

JIA Yunfei

INFO 1

Tuteur enseignant :

M.CHAUSSARD

Matière :

Programmation Avancée –

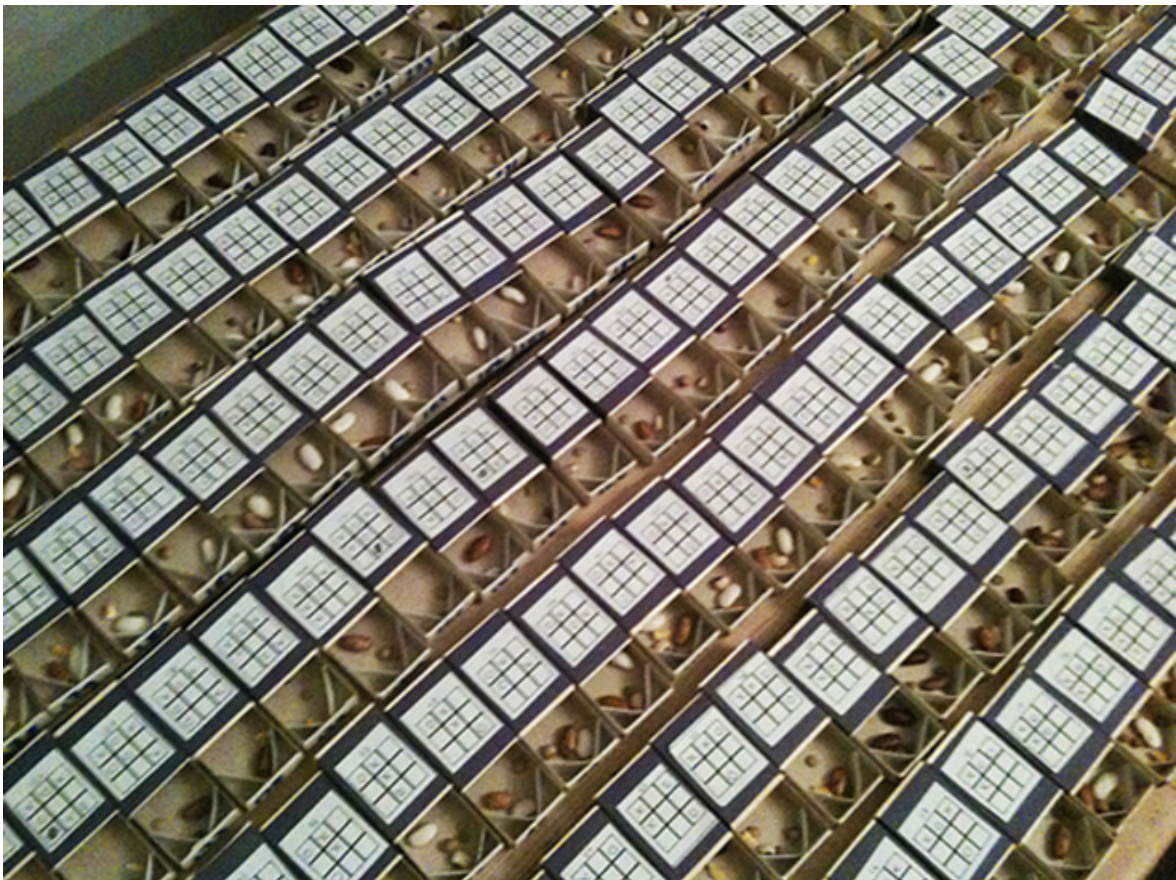
Structures de données



Rapport du projet

Morpion IA (MENACE)

Décembre 2021 – janvier 2022



Projet sur GitHub :

<https://github.com/YunfeiJIA1904/Projet/tree/main/Morpion%20Menace>

SOMMAIRE

| | |
|------------------------------|----|
| INTRODUCTION | p4 |
|------------------------------|----|

| | |
|--|----|
| I / PRÉSENTATION DU PROJET | p5 |
|--|----|

[II / PROGRESSION DU PROJET](#)

| | |
|---|----|
| A - Choix de la structure | p6 |
|---|----|

| | |
|---|----|
| B - Déroulement du projet | p8 |
|---|----|

| | |
|---|-----|
| III / RÉSULTATS OBTENUS | p22 |
|---|-----|

| | |
|----------------------------|-----|
| CONCLUSION | p23 |
|----------------------------|-----|

INTRODUCTION

La matière Programmation avancée et Structures de données a pour objectif de nous apprendre les différents moyens de stocker et de traiter les données, mais aussi de comprendre comment bien gérer la mémoire.

