# Rapport du Projet de Programmation

Halnaut Adrien, Marty Yoan, Ordonez Romain

27 avril 2016

**Résumé**

Ce rapport retrace le parcours effectué durant la réalisation du Projet de Programmation, dans le cadre de l'Unité d'Enseignement « Environnement de Développement et Projet Programmation ».

# Table des matières

# Partie 1

# Introduction

Le but de ce projet est de vérifier nos compétences en langage C, ainsi que notre utilisation de systèmes de gestions de versions (svn ou git, en l'occurrence nous avons utilisé git), et de notre environnement de développement, à savoir éditeur de texte et moyens de compilation (Make, CMake).
Afin de mettre en valeur ces compétences, le sujet de cette année 2015-2016 est basé sur les jeux de puzzle à mouvements de pièces, comme le taquin. Les formes utilisées dans ce projet sont celles de Rush Hour en premier lieu, qui était la base du projet, lequel a été commencé en version texte sur la console du terminal.
Après un certain temps, notre cahier des charges a changé, nous demandant plus de flexibilité dans notre code, afin d'implémenter d'autres jeux du même genre. Nous avons donc eu à implémenter un autre jeu, l'Âne Rouge.
Plus tard, après que toutes les modifications soient implémentées, il nous a été demandé de réaliser un solveur, capable de nous donner le nombre de coups minimal à effectuer pour un niveau donné, ainsi qu'une interface graphique réalisée à l'aide d'une bibliothèque graphique (SDL, ou sa couche simplifiée MLV).
Nous verrons dans ce rapport un déroulement chronologique des événements de ce projet, de sa racine (Rush Hour en console) à sa forme finale (Un jeu flexible, possédant un solveur, et une interface graphique).
Vous pouvez retrouver notre repositoire à l'adresse `https://github.com/Echoffee/puzzle-game-TEA`.

# Partie 2

# Développement - Rush Hour

La première partie du projet consistait à implémenter une version console du jeu Rush Hour. Nous avions à notre disposition 2 headers : `game.h`, et `piece.h`, qu'il nous fallait implémenter dans le but d'obtenir une librairie, `libgame.a`, ainsi qu'une batterie de tests pour le module `piece.c`, et il nous fallait réaliser celle de `game.c`.

Après avoir réalisé cette librairie, nous avons commencé à travailler sur l'interface. Nous avons donc produit un ensemble de fonctions utilitaires, répertoriées dans `utility.h`, utilisées un peu partout dans les fichiers principaux `game.c` et `piece.c`, ainsi que dans les fichiers permettant la formation de l'interface, `function_interface.c` accompagné de son header `function_interface.h` pour la modularité et l'utilisation de ses fonctions dans `interface_txt.c`, qui contient l'interface en elle même.

Pour rendre cette interface plus lisible, et agréable à l'œil, nous avons utilisé le système de coloration ANSI, qui permet notamment de modifier la couleur de fond à un endroit donné, ou la couleur de la police du terminal, toujours à un endroit donné. Pour alléger le travail de création de niveau, dans l'optique d'une génération future par identificateur, nous avions eu l'idée de créer le jeu par un `id`. Cependant, nous allions avoir besoin d'utiliser des fonctions, certes fournies par le header `string.h`, mais que nous avons trouvé plus gratifiant de réaliser par nous mêmes. Ainsi, les fonctions `strcmp(char* a, char* b)`, `toLower(char s)` et `isDigit(char s)` ont été rééditées par nos soins, afin de correspondre à ce qu'on voulait, à savoir les fonctions `str_equal()`, `isNumber()` et `toLower()` de `function_interface.h`, utilisées dans le but de lire une entrée, ou de lire un id, voire de vérifier que les id à comparer sont égaux...

