans la première version, aucune fonction de game.c ne permettaient de savoir si une pièce se trouvaient à une case ou non. Nous avons donc utilisé un tableau d'entiers (ou int) pour sauvegarder les positions des pièces. Dans un premier temps il était mis à vide à chaque appel de la fonction, puis chaque pièce y était placées une par une, en partant de leurs coordonnées (position la plus en bas à gauche) puis en parcourant la largeur ou la longueur de celle-ci.

Avec l'arrivée de la deuxième version du projet, le module game contient une fonction : game_square_pièce, qui pour un jeu passé en paramètre ainsi que deux entiers représentant les coordonnées de la case si une pièce s'y trouve et si oui laquelle. Il n'y avait plus besoin de sauvegarder les positions des pièces dans un tableau.

Cependant l'arrivée de cette deuxième version posa un nouveau problème : comment positionner la sortie selon le jeu choisi ? Celle-ci n'étant pas implémentée dans le module game mais propre au jeu choisi. Une structure game_option_s fut la réponse au problème concernant la différenciation de l'âne_rouge et du Rush-Hour mais aussi des autres jeux possiblement implémentable sur la même base :

```
struct game_option_s{
    char name[4];
    char file[30];
    int x_end;
    int y_end;
    int h_end;
};
```

x_end et y_end qui codent la positions de la sortie (à laquelle doit se trouver la pièce 0 ou pièce à sortir), h_end qui précise la hauteur de la sortie.
Un tableau de ces structures est ensuite créé :

```
typedef struct game_option_s game_option;
game_option game_chose[] = {{"-rh", "game_rh.txt",4,3,1},{"-ar", "game_ar.txt",3,1,2}};
```

L'indice du tableau permettant de connaître le type de jeu et les informations relatives.

Cette solution fut en réalité la deuxième car la première ne convenant à ce qui était imposé. Elle consistait en l'implémentation d'une pièce fantôme (qui n'aparaissait pas et ne gêner pas ls autre pièce) codant la sortie permettant une grande modularité et l'implémentation de n'importe quel type de jeu de ce type, facilitant aussi l'affichage.

2.3.2 Les interactions entre le joueur et la machine

Après avoir réalisé le visuel du plateau de jeu il fallait pouvoir jouer. Nous avons donc commencé l'implémentation de ce qui constituerait une bonne partie de la fonction main. Nous voulions la manière suivante de jouer : le joueur choisi une pièces qu'il désire bouger puis il indique dans quel direction et enfin de combien de case, afin de correspondre à la fonction play_move du module game. Pour cela une premier boucle fut créée de la manière suivante :

```
while (!game_over(g, game_i)){...}
```

Où game_over est une fonction qui grâce à l'entier game_i, renvoie le résultat de la fonction permettant de connaître si la fin du jeu est atteinte ou non comme par exemple game_over_hr pour le Rush-Hour. Après chaque mouvement, on retente de voir si le jeu c'est terminé. Si oui on s'oriente vers la fin du programme, sinon on affiche le nouvel état du jeu.

Une deuxième boucle while testant la variable good, pour savoir si le déplacement s'est bien effectuer, est imbriquée dans la première. Si le déplacement s'est bien déroulé alors good prend la valeur true. Cette boucle est utile pour les fonctionnalités cancel qui permet de rejouer le présent coup (après sélection d'une mauvaise pièce ou d'une mauvaise direction) et restart qui permet de réinitialiser le jeu à son début. En effectuant un break on sort de cette boucle mais comme le jeu n'est pas terminé on y retourne.

Dans cette boucle se trouvent trois bloc chacun consacré à la demande d'une des données suivante : pièces, direction, nombre de cases. Ils sont organisés de la même manière. Ils font appel à une fonction chargé de mettre dans une variable passée en référence la donnée choisie par l'utilisateur et renvoie un code de retour selon cette dernière : 0 s'il s'agit de "cancel", -1 pour "exit", -2 pour "restart" et 1 pour le reste. Pour demander à l'utilisateur de rentrer une donnée, nous avons fait appel à la fonction read de la bibliothèque unistd :

```
#include <unistd.h>
```

Le principe des fonctions take_piece_num, take_direction et take_number_case et le suivant : tant que le joueur ne choisie pas une donné correcte alors le programme lui affiche un message pour stipuler ce qu'il attend de lui. Une fois fait, la fonction renvoie le code de retour correspondant.