Le projet consiste à réaliser un jeu de morpion où l'adversaire est une IA, qui apprend à chaque partie pour devenir meilleur.

Tout d'abord, je fais rapidement une introduction sur le sujet.

Puis, j'expliquerai mon choix de structure de donnée pour ce projet.

Ensuite, je développerai les démarches que j'ai effectuées et notamment les recherches, les problèmes rencontrés, les remarques intéressantes, du résultat, etc.

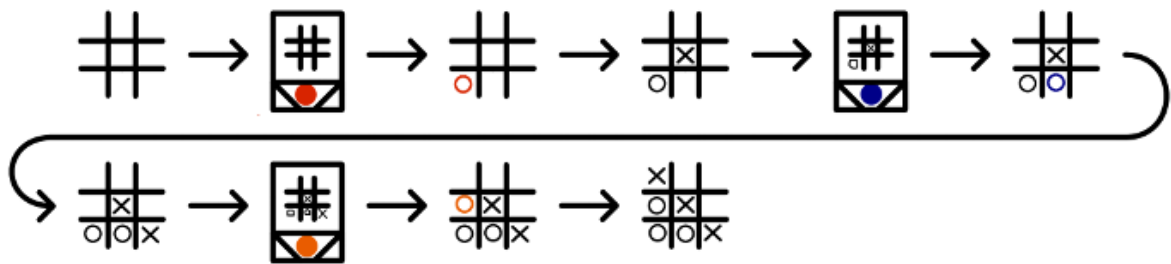
Enfin, je parlerai de ce que ce projet m'a apporté et mon avis personnel.

I / PRÉSENTATION DU PROJET

Ce projet consiste à coder une IA qui arrive à jouer au morpion et s'améliore en fonction du résultat de chaque partie pour devenir meilleur dans le jeu.

L'idée qu'on va suivre est celui de Projet MENACE proposée en 1961 par Donal Michie qui fonctionne avec les 304 matches box qui représente chacune une configuration du jeu et contient dans sa boîte des billes de couleur différente qui représente chacune un placement valide.

Le déroulement de ce système est simple, on retrouve la configuration qui représente la grille actuelle puis on tire au sort une bille de la boîte qui représente un placement valide et joue ce coup, enfin on répète la même démarche jusqu'à la fin du jeu.



Ici, le rond représente l'IA et en fonction du jeu actuel, on tire au sort une bille et joue le coup.

L'IA s'apprend à l'aide du système de "**RÉCOMPENSES et PUNITIONS**" :

- **RÉCOMPENSE** :

- Si gagnant : on ajoute 3 billes de la même couleur dans les boîtes ouverte.
- Si nulle : on ajoute 1 bille [...].

- **PUNITIONS** :

- Si perdant : on retire 1 bille [...].

Plus un chemin gagne, plus le nombre des billes jouées de ce chemin seront nombreux, donc plus de chance de retomber sur ces billes lorsque le même chemin reproduit.

II / PROGRESSION DU PROJET

A - Choix de la structure

Pour commencer, la plus importante chose est de savoir quelles sont les structures de données les plus adaptées et ainsi choisir celle qui est optimale et simple à utiliser pour ensuite la mettre en place.

Pour cela, j'ai listé toutes les structures et choisi deux entre eux qui sont la **liste chaînée** et la **table hachage**.

