Rapport de Projet Programmation
2048

Thibaut Parpaite - Aurore Seegers - Damien Le Garrec

Avril 2015

Introduction

Ce document a pour but de décrire le déroulement de notre projet de programmation en langage C au cours de notre semestre 4 de licence informatique.

Le but de ce projet était dans un premier temps de fournir une application permettant de jouer à 2048 (une bibliothèque libgrid.a ainsi qu'un exécutable). Ce jeu se joue sur une grille de 4x4 cases avec des tuiles de valeurs variées et toujours en puissance de 2. On utilise les flèches pour déplacer les tuiles dans la grille selon 4 directions : la gauche, la droite, le haut ou le bas. Lorsque deux tuiles de même valeurs entrent en "collision", celles-ci fusionnent en une nouvelle valeur égale au double de la valeur initiale. Le but du jeu est d'arriver à construire une brique de valeur la plus grande possible (2048, 4096, 8192, etc.). La partie est terminée si la grille est remplie et que aucun mouvement n'est possible.

Dans une deuxième partie, chaque groupe devait proposer un exécutable permettant de jouer à 2048 en mode graphique (grâce à la bibliothèque SDL) ainsi que deux stratégies permettant de gagner le plus souvent possible :

- une stratégie rapide (capable de jouer une partie en moins de 10 secondes)
- une stratégie plus gourmande (capable de jouer une partie en moins de 2 minutes)

L'objectif de ce rapport est de décrire la démarche adoptée durant la réalisation du projet. Nous parlerons tout d'abord du travail effectué, puis nous nous pencherons sur un problème technique rencontré. Ensuite, nous donnerons des exemples de factorisation de code et nous terminerons avec une partie dédiée aux techniques qui nous ont permis d'élaborer les stratégies.

Table des matières

1 Travail effectué

1.1 L'implémentation du jeu 2048

Le premier objectif a été d'implémenter la grille de jeu avec pour taille la variable GRID_SIDE. Dans notre cas, la variable est fixée à 4. La grille est remplie de tuiles qui sont initialisées à 0. Les tuiles sont de type "unsigned int" et pour chaque tuile le nombre affiché est 2^n où n est la valeur de la tuile. On a aussi défini un nouveau type dir pour les directions (haut, bas, droite, gauche).

On a écrit les fonctions :
- new_grid() : on alloue de la mémoire pour la structure de la grille et on initialise toutes les tuiles ainsi que le score à 0
- delete_grid (grid g) : on libère la mémoire allouée pour la grille.
- copy_grid (grid src, grid dst) : on fait une copie de la structure
- grid_score (grid g) : on retourne le score
- get_tile (grid g, int x, int y) : on retourne la la tuile qui a pour coordonnées x et y
- set_tile (grid g, int x, int y, tile t) : on modifie met la tuile t aux coordonnées [x][y]
- can_move (grid g, dir d) : on vérifie pour chaque direction si un mouvement est possible, si c'est le cas alors on retourne vrai, sinon on retourne faux. Pour cette fonctions on a factorisé le code (voir chapitre 3).
- game_over (grid g) : on perd le jeu si aucun mouvement n'est possible dans les quatre directions
- void victory (int i) : on a créé une variable qui correspond à la tuile qui doit être atteint pour gagner
- do_move (grid g, dir d) : effectue un mouvement vers la direction choisie (en deux étapes : déplacement et fusions)
- add_tile (grid g) : on ajoute une case d'une valeur de 2 ou de 4 dans une case vide au hasard
- play (grid g, dir d) : on effectue le déplacement s'il est possible, puis on fait apparaitre aléatoirement une tuile dans une case vide

En parallèle de l'écriture de ces fonctions, nous avons réalisés les test associés pour vérifier leur bon fonctionnement. Par exemple vérifier qu'après la création, toutes les tuiles sont initialisée à 0 ou tester si avec la fonction add_tile on a un bien une case en plus après avoir utilisé la fonction, etc.