Un problème se pose toute fois pour convertir une chaîne de caractère tapée par le joueur en une variable dir. Pour ce problème une autre structure a été implémenté :

```
struct dir_option_s{
      char name[6];
      dir option;
};

typedef struct dir_option_s dir_option;
dir_option direction[] = {{"up",UP},{"down",DOWN},{"right",RIGHT},{"left",LEFT}};
```

Le tableau constitué de ces structures permet de facilement choisir la bonne option. Une comparaison à l'aide de strcmp permet de trouver la bonne structure et donc l'option associée.

Pour donner l'illusion que la pièce se déplace on fait appel à la fonction system("clear"). system permet de faire appel à une commande unix passeée en paramètre ici clear.

À la fin du jeu il ne faut pas oublier de penser à libérer la mémoire en supprimant les instances de game créée : un malloc = un free.

2.3.3 Charger un jeu

Nous avons donc un jeu opérationnel, cependant ce pose la problème de l'initialisation de ce dernier. Nous avons choisie de pouvoir implémenter le jeu grâce à un fichier texte compre-

nant toutes les configurations pour un type de jeu donnée. “game_rh.txt” pour Rush_Hour, “game_ar.txt” pour l'Âne_Rouge.

Dans un premier temps il est nécessaire de choisir le type de jeu choisi en utilisant en option dans le terminal “-rh” ou “-ar”. Toujours grâce au tableau de structure game_option(vu plus haut) et la fonction strcmp il est possible de choisir le type de jeu souhaité sous la forme d'un indice game_i et ainsi de charger le bon fichier texte.

Un level (int) sera aussi passé en paramètre dans le terminal pour indiquer quelle ligne du fichier texte sera utilisée (une ligne par configuration). Une ligne se compose de la manière suivante :

```
<width>.<height>.<nb_pièce>.<donnée_pièce_1>.<donnée_pièce_2>...
```

avec <donné_pièce> :

```
<x><y><height><width><can_move_x><can_move_y>
```

C'est la fonction int_game qui se chargera de créer le jeu avec comme paramètre le fichier et le level. Pour lire une ligne on utilise la fonction fgets, on lit autant de lignes que la valeur de level. Une fois la ligne lue on utilise la fonction strtok pour découper la ligne à l'aide du caractère “.”. Les pièces sont créer une par une dans un tableau passer en paramètre à new_game. Il faut penser à supprimer les pièces de ce tableau à la fin.

L'utilité d'un tel fichier texte est de pouvoir ajouter des configurations sans avoir à recompiler le programme.

Chapitre 3

Description des tests mis en place

3.1 Principe d'évalutation paresseuse

L'intégralité des tests effectués dans le cadre de ce projet repose sur le principe d'évaluation paresseuse. Le principe d'évaluation paresseuse consiste à procéder au calcul de la fonction en ne réalisant l'évaluation des arguments qu'au moment où ils sont effectivement utilisés. Cela a plusieurs buts : l'optimisation (éviter de calculer un résultat qui pourrait ne pas etre utilisé) et la maintenabilité (exprimer des structures de données infinies). Cependant ce mode d'évaluation présente un inconvénient : la lenteur d'exécution, bien que les concepteurs de compilateurs de langages à évaluation paresseuse apportent des solutions à ce problème.

3.2 Fonctionnement général

Chaque fichier de test est composé d'une fonction *main* exécutée au lancement de l'exécutable du test, comme celle-ci :

```
int main (int argc, char *argv[])
{
  bool result= true;

  result = result && test_equality_bool(true, test1(), "test1");
  result = result && test_equality_bool(true, test2(), "test2");
  result = result && test_equality_bool(true, test3(), "test3");
  result = result && test_equality_bool(true, test4(), "test4");
  result = result && test_equality_bool(true, test5(), "test5");

  if (result) {
    printf("Youpi !\n");
    return EXIT_SUCCESS;
  }
  else
    return EXIT_FAILURE;
}
```

