

Rapport du projet de structure de données

Informatique 1ère année

Mouhab ZITOUNI

Nour El Houda GALAI

Responsable : John CHAUSSARD

Remerciements

Nous tenons à remercier M. CHAUSSARD d'avoir assuré un cours complet de structures de données et des TP bien dirigés en présentiel et en en distanciel, qui nous ont aidé à former une base solide en langage C.

Sommaire

Introduction	page 3
Structure de données utilisées pour implémenter les différentes configurations.....	page 4
La fonction de hachage permettant de stocker les différentes configurations des grilles	page 6
1.Réduction de nombre de configurations de 19683 à 304	page 6
2. Fonction de hachage.....	page 8
Remplissage de la table de hachage	page 9
Fonctionnement de l'intelligence artificielle.....	Page 11
1. Récompenses et punitions.....	Page11
2. Mécanisme de la décision	Page11

Introduction

Depuis plus de cinq décennies, l'intelligence artificielle vit une accélération dans son développement et son adoption. En effet, c'est un processus d'imitation de l'intelligence humaine qui repose sur la création et l'application d'algorithmes exécutés dans un environnement informatique dynamique.

A ce propos, ce programme permet de jouer au jeu du morpion entre l'utilisateur et l'ordinateur, ou entre ordinateur-ordinateur, en s'appuyant sur la façon MENACE qui consiste à mémoriser au fur et à mesure les coups gagnés et les coups perdus et les utiliser plus tard pour les coups suivants.

Structure de données utilisées pour implémenter les différentes configurations

Afin de modéliser les grilles représentant les différentes configurations, on a implémenté la structure `box_matches` (référée aux boîtes d'allumettes dans MENACE), qui définit une configuration, au premier lieu, par son nombre calculé sur la base 10, en passant par la base 3, en associant le chiffre 2 aux croix, le 1 aux cercles et le 0 aux espaces, et que la case en bas à droite représente les unités ; ce qui est modéliser par `poss_grille`. Au second lieu, par `billes` qui définit le nombre de billes associé à chaque case d'une configuration, qui sera défini ultérieurement de façon tel que la méthode MENACE le décrit. Au troisième lieu, par la variable `ouverte`, qui est soit 0 soit 1, pour décrire si la boîte d'allumettes est ouverte ou fermée, soit désignant si le nombre de billes va changer suite à une partie terminée (les configuration utilisées).

```
typedef struct box_matches
{
    uint16_t poss_grille;
    uint64_t *billes;
    uint8_t ouverte;
    struct box_matches *next;
} box_matches;
```

Ensuite, en vue d'implémenter la fonction de hachage, on définit la structure `tab_hachage`, qui prend, d'une part, des sous-listes, de type `liste` ; qui elle-même se représente comme une structure ayant tête et queue de type `box_matches`, et taille de type `uint32_t`. Et, d'autre part, la taille de cette « grande liste » de type `uint32_t`.

NB : On a choisit comme type de tailles des listes entier non signé sur 32 bits, pour que cela soit applicable pour un jeu de morpion sur des grilles de taille plus que 3x3 (4x4, 5x5 ,etc.).

```
typedef struct liste
{
    uint32_t taille;
    box_matches *tete;
    box_matches *queue;
} liste;
```

```
typedef struct
tab_hachage
{
    liste **tab;
    uint32_t taille;
} tab_hachage;
```



La fonction de hachage permettant de stocker les différentes configurations des grilles

1. Réduction de nombre de configurations de 19683 à 304 :

Pour commencer, le jeu de morpion sur des grilles 3x3, donne 3^9 soit 19683 possibilités, qui est un très grand nombre à parcourir à chaque fois. N'oublions pas qu'en plus des similarités entre les grilles (rotation de 90° , de 180° , de 270° , et celles combinées avec la symétrie verticale), on a aussi des configurations absurdes, tel que ce n'est pas possible de trouver une grille contenant que des « X » ou des « O » dans toutes les cases, ou même le nombre des « X » dépasse celui des « O » +1 et inversement, ou encore les configurations gagnantes tel que 3 « X » ou 3 « O » alignés, qui ne servent à rien si on les stocke dans la table de hachage, puisque le tour se termine à ce niveau-là, et pas d'intelligence artificielle peut avoir lieu à partir de ce genre de coups.

A ce propos, au début, on a essayé de modéliser les similarités en se basant sur la représentation sur la base 3, tel que la façon consiste à stocker les puissances de 3 dans 8 tableaux, représentant chacun une similarité particulière (rotation-90, rotation-180, symétrie etc..). Ensuite, la comparaison se base sur ces tableaux (en comparant les puissances de 3). Tel qu'il l'explique le code suivant :

8	7	6
5	4	3
2	1	0

	6	3	0
	7	4	1
	8	5	2

Configuration originale

```
uint8_t tab_orig[9]={8,7,6,5,4,3,2,1,0};
uint8_t tab_90[9]={6,3,0,7,4,1,8,5,2};
uint8_t tab_180[9]={0,1,2,3,4,5,6,7,8};
uint8_t tab_270[9]={2,5,8,1,4,7,0,3,6};
uint8_t tab_sym[9]={6,7,8,3,4,5,0,1,2};
uint8_t tab_sym_90[9]={8,5,2,7,4,1,6,3,0};
uint8_t tab_sym_180[9]={2,1,0,5,4,3,8,7,6};
uint8_t tab_sym_270[9]={0,3,6,1,4,7,2,5,8};
```

Configuration similaire avec rotation de 90°

```
uint16_t nbr_decim_90(uint8_t grille[3][3])
{
    uint8_t i, j, k;
    uint16_t s;
    k=0;
    for (i=0;i<3;i++)
        for (j=0;j<3;j++)
        {
            s+=pow(grille[i][j],tab_90[k]);
            k++;
        }
    return s;
```

On calcule le nombre associé à la grille avec rotation de 90° en base décimale pour que l'on compare après avec les configurations originales déjà stockées dans la table de hachage, si égalité alors les deux configuration sont similaires, et on élimine celle-là en laissant l'originale déjà existante dans la table de hachage sinon on passe à la fonction nbr_decim_180 qui fait le même travail mais avec celle de rotation de 180° (avec le tableau tab_180) et ainsi de suite. Si on a essayé toutes les similarités sans égalité, on stocke cette grille comme nouvelle configuration, et on passe à la suivante.