L'idée pour mettre en pratique ces deux structures est :

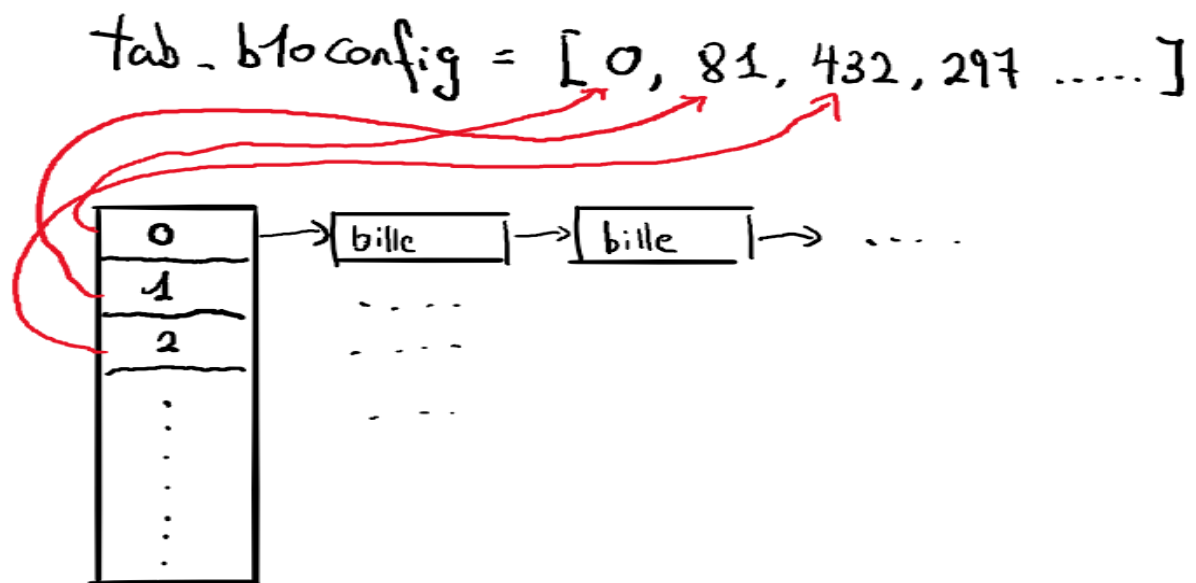
- Liste chaînée :

Une simple liste qui enregistre chaque configuration dans un maillon.



- Table hachage :

Une table hachage qui enregistre toutes les configurations et chaque configuration contient des billes (data de chaque bille représente un emplacement valide de cette configuration) et grâce à un tableau des configurations (traduit en base 10), on pourra savoir quel indice de la table hachage correspond à quelle configuration.



Après réflexion, j'ai choisi la **structure de la table hachage** qui me paraît plus simple pour gérer les billes, par exemple, lorsque l'IA gagne, je peux directement ajouter des billes dans la configuration en utilisant la fonction **add_queue()** pour une liste.

```
typedef struct _bille
{
    uint8_t data;
    struct _bille *suivant;
    struct _bille *precedent;
}bille;
```

```
typedef struct
{
    uint32_t taille;
    bille *tete;
    bille *queue;
}box;
```