Cette fonction principale possède une variale booléenne *résult* initialisée à *true* susceptible de changer de valeur au cours du déroulement des tests unitaires. Le code des lignes 5 à 9 est un exemple de évaluation paresseuse. Si *result* vaut *true*, alors le test unitaire de l'expression booléenne est exécuté et *result* prend la valeur de sa valeur booléenne de retour. Si *result* vaut *false*, alors le test unitaire de l'expression booléenne n'a pas besoin d'être exécuté, puisque l'on sait déjà que le résultat de l'expression booléenne sera *false*, et *result* prend la valeur *false*. Ainsi, le premier test est toujours exécuté, et si l'un des tests unitaires vaut *false*, alors, plus aucun test unitaire ne sera exécuté par la suite puisque par évaluation paresseuse *result* prendra toujours la valeur *false* sans arriver à l'appel du test unitaire. Lorsque tous les tests sont passés, si *result* vaut *true* alors cela signifie que tous les tests se sont bien passés (affichage d'un message "Youpi !"), sinon cela signifie que l'un des tests a échoué (affichage d'un message d'erreur personnalisé).

3.3 Fichier test_piece.c

L'intégralité des fonctions du fichier source piece.c sont testées à l'intérieur des tests unitaires *test_new_piece_rh*, *test_new_piece*, *test_copy_piece*, *test_intersect* et *test_move_piece* du fichier source test_piece.c. Toutes ces fonctions reposent sur les mêmes principe que ceux vus aux sous-sections 3.1 et 3.2 (évaluation paresseuse et utilisation de la variable booléenne *result*).

3.3.1 test_new_piece_rh

Cette fonction permet de tester le bon déroulement de la création d'une pièce de Rush Hour, quelque soit sa taille (*small*, *!small*) et son orientation (*horizontal*, *!horizontal*), et pour des coordonnées x et y variées. Il s'agit de vérifier que la nouvelle pièce crée possède bien les caractéristiques voulues (coordonnées, taille, orientation). Si les caractéristiques ne sont pas égales à celles voulues, alors la fonction *new_piece_rh* est considérée comme incorrecte. La fonction *new_piece_rh* est testée par l'intermédiaire des fonctions *get_x*, *get_y*, *get_height*, *get_width*, *can_move_x* et *can_move_y*. Toute mémoire allouée a été libérée grâce à la fonction *delete_piece* pour éviter des fuites mémoires.

3.3.2 test_new_piece

Cette fonction permet de tester le bon déroulement de la création d'une pièce quelconque, quelque soit sa taille (*width*, *height*) et ses capacités de mouvement (*move_x*, *move_y*), et pour des coordonnées x et y variées. Il s'agit de vérifier que la nouvelle pièce crée possède bien les caractéristiques voulues (coordonnées, taille, orientation). Si les caractéristiques ne sont pas égales à celles voulues, alors la fonction *new_piece* est considérée comme incorrecte. La fonction *new_piece* est testée par l'intermédiarie des fonctions *get_x*, *get_y*, *get_width*, *get_height*, *can_move_x* et *can_move_y*. Toute mémoire allouée a été libérée grâce à la fonction *delete_piece* pour éviter des fuites mémoires.

3.3.3 test_copy_piece

Cette fonction permet de tester le bon déroulement de la copie d'une pièce, pour un nombre de pièces varié. Il s'agit de vérifier que toutes les caractéristiques de la pièce copiée sont iden-

tiques aux caractéristiques de la pièce originale (coordonnées, taille, orientation). Si les caractéristiques ne sont pas identiques, alors la fonction *copy_piece* est considérée comme incorrecte. La fonction *copy_piece* est testée par l'intermédiaire des fonctions *get_x*, *get_y*, *get_width*, *get_height*, *can_move_x* et *can_move_y*. Toute mémoire allouée a été libérée grâce à la fonction *delete_piece* pour éviter des fuites mémoires.

3.3.4 test_intersect

Cette fonction permet de tester si la fonction *intersect*, qui détermine si deux pièces s'intersectent, est correcte. Il s'agit de tester la fonction *intersect* pour des pièces dont on sait qu'elles s'intersectent et pour des pièces dont on sait qu'elles ne s'intersectent pas, afin de traiter tous les cas possibles. Toute mémoire allouée a été libérée grâce à la fonction *delete_piece* pour éviter des fuites mémoires.