Pour lire une entrée, nous avons utilisé la fonction `fgets()`, de `stdio.h`, associée à nos fonctions `str_equal()` et `checkFormat`, qui vérifient si l'entrée est une commande spéciale (menu interactif, quitter...) ou bien un mouvement (nombre). Voici le code de la lecture d'entrée :

Code 2.1 – interface_txt.c

```c
void input_player(game g, char* id)
{
   char input[7] = {0, 0, 0, 0, 0, 0, 0};
   fgets(input, sizeof(input), stdin);
   toLower(input);
   bool correct = false; // booléen vérifiant si l'input est correct.
   if (str_equal(input, "help\n"))
   {
      //Affiche le menu d'aide interactif
   }

   if (str_equal(input, "exit\n"))
   {
      // ferme le jeu.
   }
   if (str_equal(input, "save\n"))
   {
      //sauvegarde dans un fichier save.txt le niveau actuel en générant son ID.
   }
   if (str_equal(input, "load\n"))
   {
      //exécute un chargement de niveau, en demandant si il s'agit d'un numéro de niveau ou
           de la sauvegarde

   }

   if (isNumber(input[0], g -> nb_pieces - 1))
   {
      //réalise un mouvement
   }
   else
   {
      if (!correct)
      {
         printf("Incorrect input. Type 'help' for more informations.\n");
         ignoreOverflow(input, 6);
      }
   }
   getIdFromGame(g, id);
}
```

Nous avons décidé, pour cette lecture, de lire l'entrée dans un buffer de taille 7. Si l'entrée est incorrecte, c'est à dire qu'elle dépasse le buffer ou ne correspond à aucune commande, on renvoie une erreur sur la sortie standard, et une nouvelle entrée est demandé au joueur. L'utilisation de `toLower()` permet de faire une entrée telle que HELP au lieu de help, par exemple. La fonction `ignoreOverflow()` permet d'ignorer toute tentative de bourrage du buffer du genre « AAAAAAAA ».

Enfin, vient la partie `main()`, permettant l'exécution de notre programme. Elle se trouve dans le fichier `rush-hour.c`. Dans celui-ci, on remarque que les niveaux sont définis par des `char*`, c'est à dire par leur `id`. Il dispose d'une fonction `loadTheGame()`, qui charge l'interface et le jeu à partir de l'`id` qu'on lui envoie en paramètre.

Côté compilation, nous avons utilisé CMake : Un fichier à la racine du projet, qui contient le nom du projet, référence les répertoires où se trouvent les fichiers source et header, ainsi que les fonctions activant le paramètre `ctest` permettant d'exécuter les batteries de tests sans avoir à les chercher une par une ; et un fichier dans le sous-dossier `src`, qui définit les dépendances de l'exécutable `rush-hour`, de la librairie `libgame.a`, l'endroit où se trouvera l'exécutable, et

qui crée un fichier texte contenant les `id` de trois niveaux de base afin d'utiliser la commande `load <niveau>` en jeu.

Enfin, niveau langage d'écriture, nous avons utilisé l'Anglais pour le code, puisqu'il était présenté comme tel dans les *headers* fournis, mais avons utilisé le Français en tant que langage de commentaire, pour faciliter l'expression de nos idées en utilisant les bons termes techniques. Notre notation respecte un code uniforme, les fonctions ont un nom `nomFonction` et les variables ont un nom `nom_variable`.

# Partie 3

# Développement - Extension

Après avoir rendu la version du jeu Rush Hour sous console, notre cahier des charges s'est vu être modifié. Le but principal de ces modifications était de rendre notre code le plus modulaire possible afin de pouvoir réaliser plusieurs jeux différents mais avec le même « gameplay ». Ainsi, il nous a été demandé de développer un autre jeu, l'Ane Rouge, afin d'appliquer ces modifications.

