Rapport projet de programmation 2016

Groupe A1_2
Gautier DELACOUR
Anthony DELASALLE
Alexis PICHON
Amélie RISI

22 avril 2016

Table des matières

Chapitre 1

Le projet

Ce projet a été réalisé dans le cadre d'une UE. Le but était de créer un jeu Rush Hour ou similaire à celui-ci à l'aide du langage de programmation C.

1.1 Première version

Le projet consistait en l'implémentation de plusieurs modules permettant de jouer à un jeu de type Rush Hour dont le principe est simple : on dispose d'un tableau de jeu sur lequel sont présentes plusieurs pièces de forme rectangulaire. L'une de ces pièces est la pièce maîtresse du jeu. Il faut donc faire sortir celle-ci en déplaçant toutes les pièces qui la gène. Cependant, chaque pièce ne peut se déplacer uniquement dans un seul type de direction, soit horizontalement, soit verticalement. Dans cette première version, il nous était demandé de fournir seulement une interface pour un terminal.

1.2 Deuxième version

Par la suite, des contraintes se sont ajoutées au développement du projet. En effet, on nous a spécifié que notre code devait désormais être valable aussi pour un jeu de type Ane Rouge. Sur ce type de jeu, les pièces peuvent aussi être de forme carrée et n'ont pas forcément de restrictions sur le type de mouvement. De plus, à la différence d'un jeu Rush Hour où la pièce maîtresse doit rejoindre le côté droit du plateau pour gagner, la pièce maîtresse doit ici rejoindre le bas du plateau de jeu pour gagner. Une fois encore le jeu doit être jouable sur une version terminal mais une interface graphique de base peut maintenant être implémentée, mais elle reste optionnelle.

1.3 Le solveur

Pour terminer un solveur est attendu, permettant bien évidemment de résoudre tout niveau de Rush Hour ou de l'Ane Rouge en fonction des paramètres entrés. Le but est de trouver un algorithme pour résoudre tout jeu de la manière la plus rapide et optimisée possible.

Chapitre 2

Notre conception du projet

Lors de notre conception du projet, nous avons eu pour but d'essayer de respecter les réglementations au niveau, par exemple des headers initiaux que l'on avait pas le droit de modifiés et de test unitaires à créer. De plus, nous avions décidés de créer une interface terminal qui soit protéger au niveau des erreurs de saisies d'argument par exemple. Néanmoins, suite à certain problème, nous n'avons pas pu délivré notre travail de la V1 dans les temps impartit avec tous les objectifs remplit. De même, lors de la réalisation de la deuxième version du projet, nous avons aussi eu des problèmes pour créer des exécutables pour l'âne rouge et le rush hour qui soit fonctionnels. Pour finir, notre solveur est complément néant suite à un manque d'organisation dans le groupe.

Pour ce projet, nous avons décidé d'utiliser un dépôt svn et cmake pour compiler notre projet.

2.1 Architecture du projet

Notre projet s'articule autour de la V1 et de la V2. Nous avons séparé les deux versions suite à la modification des headers *piece* et *game* lors du changement des spécifications.

Voici la structure de la V1 :

Build : Dossier qui permet de créer tous les fichiers liés à cmake et la compilation.

Include : Dossier pour tous les headers.

Lib : Dossier où toutes les libs sont après leur création

Src : Dossier où se trouvent tous nos .c. Ce dossier est organisé comme suit :

- Game : Ce dossier décrit toutes foctionnnalité liée à un jeu de Rush Hour, c'est-à-dire les pièces et le game ainsi que les tests unitaires de ces modules.
- Rush Hour : Ce dossier décrit le rush hour, c'est-à-dire il n'y a qu'un main dans le fichiers .c qui crée un jeu de rush hour.
- Exec : Ce dossier décrit l'excution d'un jeu.

Io_interface : Ce dossier décrit l'interface sur le terminal.

2.2 Les modules

Module piece.c

Ce module contient la structure d'une pièce ainsi que toutes les fonctions relatives à ces dernières. C'est ce module qui va nous permettre de créer les pièces de notre jeu, de les modifier (déplacer) ou de les supprimer.

Module game.c

Ce module contient la structure d'un jeu ainsi que toutes les fonctions qui s'y appliquent. Il nous permet de créer le plateau de jeu, d'y placer les pièces dedans et de les déplacer non plus dans le simple cadre d'une pièce comme pour le module précédant mais dans un envirronement de jeu avec d'autres pièces autour et des limites imposées par les dimensions du plateau.