1.2 Les stratégies

cf. partie 4

1.3 L'interface graphique

La dernière partie sur laquelle nous avons travaillé est l'interface graphique. Pour cela nous avons utilisé la bibliothèque graphique SDL2 avec les partie SDL_TTF pour la gestions de la police et des écritures et SDL_image pour l'affichage en général. Nous avons commencé par afficher la grille en adaptant l'écart entre chaque ligne de la grille en fonction de GRID_SIDE afin que la grille soit adaptable dès qu'on change GRID_SIDE. Nous avons ensuite procédé à l'affichage des tuiles en faisant en sorte que la taille du texte s'adapte à la taille de la case afin que le texte ne dépasse jamais de la case. Nous avons utilisé le même procédé afin d'afficher les couleurs des tuiles de la bonne taille. Pour les couleur des tuiles nous avons utilisé un système de decrémentation des couleurs g et b en fonction de la puissance de 2 afin que la couleur devienne de plus en plus vive.

2 Explication d'un problème technique

2.1 Problème technique rencontré

L'un des problèmes techniques rencontré concernait la complexité de la fonction do_move. Cette fonction a pour objectif de faire un mouvement dans le jeu. Dans le jeu 2048, un mouvement se fait fait en deux étapes : on bouge toutes les tuiles selon une direction et on réalise des fusions si c'est nécessaire (si deux tuiles de même valeur se rencontrent). Notre première version de do_move fonctionnait, mais elle avait une complexité élevée (en n^2) en plus d'être difficilement factorisable. Il nous a donc paru judicieux de revoir complètement le paradigme de cette fonction (i.e. : de la réécrire entièrement en pensant différemment) avec une complexité moindre et une meilleure lisibilité.

2.2 Résolution du problème

Pour effectuer un seul parcours de tableau (i.e. avoir une complexité en n), il nous faut des marqueurs (flags). On les place à chaque nouveau parcours de ligne ou colonne (selon la direction). Pour les mouvements UP et DOWN, les flags se situent sur les indices verticaux (l'axe j, parcours de colonnes). Pour les mouvements LEFT et RIGHT, les flags se situent sur les indices horizontaux (l'axe i, parcours de lignes).

On a deux types de flag :
- flagMerge : on place un flag qui correspond à la case qu'il faut comparer pour la fusion (c'est utile pour éviter des cas de fusions multiples)
- flagMove : on place un flag qui indique où il faut positionner la valeur si la fusion n'est pas réalisée

```
void do_move (grid g, dir d) {
  assert(can_move(g,d));

  switch(d) {

  case UP:
    for (int i = 0 ; i < GRID_SIDE ; i++) {
      int flagMerge = 0;
      int flagMove = (get_tile(g,i,0) == 0)?0:1;
      for (int j = 1 ; j < GRID\_SIDE ; j++) {
        int n = get_tile(g, i, j);
        if (n != 0) {
          if (n == get_tile(g, i, flagMerge)) {
            set_tile(g, i, flagMerge, n+1);
            set_tile(g, i, j, 0);
            g->score += pow(2, n+1);
            victory(n+1);
            flagMerge++;
          }
          else {
            set_tile(g, i, flagMove, n);
            if (flagMove != j)
              set_tile(g, i, j, 0);
            flagMerge = flagMove;
            flagMove++;
          }
        }
      }
    }
    break;

  case DOWN:
    ...
```