3.3.5 test_move_piece

Cette fonction permet de tester si un mouvement s'est bien effectué, pour un set de pièce variées initialisé préalablement et des distances variées. Il s'agit de comparer après le mouvement d'une pièce ses coordonnées attendues avec ses coordonnées réelles. Si les coordonnées ne sont pas les mêmes, alors la fonction *move_piece* est considérée comme incorrecte. La fonction *move_piece* est testée par l'intermédiaire des fonctions *copy_piece*, *can_move_x*, *can_move_y*, *get_x* et *get_y*. Toute mémoire allouée a été libérée grâce à la fonction *delete_piece* pour éviter des fuites mémoires.

3.4 Fichier test_game.c

L'intégralité des fonctions du fichier source game.c sont testées à l'interieur des tests unitaires *test_new_game_rh*, *test_new_game*, *test_copy_game*, *test_play_move*, *test_game_square_piece* et *test_game_over_hr*. du fichier source test_game.c. Toutes ces fonctions reposent sur les mêmes principse que ceux vus aux section 3.1 et 3.2 (évaluation paresseuse et utilisation de la variable booléenne *result*).

3.4.1 test_new_game_rh

Cette fonction permet de tester le bon déroulement de la création d'un jeu de Rush Hour. Il s'agit de vérifier que le nouveau jeu de Rush Hour crée possède bien les caractéristiques voulues (taille plateau, nombre de pièces, nombre de mouvements, pièces). Si les caractéristiques ne sont pas égales à celles voulues, alors la fonction *new_game_hr* est considérée comme incorrecte. La fonction *new_game_hr* est testée par l'intermédiaire des fonctions *game_width*, *game_height*, *game_nb_pieces*, *game_nb_moves*, *get_x*, *get_y*, *get_height*, *get_width*, *can_move_x* et *can_move_y*. Toute mémoire allouée a été libérée grâce à la fonction *delete_game* pour éviter des fuites mémoires.

3.4.2 test_new_game

Cette fonction permet de tester le bon déroulement de la création d'un jeu quelconque. Il s'agit de vérifier que le nouveau jeu crée possède bien les caractéristiques voulues (taille plateau, nombre de pièces, nombre de mouvements, pièces). Si les caractéristiques ne sont pas égales à celles voulues, alors la fonction *new_game* est considérée comme incorrecte. La fonction *new_game* est testée par l'intermédiaire des fonctions *game_width*, *game_height*, *game_nb_pieces*, *game_nb_moves*, *get_x*, *get_y*, *get_height*, *get_width*, *can_move_x* et *can_move_y*. Toute mémoire allouée a été libérée grâce à la fonction *delete_game* pour éviter des fuites mémoires.

3.4.3 test_copy_game

Cette fonction permet de tester le bon déroulement de la copie d'un jeu. Il s'agit de vérifier que toutes les caractéristiques du jeu copié sont identiques aux caractéristiques du jeu original (taille plateau, nombre de pièces, nombre de mouvements, pièces). Si les caractéristiques ne sont pas identiques, alors la fonction *copy_game* est considérée comme incorrecte. La fonction *copy_game* est testée par l'intermédiaire des fonctions *game_width*, *game_height*, *game_nb_pieces*, *game_nb_moves*, *get_x*, *get_y*, *get_height*, *get_width*, *can_move_x* et *can_move_y*. Toute mémoire allouée a été libérée grâce à la fonction *delete_game* pour éviter des fuites mémoires.

3.4.4 test_play_move

Cette fonction permet de tester si la fonction *play_move*, qui permet de faire un mouvement de pièce dans le jeu, est correcte. Elle consiste à effectuer tous les types de mouvements possible avec la fonction *play_move* et à comparer sa valeur de retour avec la valeur attendue. Si pour au moins un des cas les deux valeurs ne sont pas égales, alors la fonction *play_move* est considérée comme incorrecte. Toute mémoire allouée a été libérée grâce à la fonction *delete_game* pour éviter des fuites mémoires.