On a laissé tomber cette idée plus tard, car, à peu près, cela revient au même au niveau de complexité, et quand on a essayé de l'implémenter dans le programme, il y'avait des erreurs de segmentation donc on n'a pas voulu perdre plus de temps pour la traiter tandis qu'on a d'autre part les fonctions de similarités déjà données avec l'énoncé du projet.

2.Fonction de hachage :

Lors des tours, afin de parcourir les configurations, aisément dans un temps réduit, on a implémenté une table de hachage, qui consiste à stocker les différentes grilles en se basant sur le nombre de coups, comme le montre la fonction ci-joint :

Si $g[i][j] \neq 0$, alors cette case contient soit «X» soit «O», donc il y'a un coup de plus joué. Alors, la valeur de r , qui représente le nombre des coups, va être augmentée de 1.

```
uint8_t fnc_hachage(tab_hachage *th, uint8_t
g[3][3] )
{
    uint8_t r=0;
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            if(g[i][j])
                r++;
        }
    }
    return r;
}
```

Contrainte : les configurations ne sont pas équiparties dans la table de hachage, par exemple pour celles associées à 0 coups joués, on a une seule, pour celles associés à deux coups joués on a 72 configurations et ainsi de suite. Malheureusement, on n'a pas trouvé une autre idée pour régler ce problème, car la fonction de hachage est basée surtout sur le nombre des coups pour que cela soit accessible à l'intelligence artificielle et qu'elle soit capable de prendre sa décision à propos son prochain coup.

❖ Remplissage de la table de hachage

L'idée est basée principalement sur les deux points précédents mentionnée ; D'une part, la vérification des cas similaires, impossibles et coups gagnants pour qu'on aboutisse à un nombre optimal de configurations. Et d'autre part, le parcours consistant à la fonction de hachage, autrement dit, sur le nombre des coups :

```
void fill_hash_tab(tab_hachage *hash_tab)
{
    uint8_t key=0;
    uint8_t g[3][3] = {{0, 0, 0}, {0, 0, 0}, {0, 0, 0}};
    add_tete_th(hash_tab, g);

    box_matches *e = NULL;

    while(next_configuration(g)==CONTINUE) {
        key = fnc_hachage(hash_tab, g);
        if (impossible_config(g)==0 && end_game(g)==0 && key%2==0 &&
key<8){
            key = key/2; // les coups que le pc va rencontrer
            e = hash_tab->tab[key]->tete;

            while( e != NULL && symetrie(e->poss_grille, g )==0) {
                e = e->next;
            }

            if (e == NULL)
            {
                add_tete_th(hash_tab, g);
            }
        }
    }
}
```

Key reçoit la valeur que la fonction de hachage va renvoyer. On vérifie si elle est paire, car dans notre programme, c'est l'ordinateur qui va commencer de jouer, donc il va rencontrer que les configurations tel que le nombre des «X» est égal à celui des «O», autrement dit les configurations tel que fnc_hachage renvoie un entier pair.

D'autre part, on a divisé ce key par deux, car, comme expliqué, on s'intéresse qu'aux configurations tel que l'ordinateur reçoit, donc ça ne sert à rien de stocker celles associées aux key impairs.

Fonctionnement de l'intelligence artificielle

1. Récompenses et punitions :

Comme expliqué par le fonctionnement de MENACE, l'apprentissage de l'ordinateur, autrement dit, l'intelligence artificielle est basée sur un système de récompenses et de punitions.

L'idée consiste à conserver les configurations utilisées par l'ordinateur, temporairement, jusqu'à la fin du jeu. Ensuite, si c'était un coup gagnant pour l'ordinateur il ajoute 3 billes aux configurations utilisés, si c'est nul il ajoute une seule, et si c'était perdu il retransche une bille de chacune.

2. Mécanisme de la décision :

L'idée consiste à choisir au hasard une des billes associées à une configuration donnée.

Pour ce faire, on peut assimiler la variable *billes associée à la structure box_matches par un tableau tel que chaque case contient le nombre de billes de chaque case, et on choisit aléatoirement une des cases de ce tableau, puis on fait le parcours à l'envers, pour savoir c'est la bille propre à quelle case.

```
uint8_t random_select(box_matches *e){
    srand(time(0));
    uint32_t total_billes=0;
    uint32_t random_bille=0;
    uint8_t j=0;
    for (int i = 0; i < 9; ++i){
        total_billes = total_billes + e->billes[i];
    }
    random_bille = (rand()%total_billes)+1;
    while(random_bille>0)
    {
        random_bille = random_bille - e->billes[j];
        j++;
    }
    return j-1;
}
```

Cette fonction renvoie j-1 qui va être l'indice que l'on associe à la grille par cette façon :

```
/*extrait de la fonction menace_play*/
menace_choice = random_select(nv);
g[menace_choice/3][menace_choice%3] = 1;
```

L'entier que random_select renvoie, nous donne l'indice du ligne en le divisant par 3 et l'indice de colonne avec le modulo 3.

Et c'est logique, puisque la case ayant le nombre de billes le plus grand est comme si un intervalle le plus grand tel que la probabilité de tomber sur celui-ci est plus que les autres.