3 Exemple de factorisation de code

Ci-dessous, nous avons un exemple de factorisation de code. C'est le code de can_move qui a été factorisé grâce à la fonction can_move_facto(grid g,int imin, int imax, int jmin, int jmax, int i1, int j1) et pour chaque direction on doit mettre les paramètres correspondant. Les quatre premiers paramètres servent de minimum et de maximum pour les deux for et les deux derniers paramètres(i1 et j1) servent à incrémenter i et j.

```
static bool can_move_facto (grid g, int imin, int imax, int jmin,
int jmax, int i1, int j1) {
  for (int i = imin ; i < imax ; i++) {
    for (int j = jmin ; j < jmax ; j++) {
      if((g->t_grid[i][j] == g->t_grid[i+i1][j+j1] ||
            g->t_grid[i+i1][j+j1] == 0) && g->t_grid[i][j] != 0)
        return true;
    }
  }
  return false;
}

bool can_move (grid g, dir d) {
  switch(d) {

  case UP:
    return can_move_facto(g, 0, GRID_SIDE, 1, GRID_SIDE, 0, -1);
    break;

  case DOWN:
    return can_move_facto(g, 0, GRID_SIDE, 0, GRID_SIDE-1, 0, 1);
    break;

  case LEFT:
    return can_move_facto(g, 1, GRID_SIDE, 0, GRID_SIDE, -1, 0);
    break;

  case RIGHT:
    return can_move_facto(g, 0, GRID_SIDE-1, 0, GRID_SIDE, 1, 0);
    break;

  default:
    break;
  }
}
```

4 Démarche pour l'élaboration des stratégies

Le jeu étant fonctionnel pour un joueur humain, la prochaine étape consistait à créer une intelligence artificielle capable d'atteindre 2048 presque systématiquement.

4.1 Première approche : stratégies triviales

La première stratégie qui nous est venue à l'esprit est une stratégie aléatoire qui choisie une direction au hasard à chaque coup. Elle a été assez simple à implémenter, mais la performance était faible (on ne dépassait presque jamais 256).

```
dir play_move_random(strategy s, grid g) {
    dir t[] = {UP, DOWN, LEFT, RIGHT};
    int i = rand()%4;
    return t[i];
}
```

Ensuite, nous avons pensé à une autre stratégie triviale qui consiste à alterner successivement les mouvements UP, DOWN, LEFT et RIGHT. Son implémentation a été tout aussi simple que pour random, mais la performance n'était pas au rendez-vous.

```
dir play_move_basique(strategy s, grid g) {
      dir t[] = {UP, DOWN, LEFT, RIGHT};
      return t[((*(int*)s->mem)++)%4];
}
```

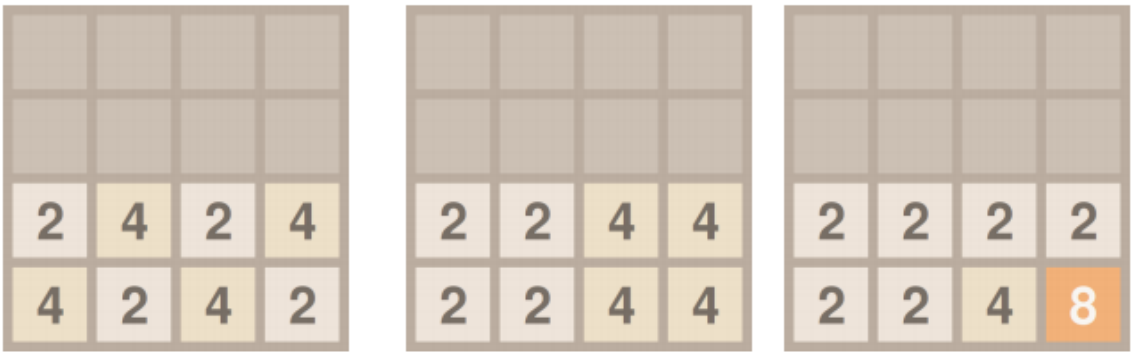
4.2 Évaluation du "score" d'une grille

Pour élaborer une stratégie plus efficace, nous avons choisi d'attribuer à chaque état du plateau un "score". Ce score est un nombre qui est d'autant plus grand que la configuration est favorable pour le joueur. Techniquement, c'est une fonction qui prend en paramètre une grille et renvoie un entier. Une des difficultés pour avoir une bonne intelligence artificielle est d'avoir une bonne fonction de score.

Ce score est calculé à partir de différents critères que nous avons choisis heuristiquement et que nous avons ensuite pondérés.

Par exemple un critère à prendre en compte est le nombre de cases vides. En cherchant à maximiser le score, nous maximiserons les cases vides et privilégierions les directions qui fusionnent le plus de tuiles.

Mais ce n'est qu'un critère basique que nous pouvons coupler à d'autres.