Suite à notre nouveau cahier des charges, nous avons dû revoir certaines de nos fonctions ainsi que notre système d'ID car l'ancien n'était pas adaptable aux nouvelles spécifications concernant les parties générées. Ainsi, nous avons découpé notre nouveau système en trois. La première contient le nombre de pièces du jeu suivi de la lettre `n` , la deuxième nous permet de savoir les dimensions du plateau, la largeur suivie de la lettre `x` puis la hauteur, et la troisième partie pour stocker les pièces. Cette troisième partie est extensible, dans le sens où elle dépend uniquement du nombre de pièces, il faut mettre la lettre `p` pour signaler le début d'une pièce, suivie d'un chiffre permettant de savoir si les déplacements horizontaux et verticaux de la pièce sont autorisés, 0 si la pièce ne peut pas bouger, 1 si la pièce peut bouger verticalement, 2 si elle peut bouger horizontalement et 3 si elle peut se déplacer dans les deux sens. Ensuite, on ajoute la lettre `w` suivie de sa largeur, puis la lettre `h` suivie de sa hauteur, et enfin, la lettre `x` suivie de son abscisse puis la lettre `y` suivie de son ordonnée. Et on répète ce schéma pour chaque pièce que l'on souhaite ajouter. La figure 3.1 est une illustration du notre système d'ID afin de mieux comprendre son fonctionnement.

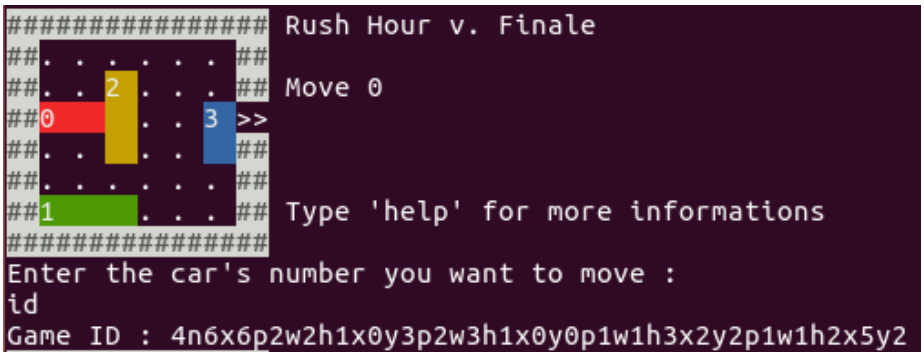


FIGURE 3.1 – Un exemple de partie avec son ID.

Ensuite, nous nous sommes demandés si nous devions intégrer toutes les fonctions concernant exclusivement Rush hour, et l'Ane Rouge, dans une même archive, ce qui aurait impliqué d'avoir des fonctions inutilisées lors de l'exécution d'un des deux programmes. Nous n'avons pas voulu faire cela, ainsi nous avons décidé de mettre les fonctions spécifiques à Rush Hour et à l'Ane Rouge dans un fichier séparé, respectivement `game_rh.c` et `game_ar.c` et de créer une archive pour chaque jeu.

Par la suite, nous avons créé une fonction écrivant dans un fichier `config.ini` le nom du jeu en cours d'exécution afin de pouvoir afficher le nom du jeu sur notre interface console, ainsi que de savoir quelle condition de victoire le programme devait vérifier, cela nous a permis d'éviter de trop grosses modifications sur le code de notre interface console suite au nouveau cahier des charges.

# Partie 4

# Développement - Solveur

Le solveur a eu le droit à plusieurs formes d'algorithme avant d'être plutôt stable et fonctionnel, bien qu'imparfait.

## 4.1 Premier solveur - Hasard

Plutôt sceptique par rapport au temps d'exécution d'un potentiel solveur brute-force, nous avons tenté dans un premier temps de faire un solveur bougeant une pièce sélectionnée au hasard d'une case dans une direction au hasard (mais valide). Avec surprise, nous avons remarqué que les niveaux basiques demandant qu'une trentaine de coup maximum étaient résolus en peu de temps (en ayant des nombres de coups complètement ridicules, atteignant les 1800 coups pour un puzzle résoluble en 10).

Confiant de cette rapidité de calcul pour un résultat loin d'être optimal, nous avons essayé d'exploiter ces données.