Chapitre 3

Les tests

Chapitre 4

Le solveur

4.1 Première approche

Dans un premier temps, notre approche de la conception du solveur s'est orientée vers l'encadrement du problème. Il s'agit de trouver une méthode de résolution d'un problème dans l'espace et le temps, le jeu étant en deux dimensions et se déroulant en étapes successives. Pour ce faire, il a été indispensable de s'interroger sur les structures de données susceptibles de convenir pour le stockage des tours d'une partie. Les contraintes du jeu imposaient de s'intéresser aux structures de données ordonnées, telles que les listes chaînées ou les arbres. Toutefois, il ne faut pas négliger de prendre en considération que pour un tour donné dans une partie, il existe plusieurs actions possibles. Cela élimine la possibilité d'utiliser des listes chaînées. L'autre information importante dégagée est relative à la recherche d'un plus court chemin : si un tour donné de la partie permet plusieurs choix, il est également vrai que différents choix de mouvements peuvent mener à une même configuration des pièces. Parce que ces chemins peuvent être de taille différente, la structure de données choisie est celle d'une table de hachage à adressage chaîné. L'utilisation d'une table de hachage permet de trier les configurations des pièces indépendamment d'un avancement dans le temps. La structure des clés de hachage correspondant à la position des pièces à une étape donnée :

— un premier chiffre pour la position en X de la pièce
— un second pour la position en Y

Ces coordonnées sont celles de l'angle bas-gauche de la pièce en question Ce choix d'organisation permet un stockage des tours indépendant de nombre de tours écoulés, qui sont stockés dans chaque chaque cellule de la table. Les cellules sont organisées de la façon suivante :

— clé de hachage
— tableau de pointeurs pour les tours suivants possibles

4.2 Deuxième approche

Chapitre 5

Analyse mémoire

5.1 Valgrind

5.2 Couverture de code

Chapitre 6

Organisation

6.1 Répartition des tâches

Première partie du projet pour la V1

La première étape du projet nous demandait d'implémenter les modules **piece.c** et **game.c** sur lesquels seront effectués des test unitaires. Ces modules contiennent les fonctions qui permettent au jeu de fonctionner, un dernier module gérant l'affichage du jeu sur terminal. Il y avait 4 grands axe à développer et nous nous sommes donc répartis les taches de la sorte :

Gautier : implémentation du test unitaire sur game

Anthony : implémentation du module piece

Alexis : implémentation du module d'affichage du jeu

Amélie : implémentation du module game

Par la suite Alexis s'est occupé de structurer notre projet pour un repérage plus facile des fichiers et de faire les Makelists permettant d'obtenir un exécutable.

Suite de projet pour la V2

L'évolution du projet vers la v2 nous demandait de faire en sorte que notre code, qui devait permettre lors de la V1 de jouer à Rush Hour, puisse aussi nous permettre de jouer à l'Ane Rouge. Il a donc fallut revoir les modules **piece.c** et **game.c** ainsi que le test unitaire sur **game.c**. De plus le test unitaire sur **piece.c** de la V1 n'étant plus en vigueur il a aussi fallut implémenter les test unitaire de ce module.

Mise à niveau de module game Amélie

Mise à niveau du module piece Anthony

Implémentation des tests unitaires sur game et piece Equitablement réparti entre Gautier et Anthony

Implémentation du module d'affichage de l'Ane Rouge Alexis

6.1.1 Module piece

Structure d'une pièce

Chaque pièce du jeu est défine par les coordonnées (x,y) de son angle bas-gauche, sa hauteur et sa largeur sur le plateau ainsi que par sa faculté à se déplacer de manère verticale et/ou horizontale. On a donc une structure à six champs qui sont :

- **x** un entier correspondant à la coordonnée de l'angle bas-gauche de la pièce sur l'axe des abscisses
- **y** un entier correspondant à la coordonnée de l'angle bas-gauche de la pièce sur l'axe des ordonnées
- **width** un entier correspondant à la largeur de la pièce
- **height** un entier correspondant à la hauteur de la pièce
- **move_x** un booléen indiquant si la pièce peut etre déplacée horizontalement
- **move_y** un booléen indiquant si la pièce peut etre déplacée verticalement