(a) Configuration non-monotone (b) Configuration progressive (c) Meilleure tuile

FIGURE 4.1 – Exemple de grilles

Par exemple sur la figure 4.1 la stratégie "nombre de cases vides" donne le même score (8). Pourtant, la configuration (a) est très mauvaise pour le joueur car les tuiles 2 coincées entre des 4 seront très difficiles à fusionner avec d'autres. La configuration (b) est bien meilleure. Une façon de s'en rendre compte est de parcourir chaque ligne et chaque colonne de la grille et de compter le nombre de fois où l'agencement des tuiles est non monotone. Pour la configuration (a), cela se produit deux fois par ligne alors que ça ne se produit pas pour la configuration (b). La monotonicité sera donc un critère important dans l'élaboration de notre fonction score.

La configuration (c) est plutôt meilleur que la configuration (b) car on a réussi à obtenir une tuile plus grosse. On peut remarquer que la plus grande tuile est dans un coin, ce qui est en général une bonne chose. Cela nous rajoute donc deux critères : meilleure tuile, meilleure tuile située dans un coin (ou à défaut sur un bord).

Ci-dessous se trouvent la liste des fonctions qui permettent de quantifier les différents critères que nous avons retenus :

- count_empty_tiles : retourne le nombre de cases vides
- count_non_monotonicity : retourne le nombre de non monotonicités
- search_best_tile : retourne la valeur de la tuile la plus grande
- is_best_tile_in_corner : retourne 1 si la meilleure valeur se trouve dans un coin
- is_best_tile_on_edge : retourne 1 si la meilleure valeur se trouve sur un bord

Le score est ensuite calculé grâce à ce calcul pondéré :
10 * empty_tiles - 11 * non_monotonicity+ best_tile + 60 * best_tile_in_corner + 50 * best_tile_on_edge

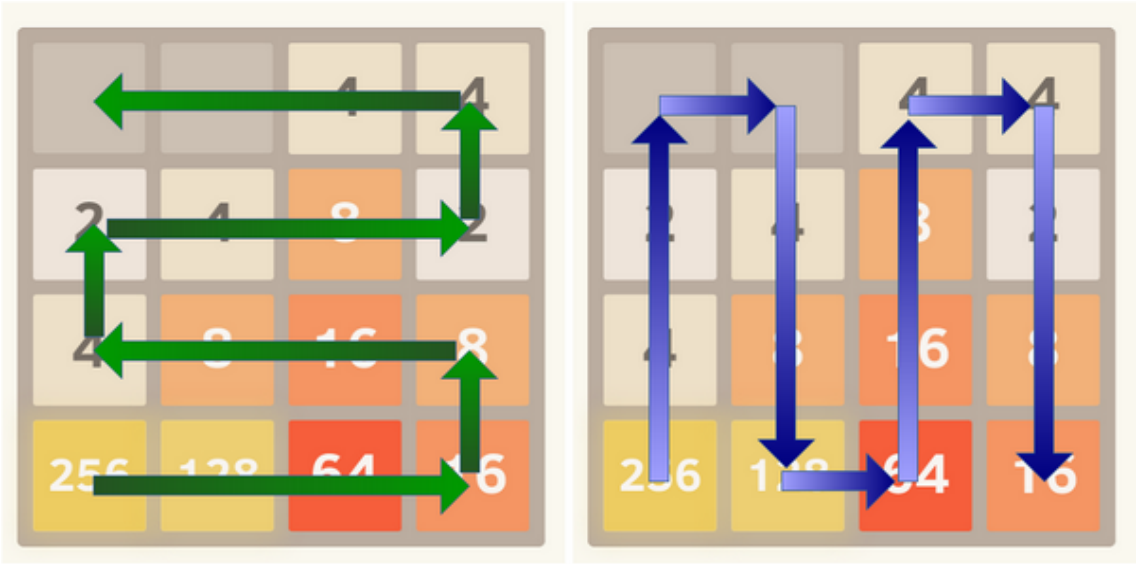


FIGURE 4.2 – La stratégie idéale

Sur la figure 4.2, nous pouvons voir une configuration de grille idéale "en serpent". On peut constater que les critères que nous avons cités ci-dessus sont présents : meilleure tuile dans un coin, peu de non monotonicités, le plus de cases vides possibles, etc.