## 4.2 Second solveur - Analyse de fréquences

Il est évident que si l'on lance plusieurs fois notre premier algorithme, on obtiendra plusieurs solutions de résolution du puzzle, avec des nombres de coups plus ou moins grands. Nous sommes partis du principe que, si dans notre exemple le nombre de coup minimal est 10, alors on peut tracer un graphe de 10 nœuds, contenant chacun une disposition du plateau représentant ces 10 mouvements. Ainsi le premier nœud sera la configuration initiale et le dernier la configuration finale avec la pièce #0 à l'arrivée. Ainsi, il est donc obligatoire que parmi les nombreux mouvements exécutés par le solveur, il s'y trouverait forcément ces 10 configurations (voir 4.1). Ainsi, si l'algorithme tourne plusieurs fois, on retrouvera des configurations semblables entre chaque parcours, ses mouvements clés notamment. On peut donc relever ses mouvements les plus fréquents et donc obtenir le parcours optimal.
Après quelques tests, nous avons remarqué qu'il nous fallait un trop grand nombre de parcours afin d'obtenir des résultats exploitables, et que nous perdions grandement en performances avant d'avoir des résultats convenables (toujours sur le puzzle de 10 coups, lancer 100 fois l'algorithme ne permettait qu'ob-

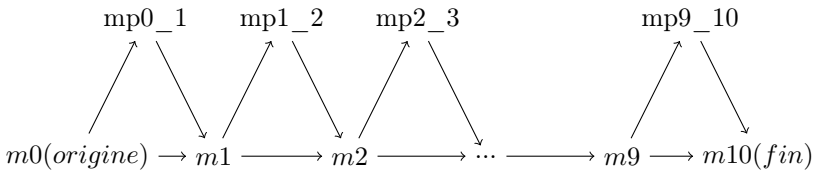


FIGURE 4.1 – Graphe de mouvements. $mn$ est un mouvement du parcours optimal et $mpn\_n+1$ est un ensemble de mouvements parasites.

tenir 2 mouvements sur 10, soit de descendre des 1800 coups à un peu plus de 780). Nous avons donc abandonné l'idée d'un solveur qui fonctionne par le hasard.

## 4.3 Troisième solveur - BFS (Première tentative)

Le hasard étant ainsi écarté des pistes de réflexion et le brute-force toujours à éviter, il fallait trouver une autre approche de résolution de puzzle. Nous avons ainsi cherché comment ne pas tomber dans le brute-force. Après avoir modélisé le déroulement d'un jeu de puzzle sous forme d'un graphe avec chaque noeud étant une configuration du plateau et les arêtes des mouvements possibles pour chaque nœud, nous avons remarqué que faire un algorithme brute-force revenait à faire un algorithme qui parcourait ce graphe en profondeur, c'est-à-dire aller jusqu'au bout d'une branche, vérifier si le puzzle est résolu, et si non, on remonte au nœud précédent et tester un autre parcours, et si impossible, on remonte encore, etc... On obtient un algorithme récursif. En pseudo-code, cela donnerait :

Code 4.1 – Algorithme Solveur 3

```
node[] getPossibleMoves(config c)
{
   //Teste tous les mouvements possibles et en renvoie une liste de noeuds enfants
}

bool solver(node n)
{
   if (game_over(n.config))
     return true;
    n.fils = getPossibleMoves(n.config);
    for (int i = 0; i < n.fils.length; i++)
      if (solver(n.fils[i]))
         return true;
    return false;
}
```

Le vrai code C étant un peu plus complexe, il a cependant été réalisé assez rapidement et a donné des résultats plutôt convenables sur des niveaux simples mais était catastrophique sur des niveaux légèrement plus complexes. En effet, nous avons remarqué que dans le graphe virtuel produit par l'algorithme, on obtenait de nombreuses configurations identiques sur des branches différentes. Ayant tout de suite remarqué que ce que l'on venait de coder était totalement ce que l'on voulait éviter (un brute-force, entre autres), nous avons voulu corriger notre erreur en ajoutant un tableau de configuration que l'on rencontrait, et à vérifier si nous avions déjà rencontré un cas de figure, si c'était le cas, il ne servait à rien de continuer à parcourir la branche. Lors du développement de ce correctif, nous avons réussi à donner naissance à un *Heisenbug*[2] (des heures à