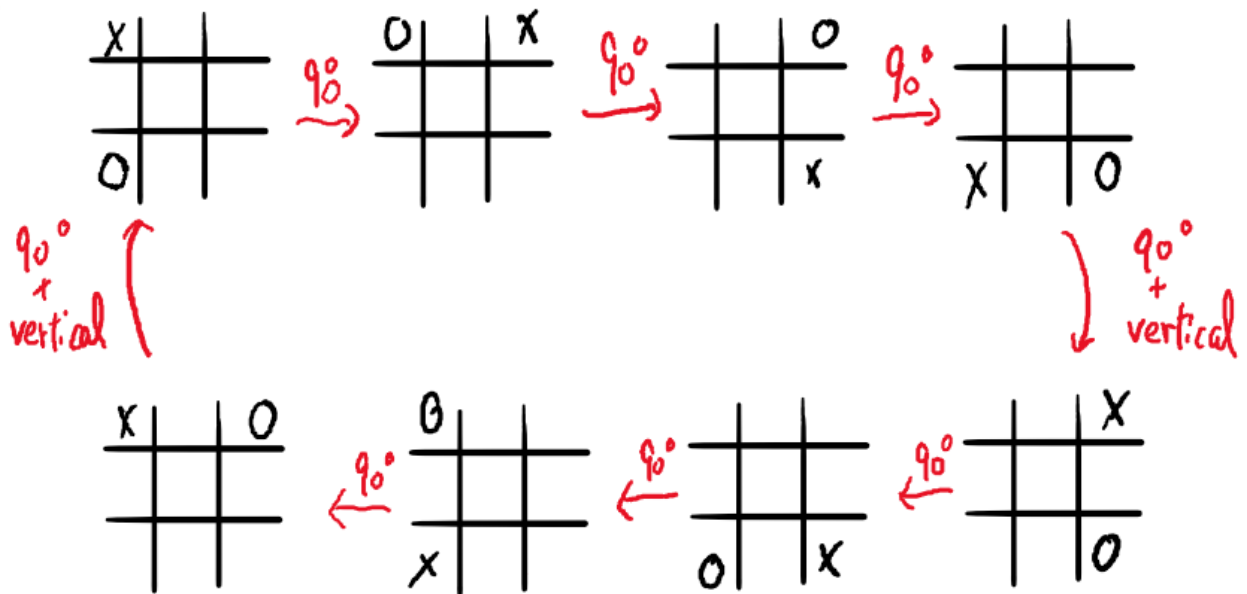
```
typedef struct
{
    box **tab;
    uint32_t taille;
}tab_box;
```

- **struct** bille : Les maillons qui représentent des billes.
uint8_t data : Les data de chaque maillon représente un emplacement valide de la configuration.
- **struct** box : Liste d'une table de hachage qui représente la configuration et qui contient des billes.
- **struct** tab_box : Une table hachage qui représente l'ensemble des configurations.

B - Déroulement du projet

Tout d'abord, je dois décider les données à utiliser (les configurations qui représentent chaque situation). Après de nombreuses recherches du projet MENACE, je constate qu'il doit y avoir au moins 304 boîtes qui représentent toutes les configurations (en faisant des rotations, on aura 7 autres configurations donc en total, on a 304*8 configurations).

Et j'ai décidé de traduire ces 304 configurations en base 10, puis, les écrire dans un fichier **b10Config.txt** et faire la même démarche pour les bases 3 dans un fichier **b3Config.txt**.



Ensuite, les datas(emplacement valide) enregistrées dans chaque bille(maillon) est représenté sous forme un chiffre entre 0-8 et qui représente les 9 cases de la grille du jeu, car la grille en 1d est plus simple à traiter que celle de 2d.(par exemple 0 au lieu de [0][0], 8 au lieu de [2][2], etc.)

| | | |
|---|---|---|
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

Après d'avoir défini la structure et obtenu une idée claire, j'ai passé au codage des fonctions qui sert à traduire une grille 3x3 (un tableau 1d de taille 9) en base 10 et l'inversement.

- ❖ **Pour passer de grille en base 10**, comme les valeurs de la grille est déjà en quelque sorte une base 10, donc il suffit juste d'appliquer la formule pour passer en base 10.

```
uint32_t grille_to_b10(uint8_t grille[9])
{
    uint32_t result = 0, exp = 0;
    for (int j = 8; j >= 0; j--)
    {
        result += grille[j] * pow(3, exp);
        exp++;
    }
    return result;
}
```

- **uint8_t grille[9]** : un tableau de 9 entiers entre 0-2 où 0 est vide, 1 est croix, 2 est rond.

- ❖ **Pour passer de base 10 en grille**, utilisation de division successive puis inverser le résultat:

```
void b10_to_grille(uint32_t b10, uint8_t* newgrille)
{
    uint8_t temp[9] = {0};
    uint32_t quotient = b10, i = 0, compt;
    //trouver le base3 et mettre dans temp[9] (ici tab est inversé)
    //ici on a utiliser le technique de division successive et pour trouver la base3, on doit
    enregistrer tous les reste de chaque division
    while (quotient != 0)
    {
        temp[i] = quotient % 3;
        quotient = (quotient - quotient % 3) / 3;
        i++;
    }
    compt = 9;

    //remettre tab en bon ordre cad inverser le tab qu'on a récupéré précédemment
    for (int i = 0; i < 9; i++)
    {
        newgrille[i] = temp[compt - 1];
        compt--;
    }
}
```

- **uint32_t b10** : une configuration traduit en base 10.
- **uint8_t* newgrille** : un tableau de 9 entiers comme celle de la grille et qui est modifié directement après l'appelle de la fonction. (utilisée dans la fonction pour trouver tous les similaires d'une configuration, cad d'abord transformer une configuration de base 10 en grille puis faire les rotations).