Pour améliorer la fonction score, il faut valoriser les grilles qui se rapprochent le plus de cette configuration.

4.3 Algorithme MINMAX

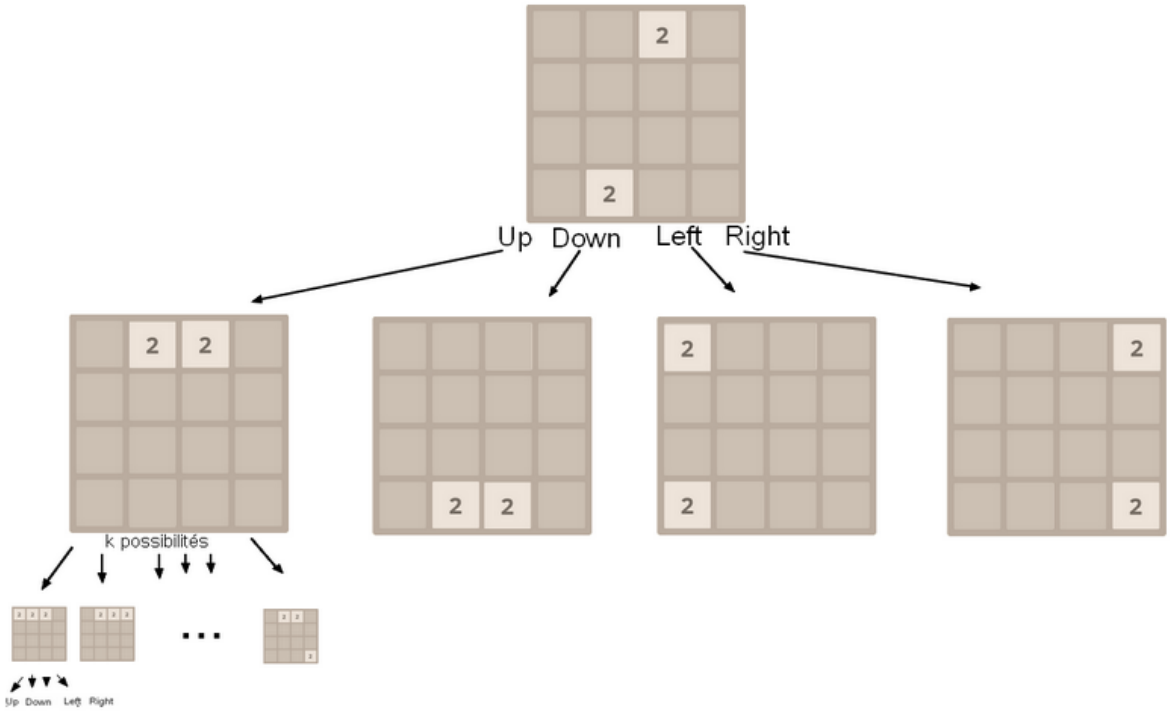


FIGURE 4.3 – Parcours récursif en profondeur (algorithme minmax)

Après avoir élaboré notre stratégie score, nous avons choisi d'utiliser l'algorithme MINMAX qui est un classique de théorie des jeux et d'intelligence artificielle. Au lieu de directement jouer "la direction qui maximise le score", on va considérer le choix de l'autre joueur (ici le placement aléatoire de la tuile après un déplacement). Il y a donc autant de choix possibles (k) que de cases vides. Comme il s'agit d'un choix aléatoire, on ne va pas prendre le cas "min" comme on le fait habituellement, mais une moyenne des scores obtenus à cet étage.

Comme il est représenté sur la figure 4.3, on va donc successivement alterner un algorithme qui choisit le max parmi les quatre directions possibles, puis un calcul de la moyenne des k grilles possibles, puis le max parmi les quatre directions possibles pour chacune des k grilles, etc.