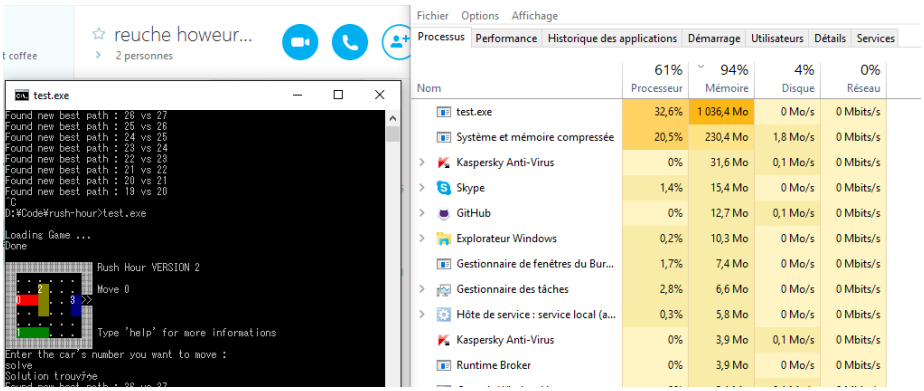


FIGURE 4.2 – Un truc qui ne va pas bien probablement.

passer en debug sur `gdb` pourront le prouver) et ont ainsi complètement perturbé le développement du solveur #3, sans compter les millions (sans exagérer, voir 4.2) d'erreurs que `valgrind` a réussi à nous trouver en passant plus de 5 minutes sur notre pauvre niveau à 10 mouvements, en faisant un `SegFault` en exécution normale mais non en `gdb` (sûrement à cause de la virtualisation de la mémoire).

Afin de préserver la santé mentale de l'équipe, nous avons décidé de repartir de zéro sur le solveur, en gardant à l'esprit de ne pas repasser par des configurations déjà rencontrées.

## 4.4 Quatrième Solveur - BFS (Seconde tentative)

Après nous être informés sur les possibles implémentation d'un algorithme BFS[1] (*Breadth-First-Search*, *Parcours en largeur* en français), nous nous sommes orientés sur l'implémentation d'un tel algorithme en passant par une file. La file sera utilisée telle que décrite en figure 4.3. Pour éviter de passer par les mêmes configurations, nous avons décidé de créer une sorte de tableau pour contenir les configurations déjà rencontrées, sauf que pour ne pas risquer de nouveaux problèmes au niveau de la mémoire, nous avons utilisé une liste simplement chaînée. Le développement du solveur s'est avéré être plus simple et mieux organisé que précédemment, et nous a donné de très bons résultats, à savoir 0.025s pour notre niveau à 10 coups, et s'est plutôt bien débrouillé au concours (aucun plantage, et a fini premier sur une configuration).

Nous aurions pu l'améliorer en optimisant par exemple la vérification de cas déjà rencontrés ou alors le calcul des mouvements possibles, mais nous avons préféré nous appuyer sur le développement de l'interface graphique laissé jusqu'à présent aux mains d'un seul membre du groupe.

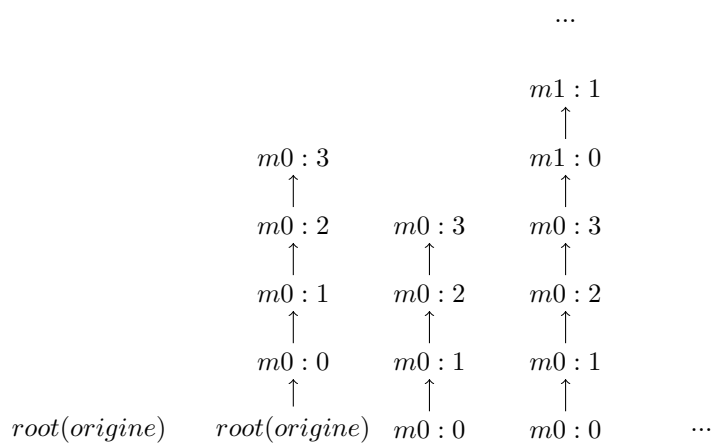


FIGURE 4.3 – Représentation de la file : le noeud *root* crée des objets dans la file par rapport à sa configuration (ses mouvements possibles) puis sort de la file, et on continue pour l'objet suivant, etc...