3.4.5 test_game_square_piece

Cette fonction permet de tester si la fonction *game_square_piece*, qui donne le numéro de la pièce occupant la case localisée par les paramètres x et y, est correcte. Un plateau de jeu est matérialisé par un tableau à deux dimensions : chaque case vide est représentée par la valeur -1, et chaque case occupée par une pièce est représentée par le numéro de la pièce en question. Le teste de la fonction *game_square_piece* consiste à comparer la valeur de chaque case du tableau à la valeur de retour de la fonction *game_square_piece*. Si pour au moins un des cas les deux valeurs ne sont pas égales, alors la fonction *game_square_piece* est considérée comme incorrecte. Toute mémoire allouée a été libérée grâce à la fonction *delete_game* pour éviter des fuites mémoires.

3.4.6 test_game_over_hr

Cette fonction permet de tester si la fonction *test_game_over_hr*, qui permet de déterminer si la pièce 0 est sortie, est correcte. Elle consiste à appeler la fonction *test_game_over_hr*

lorsque la pièce 0 n'est pas sortie, et lorsqu'elle est sortie, et de comparer la valeur de retour de la fonction à la valeur attendue. Si pour au moins un des cas les deux valeurs ne sont pas égales, alors la fonction *test_ game_ over_ hr* est considérée comme incorrecte.Toute mémoire allouée a été libérée grâce à la fonction *delete_ game* pour éviter des fuites mémoires.

Chapitre 4

Description du solveur

Le solveur demandé est un programme qui doit être capable de dire s'il existe une solution et si oui quelle est le nombre de mouvements minimum pour y parvenir.

Le principe que nous avons utilisé est le suivant : en partant de l'initialisation d'un jeu nous allons tester tous les déplacements possibles jusqu'à trouver une solution au jeu, si toutes les possibilités ont été testées sans pouvoir finir le jeu, ce dernier n'a pas de solutions. Pour être sur de trouver la plus courte ou rapide des réponses possibles nous allons procéder de la manière qui suit : à partir d'une configuration donnée on exécute toutes les possibilités possibles, c'est à dire toute possibilité qui découle de cette configuration en utilisant un seul déplacement, que l'on met dans une file. Si une configuration termine le jeu on renvoie le nombre de mouvements qu'il aura fallu pour y parvenir, sinon on refait la même chose à partir de la configuration suivante dans la file. Il est très important de ne jamais enfiler une configuration déjà testée. C'est pourquoi chaque configuration doit être sauvegardée pour pouvoir tester si une possibilité a déjà été jouée.

Le grand problème d'une telle solution est qu'elle demande énormément de mémoire. En effet (dans un cas extrême) pour un plateau de 6x6 avec 36 pièces d'une case, il existe 36 ! possibilités donc un ordre de grandeur de 10^{41}. 1Go de mémoire représente 8 x 10^9 bits. Il faudrait donc 10^{32} bits de mémoire pour sauvegarder toutes les possibilités. Heureusement ici certaines configurations ne peuvent se réaliser car les pièces se bloquent les unes avec les autres, cependant l'espace requis reste conséquent il faut donc penser à optimiser l'utilisation de l'espace mémoire.

Ici les possibilités sont rassemblées dans un arbre appelé "arbre d'analyse". La racine de cette arbre contient donc la configuration initiale du jeu. Les fils sont les configurations possibles avec un déplacement de plus (attention déplacement != de mouvements). Chaque variable de type game est convertie en un tableau d'entier int* game_int de taille nb_pieces de la manière suivante : une case contient un entier iixxyy, c'est à dire (numéro de pièce * 10000 + x * 100 + y). Chaque configuration de l'arbre est pointée, et donc représentée dans une table de hachage de taille 1024, qui contient une liste chaînée de manière dynamique. La fonction d'accès de cette table qui donne l'indice du tableau à utiliser est la suivante : (somme de tous les entiers du tableau int* game_int)%1024. On compare ensuite les tableaux déjà présents de celui que l'on propose. Si ce dernier n'existe pas alors on sauvegarde l'adresse de ce dernier dans l'arbre. On évite ainsi une copie inutile des données. L'utilisation de ce tableau d'entiers permet d'économiser

24*32 bits soit 96 octets de mémoire par nouvelle configuration sauvegardée. En revanche cette solution ralentie considérablement le calcule car il faut convertir game en game_int et pouvoir faire l'inverse pour pouvoir jouer depuis cette possibilité. Chaque noeud de l'arbre contient le tableau game_int et le nombre de mouvements qu'il aura fallu pour y parvenir. Il est donc facile de renvoyer le nombre de mouvements joués.