Ensuite, comme j'ai choisi de manipuler seulement du 1d, pour cela je suis obligé de modifier la fonction suivant du fichier **aide_projet** en 1d en changeant simplement les case de 2d en 1d, par exemple: [2][2] en 8 (en raison de la longueur du code je ne le mettrai pas ici) :

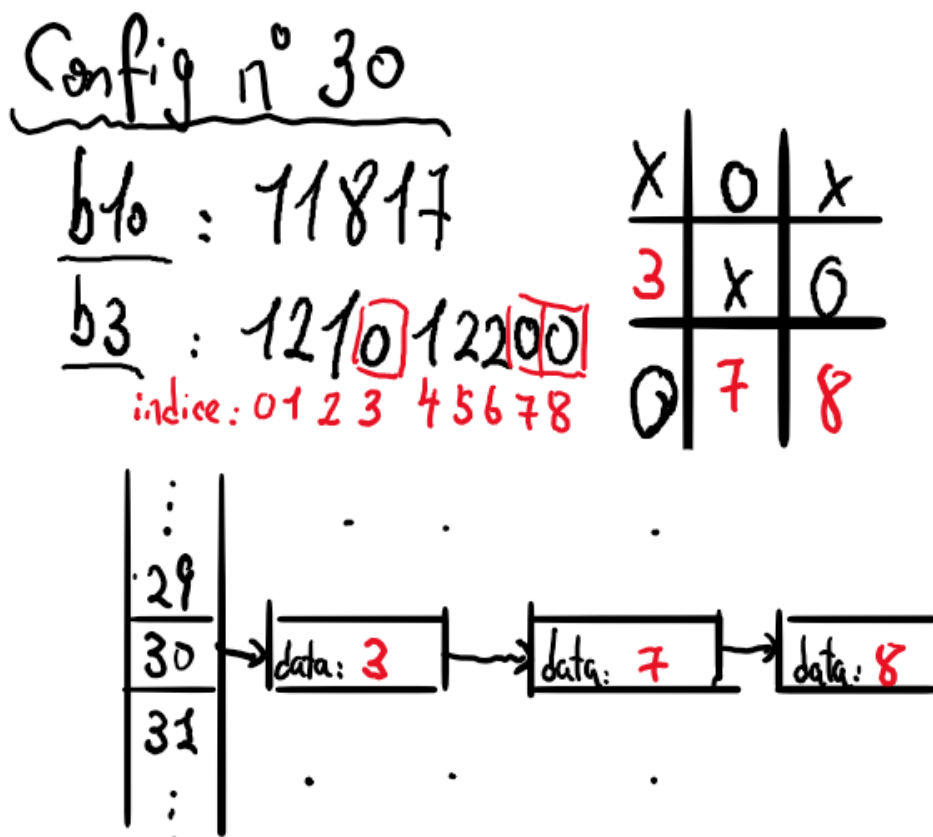
```
void appliquer_transformation_base(uint8_t grille[3][3], transformation tr)
```

et qui prennent maintenant des grilles de 9 entiers :

```
void appliquer_transformation_base(uint8_t grille[9], transformation tr)
```

Après la modification et création de la fonction de base (les fonctions de structure comme `add_queue()` pour une liste, etc.), je me suis lancé dans le codage du jeu avec les 304 configurations de base qui sont utilisées dans le projet MENACE.