## 4.5 Explication du code du solveur

### 4.5.1 Les Structures

Notre code utilise 3 structures pour fonctionner :

1. `nodeQueue` qui permet de créer la file, chaque objet possède un pointeur vers l'objet suivant de la file, et une `map`.
2. `map` qui permet de tracer le déroulement des mouvements entrés par le solveur, et possède une structure `game`, donc une configuration du plateau. Dans la version rendue, tout ce qui est traçage a été laissé en commentaire puisqu'il n'était pas demandé de telles fonctionnalités lors de son rendu.
3. `list` qui n'est qu'une liste simplement chaînée.

### 4.5.2 Code

Code 4.2 – isCleared

```
bool isCleared(nodeQueue n, bool isRH)
{
   bool cleared;
   if (isRH)
      cleared = game_over_hr(n->m->g);
   else
      cleared = game_over_ar(n->m->g);
   return cleared;
}
```

Fonction simple qui vérifie si la configuration du plateau d'un noeud de la file termine le jeu. On note l'utilisation d'un `bool isRH` qui limite l'utilisation du solveur sur l'Ane Rouge et Rush-Hour.

Code 4.3 – compareMap

```
bool compareMap(map m1, map m2)
```

```
{

   for (int i = 0; i < game_nb_pieces(m1->g); i++)
   {
      if ((get_x(game_piece(m1->g, i)) != get_x(game_piece(m2->g, i))) ||
            (get_y(game_piece(m1->g, i)) != get_y(game_piece(m2->g, i))))
         return false;
   }
   return true;
}
```

Cette fonction vérifie si 2 `map` ont des configurations identiques (sans compter le nombre de mouvements). Afin de minimiser le nombre d'opérations effectuées et de gagner du temps dans la résolution, la fonction s'arrête dès que 2 pièces n'ont pas les mêmes coordonnées et renvoie `false`.

Code 4.4 – checkMapExistence

```
map checkMapExistence(map m, list origin)
{
   if (compareMap(m, origin->m))
   {
      delete_game(m->g);
      free(m);
      return NULL;
   }
   if (origin->next == NULL)
   {
      newListItem(m, origin);
      return m;
   }
   return checkMapExistence(m, origin->next);
}
```

Cette fonction vérifie si la `map` passée en paramètre a déjà été rencontrée. Si elle l'a été, on *free* la nouvelle `map` et on retourne `NULL`. Sinon, on l'ajoute à la liste de configurations déjà rencontrées. Au passage, `origin->next` stocke dans la nouvelle `map` à partir de quelle configuration on est arrivé à cet état-là, et donc permet le traçage à la fin de la résolution du puzzle.

Code 4.5 – fillQueue

```
void fillQueue(nodeQueue currentNode, nodeQueue queueTop, map previousState, list listMap,
     bool* cleared, bool isRH)
{
   for (int p = 0; p < game_nb_pieces(currentNode->m->g); p++)
   {
        for (int d = 0; d < 4; d++)
        {
           map r = createNewState(previousState, p, d, 1);
           if (r != NULL)
           {
              r = checkMapExistence(r, listMap);
              if (r != NULL)
              {
                 queueTop = newQueueItem(r, queueTop);
                 if (isCleared(queueTop, isRH))
                 {
                    currentNode = queueTop;
                    *cleared = true;
                    return;
                 }
              }
           }
        }
   }
}
```

Cette fonction est celle qui attribue des objets supplémentaires à la file. Les `for` sont imbriqués pour tester toutes les directions possibles. `createNewState`

crée une `map` et renvoie `NULL` si le mouvement est impossible. Afin d'éviter de passer plus de temps à tester des mouvements alors que nous obtenons une configuration gagnante, on vérifie sur le coup grâce à `isCleared` et si c'est le cas, on arrête tout.