La fonction *new_piece_rh(int x, int y, bool small, bool horizontal)*

Cette fonction nous permet de créer une pièce de jeu initialisée aux coordonnées (x,y) entrées en paramètres. Les deux booléens *small* et *horizontal* vont nous permettre à eux deux de déterminer les valeurs des champs *width*, *height*, *move_x* et *move_y*. En effet cette fonction n'est valable que dans le cas d'un jeu Rush Hour où les pièces ne peuvent effectuer qu'un seul type de movement et dont la taille (largeur ou hauteur en fonction de son inclinaison) est de 2 ou de 3, le second champs étant à 1. Par exemple une pièce initialisé avec un *small* à *false* et un *horizontal* à *true* aura une largeur de 3 et une hauteur de 2. De plus étant horizontale elle aura *move_x* à *true* et *move_y* à *false*.

```
1  piece new_piece_rh(int x, int y, bool small, bool horizontal)
2  {
3    piece p = malloc(sizeof(*p));
4    if (p==NULL)
5    {
6      fprintf(stderr, "Allocation probleme");
7      exit(EXIT_FAILURE);
8    }
9    p->x = x;
10   p->y = y;
11   if (horizontal)
12   {
13     p->width = 3;
14     p->height = 1;
15     p->move_x = true;
16     p->move_y = false;
17     if (small)
18       p->width -= 1;
19   }
20   else
21   {
22     p->width = 1;
23     p->height = 3;
24     p->move_x = false;
25     p->move_y = true;
```

```
26      if (small)
27        p->height -= 1;
28    }
29    return p;
30  }
```

On se contente ici d'allouer dynamiquement un espace mémoire à cette nouvelle pièce avant de lui attribuer les valeurs souhaitées.

La fonction *new_piece(int x, int y, int width, int height, bool move_x, bool move_y)*

Le principe est le même que la fonction précédante hormis le fait qu'on initialise ici nous-même les valeurs de *width*, *height*, *move_x* et *move_y* afin de convenir aux possibilités offertes par un jeu de même type que l'Ane Rouge. Dans ce jeu les pièces n'ont pas de restrictions de dimension en dehors de celles imposées par le plateau de jeu et peuvent se déplacer à la fois en abscisse et en ordonnée.

La fonction *delete_piece(piece p)*

Simple fonction permettant de supprimer une pièce créée. On se contente de vérifier si la pièce passée en paramètre n'est pas *null* avant de faire un *free* de cette pièce.

La fonction *copy_piece(cpiece src, piece dst)*

Cette fonction prend deux pièce en paramètre et vérifie si elle ne sont pas *null*. Si le test passe on copie les valeurs de la pièce *src* dans la pièce *dst*. Dans le cas contraire un message d'erreur est affiché pour avertir qu'au moins une des pièces entrées en paramètre ne convient pas.

La fonction *move_piece(piece p, dir d, int distance)*

Cette fonction nous permet de décaler les coordonnées (x,y) de la pièce entrée en paramètre de *distance* dans la direction *d* choisi. Si on veut bouger la pièce verticalement on modifie la coordonée *y* sinon la coordonnée *x*. La *distance* est additionnée ou soustraite en fonction de la direction. Bien évidemment on vérifie avant toute modification que les champs de la pièce sont en accord avec le déplacement souhaité.

La fonction *intersect(cpiece p1, cpiece p2)*

Dans cette fonction on test si deux pièces entrées en paramètres se croisent, c'est-à-dire si elles occupent une même place sur la plateau, auquel cas elle retourne *true*. Afin de savoir si il y a effectivement intersection ou pas on commence par tester si les deux pièces ont les mêmes coordonnées (x,y) : si oui la fonction retourne *true* dès maintenant. Sinon on poursuit en comparant les angles des pièces. Si l'angle de la bas-gauche de la première pièce et situé avant l'angle

haut-droit de la seconde pièce et si l'angle haut-droit de la première pièce est aussi situé après l'angle bas-gauche de la seconde pièce alors il y a intersection.

```
1  bool intersect (cpiece p1, cpiece p2)
2  {
3    if (get_x(p1) == get_x(p2) && get_y(p1) == get_y(p2))
4      return true;
5    int x1=get_x(p1);
6    int x2=get_x(p2);
7    int y1=get_y(p1);
8    int y2=get_y(p2);
9    if ((x1 < x2+get_width(p2)) && (x1+get_width(p1) > x2))
10     if ((y1 < y2+get_height(p2)) && (y1+get_height(p1) > y2))
11       return true;
12   return false;
13 }
```