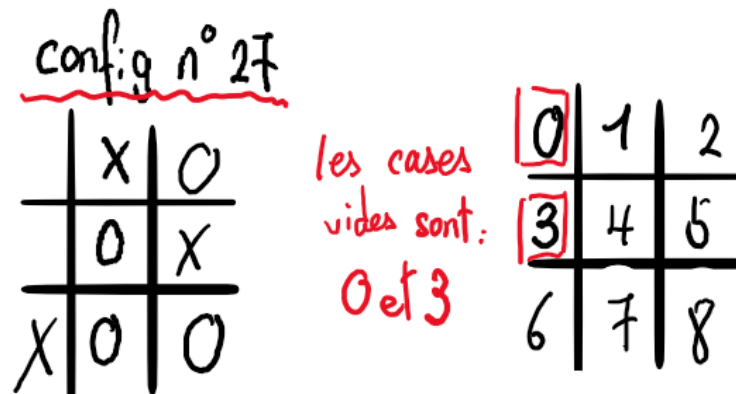
Pour commencer, je dois d'abord **charger dans un tableau les 304 configurations**(en base 10) à l'aide du fichier `b10Config.txt` et **initialiser les billes de chaque configuration** à l'aide du fichier `b3Config.txt`. (détecter les case vide en fonction des indices des 0 présentent dans la base 3) Voir schéma explicatif suivant.



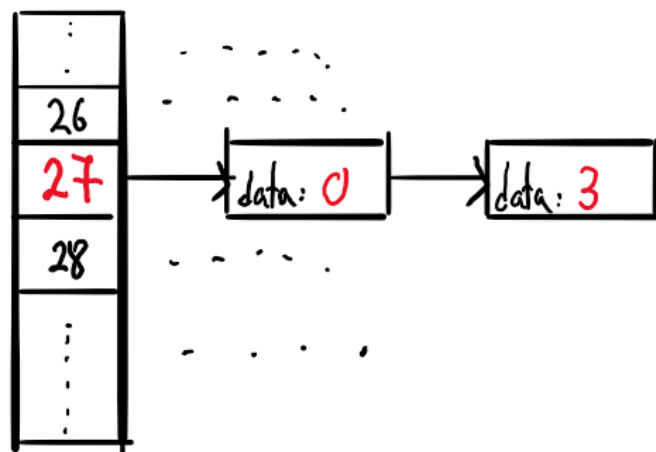
Pour le codage du jeu, j'ai suivi l'idée suivante :

- Pour le tour de l'IA, on retrouve tous les similaires de la grille actuelle (traduit en base 10) et recherche un par un dans le tableau des 304 configurations (qui est défini en haut) pour trouver celle qui lui correspond.

Puis, j'ai fait une fonction **random_bille()** qui tire au sort une bille aléatoirement de la configuration actuelle du jeu et en fonction de la bille tirée, on place le coup. (dans l'exemple suivant, on tire une bille aléatoirement cad soit 0, soit 3 et ce chiffre correspond à un emplacement valide de la configuration)



Donc dans la liste du config n° 27 :



Après le codage de la fonction AI vs Joueur (jeu de base) qui fonctionne seulement sur une game (sans la mise à jour des billes) pour tester si la détection de la configuration fonctionne correctement. Je remarque qu'il arrive à trouver la bonne configuration dans les 304, mais l'IA écrase chaque fois les cases où il y a déjà la présence d'un coup.

Après réflexion, j'ai compris que le problème est qu'une configuration a 7 autres similaires, mais chaque similaire ont des numéros des emplacements valides différents. (Voici un schéma d'explicatif)

| | | |
|---|---|---|
| X | 1 | 0 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

 $\xrightarrow{90^\circ}$

| | | |
|---|---|---|
| 0 | 1 | X |
| 3 | 4 | 5 |
| 6 | 7 | 0 |

— : les entiers qui correspond à l'emplacement vide de la grille actuelle.

Ici, on voit bien que ces deux configurations sont identiques à une rotation de 90° près, mais qu'ils n'ont pas les mêmes numéros des emplacements valide.

J'ai donc très vite changé l'idée et essayer de mettre les 304*8 configurations dans le tableau, mais si je mets les 304*8 configurations, cela veut dire que chaque fois je dois chercher tous ses similaires dans le tableau des configurations et modifier ses billes, autrement dit, jouer un coup est égal à parcourir 8 fois le tableau de 304*8 et modifier 8 fois la structure, qui n'est pas du tout optimal.