Code 4.6 – solve

```
void solve(game g, bool rh, mode cmd)
{
   map origMap = newMap(g, NULL);
   list listMap = newListItem(origMap, NULL);
   nodeQueue root = newQueueItem(origMap, NULL);
   nodeQueue currentNode = root;
   nodeQueue top = root;
   bool cleared = false;
   while (!cleared && currentNode)
   {
      top = getTop(top);
      fillQueue(currentNode, top, currentNode->m, listMap, &cleared, rh);
      if (!cleared){
         currentNode = currentNode->next;
      }
   }
game_nb_moves(currentNode->m->g), temps);

game_nb_moves(top->m->g), temps);
   int nbFinal = cleared?game_nb_moves(top->next->m->g):-1;
   map mh;
   switch(cmd)
   {
      case OPTI_MOVES:
      printf("%d\n", nbFinal);break;

      case HINT:
         mh = top->next->m;
         while (mh->from != origMap)
            mh = mh->from;
         drawInterface(mh->g, "HINT");break;
   }
   nodeQueue tmp_node = root;
   while(tmp_node != NULL){
      currentNode = tmp_node;
      tmp_node = currentNode -> next;
      if(currentNode -> m -> g != NULL){
         delete_game(currentNode -> m -> g);
         if (currentNode -> m != NULL)
            free(currentNode -> m);
      }
      free(currentNode);
   }
   list tmp_list = listMap;
   while(tmp_list != NULL){
      listMap = tmp_list;
      tmp_list= listMap -> next;
      free(listMap);
   }
}
```

La fonction d'origine est celle-ci. Les premières lignes servent à initialiser les différentes structures, et le tout se déroule dans la première boucle `while`. Après avoir trouvé les mouvements possibles, on passe à l'objet suivant dans la file. Le nombre de mouvement minimal est stocké dans `nbFinal`. Le `switch` permet de lier les fonctionnalités du solveur avec l'interface texte du jeu (commandes `solve` et `hint`). On libère *tout* l'espace alloué au programme à la fin de la fonction.

Code 4.7 – trace

```
void trace(nodeQueue final)
{
   while (final->m ->from)
   {
```

```
    map tmp = final->m;
    drawInterface(final->m->g, "");
    delete_game(final->m->g);
    final->m = final->m->from;
    free(tmp);
  }
}
```

Cette fonction permet de tracer les mouvement effectués par le solveur et parcourant les `from` des structures `map`.

# Partie 5

# Développement - Interface graphique

Après avoir fini le développement du projet en version texte (console), nous devions terminer par la création d'une interface graphique de celui-ci.

Nous avons eu le choix entre deux librairies afin de réaliser notre interface graphique, MLV et SDL. Nous avons choisi de travailler sous SDL 1.2 car nous avons une bonne connaissance du langage C et que nous trouvions intéressant de se familiariser avec cette librairie. Étant donné notre manque de connaissance sur l'utilisation de SDL, nous avons appris grâce à des tutoriels en ligne, et principalement sur ceux du site OpenClassrooms[3] qui détaille la SDL en version 1.2 et c'est en partie pour cela que nous n'avons pas travaillé sous la version 2.0, car le manque de tutoriel et d'exemple ne nous aurait que ralenti dans le développement de l'interface.

Sur cette partie du projet, nous avons pu intégrer plusieurs aspects de la SDL comme l'intégration de sons grâce à `SDL_Mixer`, l'intégration de texte grâce à `SDL_ttf` et l'ajout d'image avec `SDL_Image`. Ayant travaillé au deuxième semestre sur un projet en Python sous `tkinter`, notre principale remarque fut de constater le manque de gestion de bouton sous SDL, ainsi nous avons développé une structure « `button_s` »ainsi que trois fonctions, une permettant de créer une structure « `button_s` », une autre permettant de savoir si des coordonnées $(x; y)$ se trouvaient bien sur la surface d'un bouton, et une dernière pour libérer l'allocation du bouton. Cette structure est très simple, elle est composé de deux champs représentant les coordonnées $(x; y)$ du coin supérieur gauche du bouton, et de deux autres champs correspondant à la largeur et la hauteur du bouton.