6.1.2 Module game

Structure d'un jeu

Le plateau de jeu est définie par ses dimensions hauteur*largeur et l'ensemble de pièces qui le compose. Sa structure est donc composée des champs *board* un tableau 2D représentant le plateau de jeu, les entiers *width* et *height* correspondants respectivement aux dimensions en largeur et en hauteur du plateau ainsi que du champs *piece* qui va contenir toutes pièces qui seront présentes sur le jeu. On dispose également du champs *nb_piece* pour connaître le nombre exact de pièces présentes en jeu et d'un dernier champs *nb_move* pour savoir combien de mouvement de pièces ont été réalisé depuis le début du jeu. Ce dernier champs est inclus dans la structure car il est propre à chaque jeu.

La fonction *new_game (int width, int height, int nb_pieces, piece *pieces)*

Pour créer un nouveau plateau de jeu on initialise un tableau 2D par allocation dynamique de dimension *width* et *height*. On copie ensuite chaque pièces du tableau de pièces passé en paramètre dans le champs *piece* après s'être assuré qu'elle ne soit pas en dehors du plateau. Une fois ces deux étapes effectuées on place les pièces sur le plateau en attribuant à chaque case du plateau le numéro de la pièce qu'il l'occupe ou -1 si aucune pièce n'est présente.

```
1  game new_game (int width, int height, int nb_pieces, piece *pieces)
2  {
3    if(pieces==NULL || nb_pieces<=0 || pieces[0]==NULL || width<=0 || height<=0)
4        return NULL;
5
6    game new_g = malloc (sizeof (*new_g));
7    if (new_g==NULL)
8        fprintf(stderr,"Problem in the allocation of newGame!!!\n");
9
10   new_g->board = malloc (sizeof(*(new_g->board))*height);
11   if (new_g->board==NULL)
12       fprintf(stderr, "Problem in the allocation of newGame's board (height)\n");
13   for (int j= 0; j<height; ++j)
14   {
15       new_g->board[j] = (int *) malloc (sizeof(*(new_g->board[j]))*width);
16     if (new_g->board[j]==NULL)
```

```
17          fprintf(stderr, "Probleme in the allocation of newGame's board (width)\n");
18      }
19
20      new_g->width = width;
21      new_g->height = height;
22      new_g->nb_move = 0;
23      new_g->nb_piece = 0;
24
25      new_g->piece = malloc (nb_pieces * sizeof(*pieces));
26      if (new_g->piece==NULL)
27          fprintf(stderr,"Problem in allocation of piece in structure\n");
28
29      for (int i=0; i<nb_pieces; ++i)
30      {
31          if (!is_in_grid(new_g, pieces[i]))
32            fprintf(stderr, "Piece %d out of board\n", i);
33
34          new_g->piece[i] = new_piece(get_x(pieces[i]), get_y(pieces[i]), get_width(pieces[i]), get_h
35          new_g->nb_piece += 1;
36      }
37
38      positionning(new_g, (cpiece*)new_g->piece);
39
40      if (new_g->nb_piece != nb_pieces)
41      {
42          delete_game(new_g); // free of all we allocated before
43          new_g = NULL;
44      }
45      return new_g;
46  }
```

La fonction *game new_game_hr (int nb_piece, piece *piece)*

Dans ce cas, on créer un jeu dont les conditions sont propres à Rush Hour. Il s'agit d'un simple appel à la fonction *new_game* avec les dimensions d'un plateau de Rush Hour.

La fonction *void delete_game (game g)*

Elle nous permet de supprimer un jeu en libérant l'espace mémoire qu'il occupait. On supprime chaque pièce du champs *piece* puis la plateau lui-même.