Notre solveur peut résoudre les possibilités les plus complexes du *Rush-Hour* avec un maximum de 3 secondes sur des ordinateurs de faible configuration et résout les cas simples du jeu de l'*Âne-Rouge* mais semble impossible de résoudre la configuration initiale de ce dernier. Nous pensons que le temps de calcul est exponentiel selon le nombre de possibilités, c'est à dire que plus l'on s'éloigne de la racine de l'arbre d'analyse, plus le nombre de possibilités augmente et plus il devient long de trouver la solution. Le programme semble correct en théorie mais semble faire trop de calcul à partir d'un certain seuil.

Deuxième partie

Le travail

Chapitre 1

Organisation et répartition du travail

1.1 Logiciel de gestion de versions : Github

Nous avons dû choisir un logiciel de gestion de versions. On nous a présenté *svn* et succinctement *git*. Après un peu de lecture, il s'est avéré que *git* a plus de fonctionnalités que *svn* et est open source. En plus, nous n'avons pas envie de tester la fiabilité du server qui nous était mis à disposition pour *svn*. Nous avons choisi *git* et le service d'hébergement *github*. Nous avons mis en place le dépôt et rédigé un tutoriel (voir tutoriel en annexe). Nous avons aussi mis en place un dépôt test pour nous permettre d'expérimenter l'utilisation de github sans conséquence sur le projet. Ce dépôt test nous a aussi permis de partager de la documentation et de nous entraider.

(voir figure 5.1 de l'annexe page 25)

Github c'est révélé à l'usage très pratique. Le système de branche permet une grande flexibilité. Les branches permettent d'expérimenter de nouvelle implémentation sans contrainte. Et de ne pas craindre de ne pas pouvoir revenir à son implémentation de départ. Il est évident qu'il facilite grandement le développement à plusieurs.

1.2 Répartition du travail

Chapitre 2

Découpage de l'archive / des modules

2.0.1 Structure du projet

Dans la première version, nous avions organisé le projet avec de nombreux dossiers et nous nous sommes retrouvés avec une arborescence beaucoup trop lourde qui nous handicapait lors du développement nous obligeant à nous déplacer inutilement de dossier en dossier. Nous avons drastiquement réduit cela.

Nous avons commencé la compilation du projet avec un Makefile. Mais nous avons rapidement décidé de passer à CMake. L'utilisation de CMake nous obligeait à chaque compilation du projet à nous déplacer dans un dossier build et le *clean* ne supprimait pas tous les fichiers de configuration. Nous avons donc eu l'idée d'utiliser un Makefile qui utiliserait un script qui ferait cela à notre place. Le Makefile nous permet aussi de supprimer intégralement les fichiers de configuration généré par CMake qui lors du développement pouvait générer des blocages. Grace à cela, nous pouvons compiler simplement et nous retrouvons une archive propre à l'exécution du *make clean*. Le développement du solveur a apporté un important nombre d'exécutable et de fichiers d'entête. Nous avons donc décidé de mettre les fichiers d'entête dans un dossier *include* dédié et de faire de même avec les fichiers du solveur.

À l'heure actuelle notre structure et notre arborescence, par rapport au cahier des charge, nous semble approprié et claire.

(voir figure 5.2 de l'annexe page 26)

2.0.2 Utilisation de Cmake

CMake permet d'automatiser la génération de makefile.