À cause de cela, j'ai repris mon idée de base avec les 304 configurations et essayer de trouver un autre moyen de résoudre ce problème.

Après des réflexions, j'ai remarqué qu'il faut aussi faire la rotation des emplacements des numéros des cases en même temps que les rotations de la grille.

| | | |
|---|---|---|
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

 $\xrightarrow{90^\circ}$

| | | |
|---|---|---|
| 6 | 3 | 0 |
| 7 | 4 | 1 |
| 8 | 5 | 2 |

| | | |
|---|---|---|
| X | 1 | 0 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

 $\xrightarrow{90^\circ}$

| | | |
|---|---|---|
| 6 | 3 | X |
| 7 | 4 | 1 |
| 8 | 5 | 0 |

Après cette modification, le jeu fonctionne correctement sans l'écrasement des cases.

Puis, lorsque je suis arrivé à la partie 4 du projet où indication nous dit que le jeu doit commencer aléatoirement soit par joueur, soit par l'IA.

Après la modification (en ajoutant une petite fonction qui renvoie 0 ou 1 aléatoirement), des nouveaux problèmes survient. Il s'agit des configurations introuvables dans mon tableau des 304 configurations, je me suis donc lancer dans la recherche pour obtenir plus de détails sur le projet MENACE et j'ai remarqué par surpris que pour le projet MENACE, c'est toujours l'IA qui commence en premier, autrement dit, dans les 304 configurations du projet MENACE existent seulement des configurations où l'IA commence en premier.

Je me suis alors décidé de recherche des configurations par moi-même en filtrant tous les cas dans une grille de 3x3.

Pour ce faire, je dois utiliser la fonction **next_configuration()** du fichier **aide_projet** qui est en 2d de base et la transforme en 1d pour générer toutes les configurations, puis, utiliser les fonctions **grille_valide()** et **end_game()** pour filtrer tous les cas impossibles et les cas gagnants, ensuite, à l'aide de la fonction **tab_similaire()** pour enlever tous les similaires et enfin, les écrire dans le fichier b10Config.txt (à l'aide du fichier b10Config.txt, on pourra aussi générer le fichier b3Config.txt).

❖ **Fonction next_configuration**, qui sert à générer toutes les configurations possibles du jeu.

```
uint8_t next_configuration(uint8_t grille[9])
{
    // incrémenter la première case de la grille
    grille[0]++;
    for (int i = 0; i < 9; i++)
    {
        // vérifier si l'incrémentatation dépasse 3, car les cas possibles sont seulement : 0, 1 et 2
        // avec 0 pour case vide, 1 pour x et 2 pour o.
        if (grille[i]==3)
        {
            // si ça dépasse on remet à 0.
            grille[i]=0;

            // et si on n'est pas à la dernière case, on incrémente la case suivante, etc.
            if (i<8)
            {
                grille[i+1]++;
            }
            else return 1;
        }
    }
    return 0;
}
```

- **uint8_t grille[9]** : un tableau contient 9 entiers, qui représente la grille du jeu.

- ❖ **Fonction grille_valide**, qui vérifie si un cas est valide ou non, return 1 si valide, sinon 0.

```
uint8_t grille_valide(uint8_t grille[9])
{
    uint8_t nbx=0, nbo=0, res=0;

    //compter nombre de x et o
    for (int i = 0; i < 9; i++)
    {
        // 1 == x et 2 == o
        if (grille[i]==1) {nbx++;}
        if (grille[i]==2) {nbo++;}
    }

    //comme x commence tout le temps donc on verifie si la diff entre o et x est égale à 0 ou 1
    if ((nbx == nbo || nbx==nbo+1 ))
    {
        res=1;
    }
    return res;
}
```

- **uint8_t grille[9]** : idem

- ❖ **Fonction create_b10Config**, permet de créer un fichier b10Config.txt qui contient toutes les configurations après le filtrage(suppression des similaires, cas impossible, etc.)

```
void create_b10Config()
{
    uint8_t g[9] = {0, 0, 0, 0, 0, 0, 0, 0, 0};

    FILE *out = fopen("b10config.txt", "w");

    uint8_t mini;
    uint32_t similaire[8]={0,0,0,0,0,0,0,0};

    // on écrit d'abord la configuration qui représente la grille vide.
    fprintf(out, "%i\n", grille_to_b10(g));
    printf("\n");
    while (next_configuration(g)==0 )
    {
```