Les algorithmes sont identiques pour la stratégie fast et la stratégie efficient, et seule la profondeur de recherche change dans l'algorithme MINMAX.

Il nous a d'ailleurs paru judicieux d'ajuster dynamiquement la profondeur en fonction du nombre de cases vides. En effet moins il y a de cases vides, plus on augmente la profondeur de recherche.

4.4 Performances

Les stratégies random et basique ont des performances faibles. En effet sur 100 essais elles ont atteint 4 fois 512, 55 fois 256 et moins le reste du temps.

La stratégie fast est plutôt performante en atteignant 2048 dans 35% des cas et 1024 dans 43% des cas (cf. figure 4.4).

Enfin, la stratégie efficient obtient 2048 ou au moins 1024 à quasiment tous les coups.

Version «fast» : 100 parties 10s max			Tuile Max										
nom stratégie	meilleur scor	temps exécution	8	16	32	64	128	256	512	1024	2048	4096	8192
strategy_fast	55320	821.9367870	0	0	0	0	0	0	0	14	47	39	0
Devil's Strat Fast	55904	818.6159200	0	0	0	0	0	0	2	13	46	39	0
A4_bonnet_jajoux_fast	50636	245.4891560	0	0	0	0	0	2	8	47	42	1	0
A2 Labbe Labaye Sarrabayrouse Fast	32772	663.6691480	0	0	0	0	0	1	12	48	39	0	0
A4_parpaite_seegers_legarrec_fast	15788	745.4223890	0	0	0	0	0	6	16	43	35	0	0
fast algo groupe H	31420	213.1464590	0	0	0	0	1	6	39	47	7	0	0
A1_bedin_etcheverry_schiapparelli_fast	27464	27.4497190	0	0	0	0	2	10	53	33	2	0	0
Strat��gie rapide (Bent��jac folliet lenoir)	14312	958.6562740	0	0	0	0	0	1	11	88	0	0	0
BINEAU PORTELAS FAST	15072	914.9201430	0	0	0	0	2	5	42	51	0	0	0
libA2_badji_peccavet_truong_fast.so	14172	335.59647	0	0	0	0	5	41	49	5	0	0	0
fast	6004	0.0685810	0	0	0	14	45	35	6	0	0	0	0
_benetti_daugieras_raby_fast	5592	0.0910860	0	0	3	9	35	48	5	0	0	0	0
	6392	0.0829520	0	0	1	12	37	45	5	0	0	0	0
PirroneSauzetBendi-OuisRapide	6016	0.1799710	0	0	0	1	26	69	4	0	0	0	0
A3_herfurth_lirzin_paulmier_fast	6608	0.6226280	0	0	2	6	33	55	4	0	0	0	0
PASY	4680	0.0868170	0	0	0	17	59	23	1	0	0	0	0

Stratégies discalifiées (erreur de chargement ou erreur à l'e
libA1_AlChahid_Aussignac_Diall_Guichard_fast.so
libA2_bifert_daubasse_gallardo_fast.so
libA2_bonnet_borde_pinero_fast.so
libA3_bourdarie_cubero_tolar_fast.so
libA3_camus_pellegrini_cougouilles_fast.so
libA3_elmassari_lequerec_negueloua_fast.so
libA3_lugan_guessoum_wolski.so
libA4_dietrich_trottier_romdan_fast.so
libstrat_over9000.so

FIGURE 4.4 – Résultats du tournoi sur la stratégie fast

Bilan : objectifs atteints et non-atteints

Les objectifs principaux de ce projet ont été atteints :
- travailler en groupe avec un environnement de développement collaboratif (github)
- développer une application permettant de jouer à 2048 sur terminal, puis en mode graphique
- proposer des stratégies (plus ou moins rapides) permettant de jouer automatiquement à 2048 avec des performances raisonnables

Cependant, certains points auraient mérité d'être plus aboutis (peaufinement) :
- factorisation du code à certains endroits
- interface graphique plus agréable avec des boutons
- amélioration possible de la stratégie efficient avec le paradigme "en serpent" (obtenir 4096 ou même 8192)