La fonction *play_move(game g, int piece_num, dir d, int distance)*

Cette fonction tente de déplacer la pièce de numéro *piece_num* de *distance* cases vers la direction *d*. Si le mouvement est valide, c'est-à-dire qu'à l'issue de son déplacement elle est toujours dans la plateau, qu'elle ne vient pas occuper une case déja prise et que la direction de mouvement lui est permise, alors la pièce est déplacée et le nombre de mouvement effectué est incrémenté. Dans le cas où une des conditions ne serait pas valide, la fonction renvoie *false* et la pièce n'est pas bougée. On créé donc une autre pièce initialisé avec les valeurs que prendrait la pièce *piece_num* si elle était déplacée maisqui ne sera pas placée sur le plateau. Elle nous sert à vérifier que la position qu'occuperait la pièce *piece_num* après déplacement est valide ou non. Si elle est valide on modifie les valeurs de la pièce *piece_num* et on actualise la numérotation des cases du tableau.

```
1  bool play_move(game g, int piece_num, dir d, int distance)
2  {
3       if(piece_num>=g->nb_piece || piece_num<0)
4           return false;
5
6       int x = get_x(game_piece((cgame)g, piece_num));
7       int y = get_y(game_piece((cgame)g, piece_num));
8       int p_height = get_height(game_piece((cgame)g, piece_num));
9       int p_width = get_width(game_piece((cgame)g, piece_num));
10
11      if(can_move_y((cpiece)g->piece[piece_num]) && (d==UP || d==DOWN))
12      {
13          int new_h = 0;
14          if (d==DOWN)
15          {
16              new_h = y - distance;
17          }
18          else
19          {
20              new_h = y + distance;
21          }
22          piece tmp_p = new_piece (x, new_h, p_width, p_height, false, true);
23          if (is_in_grid((cgame)g, (cpiece)tmp_p))
24          {
25              for (int i=0; i<game_nb_pieces(g); ++i)
26              {
27                  if (piece_num!=i && intersect(tmp_p, g->piece[i]))
28                      return false;
29              }
30              move_piece(g->piece[piece_num],d,distance);
31              g->nb_move += distance;
32              positionning(g, (cpiece*)g->piece);
33              return true;
34          }
35      }
36      if(can_move_x((cpiece)g->piece[piece_num]) && (d==LEFT || d==RIGHT))
37      {
38          int new_w = 0;
39          if (d==LEFT)
40          {
41              new_w = x - distance;
42          }
43          else
44          {
45              new_w = x + distance;
46          }
47          piece tmp_p = new_piece (new_w, y, p_width, p_height, true, false);
48          if (is_in_grid((cgame)g, (cpiece)tmp_p))
49          {
50              for (int i=0; i<game_nb_pieces(g); ++i)
51              {
52                  if (piece_num!=i && intersect(tmp_p, g->piece[i]))
53                      return false;
54              }
55              move_piece(g->piece[piece_num],d,distance);
56              g->nb_move += distance;
57              positionning(g, (cpiece*)g->piece);
58              return true;
59          }
60      }
61      return false;
62  }
```

6.1.3 Module affichage

Gestion des erreurs

En parallèle du découpage effectué, la gestion des erreurs s'est voulue effectuée de façon fluide. Ainsi, un module "manage_error" regroupe des informations caractéristiques des erreurs (usage de macros pour différents types d'erreurs, de façon la plus précise possible) et une encapsulation pour l'affichage de messages récapitulatifs des erreurs en question lors de leur apparition. De plus, afin de minimiser l'exécution de sections potentiellement critiques du programme, lorsque cela a été possible les branchements conditionnels ont été effectués afin de prévenir l'exécution de code qui serait compromise par une erreur.

Point d'entrée

Au lancement du programme, l'unité fonctionnelle intitulée "play" prend en charge l'initialisation du programme de la façon suivante :

- branchement avec le module "data" pour initialiser les champs de la structure qui conservera les données utiles au fil de l'exécution
- appel du parseur des arguments passés en entrée au programme ("parse_argv")
- le parseur effectue un branchement avec le module "opt" dédié à la détection des paramètres en entrée

Flux d'exécution

En suite du point d'entrée, la gestion de la partie est transmise à l'unité fonctionnelle "move" qui se chargera de récupérer les instructions utilisateurs, d'effectuer les branchements avec le module "game" et de renvoyer à l'utilisateur la confirmation des modifications par le biais de l'affichage, ce passant par le module "term_mode", qui prend en charge l'affichage sur terminal en fournissant des outils de lecture/écriture sur celui-ci.

6.1.4 Le solveur

6.1.5 Les structures de données

6.1.6 Les tests

6.2 Répartition dans le temps et en espace

Chapitre 7

Les difficultés rencontrées

7.1 Au niveau de la V1

7.2 Au niveau de la V2

7.3 Au niveau du solveur

Chapitre 8

Pour aller plus loin

8.1 Analyse des problèmes rencontrés

8.2 Les améliorations à apporter au projet

8.3 Ce que nous à apporter le projet