L'interface commence donc par une fenêtre demandant le choix du jeu, une fois le jeu sélectionné, le jeu apparaît avec un compteur sur le nombre de mouvement réalisé, et deux boutons permettant de réinitialiser la partie actuelle et l'autre pour afficher une aide concernant le « gameplay ».
Nous avons fait le choix de ne pas intégrer notre solveur à l'interface graphique, car ne donnant pas un résultat optimal en un laps de temps court, et ne connaissant pas encore de moyen pour faire du « multi-threading », cela aurait figé le

programme pendant plusieurs secondes, voire minutes. Nous avons voulu rester très simple dans la création de nos fenêtres, afin de faciliter l'utilisation de ce programme.

# Partie 6

# Extension - Ajouter de nouveaux jeux

Ici nous allons expliquer comment à partir de notre code ajouter de nouveaux jeux du même style que ceux déjà proposés au-dessus. Tout d'abord, il faudra forcément modifier un peu de code, ou du moins en rajouter.

Premièrement, il faudra définir une condition de victoire dans le fichier `game.c`, un `game_over_VotreJeu`, puis créer un fichier contenant votre fonction `main()`, s'inspirant grandement des autres. Vous aurez simplement à changer la condition de la boucle dans la fonction `loadTheGame()` , écrire le nom de votre jeu à l'aide de la fonction `initFileConfig()` au début de votre `main()`. Vous devrez ajouter vos niveaux dans un fichier `vosParties.txt`dans le répertoire `bin` et spécifier dans votre `main()` le nom de votre fichier texte.

Par la suite, vous devrez modifier les *headers* afin d'intégrer votre nouvelle condition de victoire, voire vos nouvelles fonctions. Pour compiler, un simple ajout de quelques lignes dans les `CMakeLists.txt`.

Voila, normalement après ces quelques étapes et quelques dizaines de minutes, vous devriez pouvoir créer un nouveau jeu sur les bases d'un Rush Hour ou sur celles de l'Ane Rouge.

# Partie 7

# Révision - Git

Au début de notre projet, nous avons pu voir plusieurs façon de gérer un projet. Tout d'abord, nous avons vu qu'il existe une palette riche en système de gestion de version, et deux de ces systèmes sont ressortis durant nos séances de TP, SVN et Git.

Nous avons pu découvrir brièvement SVN durant nos cours, mais nous avons opté pour Git, car nous trouvions son fonctionnement plus clair, malgré les quelques complications pour s'adapter à ce système en début de projet, nous avons finalement su nous approprier convenablement Git, afin d'utiliser une grande partie de ses ressources proposées pour notre projet. Nous avons décidé de passer par une plate-forme via le site *GitHub.com*, car nous trouvions son principe sympathique et facile d'accès, en plus de proposer de nombreux outils pour gérer au mieux notre projet.

Au début du développement du jeu Rush Hour, nous nous sommes répartis les tâches afin d'éviter que plusieurs personnes du groupe travaillent sur la même partie du code, pour cela, nous avons décidé de créer des branches à plusieurs moment du projet, puis une fois le développement de la branche fini, la réintégrer proprement à la branche principale. Ceci nous a permit d'être le plus efficace possible durant le développement de notre projet, et de nous familiariser avec ce système de gestion de version.

# Bibliographie

[1] Un article Wikipedia sur la méthode Breadth-first-search *la méthode Breadth-First-Search :*. https ://en.wikipedia.org/wiki/Breadth-first_search

[2] Un article Wikipedia détaillant ce qu'est le Heinsenbug *l'explication sur l'Heinsenbug :* https ://en.wikipedia.org/wiki/

[3] Un lien vers le principal tutoriel utilisé pour la création de l'interface graphique sous SDL 1.2 *Le lien vers le tutoriel du site OpenClassrooms* https ://openclassrooms.com/courses/apprenez-a-programmer-en-c/installation-de-la-sdl