> "Il est comparable au programme Make dans le sens où le processus de construction logicielle est entièrement contrôlé par des fichiers de configuration, appelés CMakeLists.txt dans le cas de CMake. Mais CMake ne produit pas directement le logiciel final, il s'occupe de la génération de fichiers de construction standards : makefile sous Unix." (CMake. Wikipédia, L'Encyclopédie Libre, `fr.wikipedia.org/wiki/CMake`)

CMake est plus souple que Make car il permet de cibler des fichiers en leur attribuant une fonction. Par exemple, nous assignons à ce dossier, include, la fonction de contenir les fichiers d'entêtes.

```
include_directories($(NOMDUPROJET_SOURCE_DIR)/include)
```

CMake nous permet aussi de généré une commande de test.

```
add_custom_target(check COMMAND ${CMAKE_CTEST_COMMAND})
```

Et de donner un temps maximum à leur exécution.

```
set\_tests\_properties(test\_piece PROPERTIES TIMEOUT 1)
```

Il permet aussi de généré facilement une librairie. Ici la librairie game.

```
add_library (game piece.c game.c)
```

Il permet tout aussi facilement de l'utiliser.

```
add_executable(jeu main.c)
target_link_libraries(jeu game)
```

CMake est aussi portable en générant des fichiers de projet Visual Studio sous Windows, même si nous n'avons pas utilisé cette fonctionnalité. Une fois appréhendé, CMake permet d'effectuer des modifications plus rapidement et est moins verbeux que Make. Il est donc plus agréable à l'usage tous en offrant plus de fonctionnalités.

Chapitre 3

Analyse mémoire

3.1 Valgrind

L'utilisation du logiciel valgrind nous permet de constater si notre programme contient des fuites mémoire. Il nous a permis par exemple de constater de telles fuites sur le jeu et de les corriger. Elles venaient du tableau de pièce utilisé pour l'initialisation du jeu mais qui n'était pas libéré. Valgrind nous a aussi permis de constater des fuites mémoire sur le solveur qu'il nous reste à corriger.

3.2 Couverture du code

Nous avons utilisé *gcov* pour effectuer la couverture du code.

> "La couverture de code est une mesure utilisée pour décrire le taux de code source testé d'un programme. Ceci permet de mesurer la qualité des tests effectués." (Couverture de code. Wikipédia, L'Encyclopédie Libre, `fr.wikipedia.org/wiki/Couverture_de_code`)

L'outil gcov nous permet donc de vérifier l'efficacité de nos tests. Nous obtenons donc une couverture de nos fichiers testés de respectivement 87,64% pour piece.c et 92,92% pour game.c.

(voir figure 5.3 de l'annexe page 26)

A la lecture des fichiers game.c.gcov et piece.c.gcov (en annexe) nous pouvons nous rendre compte que les lignes non exécutées sont des cas de gestions d'erreurs. Nos tests sont donc correctement implémenté.

(voir game.c.gcov et piece.c.gcov en annexes)

Chapitre 4

Ce qui fonctionne et ne fonctionne pas. Pourquoi ?

L'exécutable permettant de jouer au Rush-Hour et à l'Âne-Rouge fonctionne. Il permet de jouer à plusieurs niveaux de ces jeux. Cela fait preuve d'une implémentation correcte de nos fichiers sources.

Le CMakeLists.txt fonctionne également et permet d'obtenir un exécutable valide sans erreurs ou warnings à la compilation.

Les tests test_piece.c et test_game.c compilent bien et ne détèctent pas d'erreurs dans l'implémentation des fichiers sources testés.

Le solveur marche pour une configuration de Rush-Hour classique mais ne semble pas marcher correctement pour une configuration d'Âne-Rouge.

Chapitre 5

Difficultés rencontrées

Nous avons rencontré quelques difficultés avec le choix d'une bibliothèque graphique. Après une première installation chaotique de la bibliothèque Gtk3 sur l'une de nos machines personnelles entrainant la réinstallation totale du système d'exploitation, nous nous sommes rabattus sur la bibliothèque MVL de part l'importance de la documentation accessible sur internet. Au final, nous n'avons malheureusement pas eu le temps de fournir une interface graphique.

Annexes

FIGURE 5.1 – *capture d'écran de notre page github : github.com/ManonStyle/*

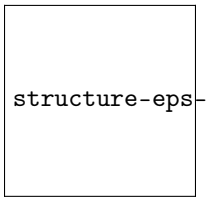

FIGURE 5.2 – *capture d'écran de l'arborescence du projet*

FIGURE 5.3 – *couverture de code des fichiers piece.c et game.c*