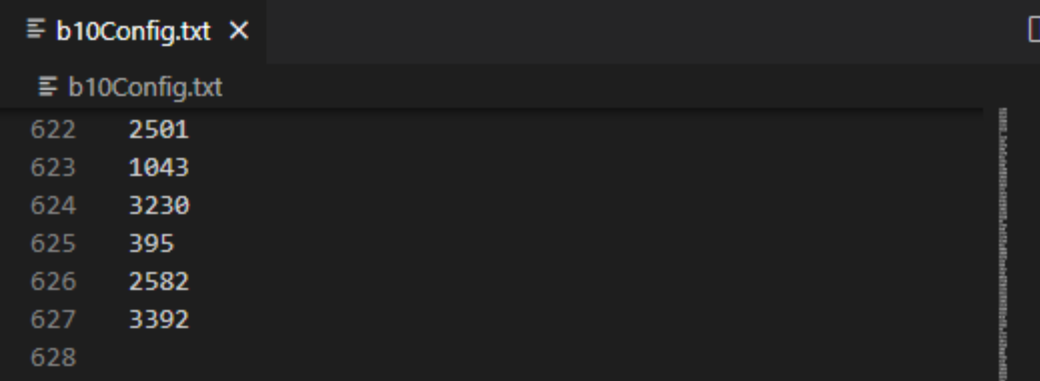
```

tab_similaire(grille_to_b10(g),similaire);

// trouver l'indice de la valeur mini
mini=min_tab(similaire);
// si le config actuelle n'est pas le plus petit de tous les similaires j'ignore
if (mini==0 )
{
    // si grille valide et game non fini
    if (grille_valide(g)==1 && end_game(g) == 0)
    {
        // print dans le fichier .txt
        fprintf(out,"%i\n",grille_to_b10(g));
    }
}
}
fclose(out);
}

```

Après cette manipulation, j'ai trouvé 627 configurations sans les similaires et le jeu n'affiche plus d'erreur sur configuration introuvable.

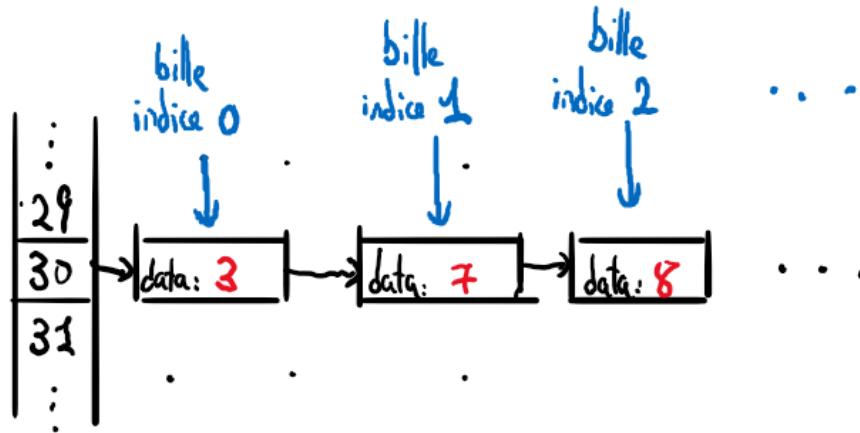


| Index | Value |
|-------|-------|
| 622 | 2501 |
| 623 | 1043 |
| 624 | 3230 |
| 625 | 395 |
| 626 | 2582 |
| 627 | 3392 |
| 628 | |

Une fois le problème des configurations est résolu, j'ai passé au codage du jeu version complet avec la mise à jour des billes à la fin de chaque patie.

Pour ce faire, je me suis base sur l'idée suivante :

1. On tire au sort une bille à l'aide de la fonction **random_bille()**, le numéro qu'on a tiré (correspond à l'indice de la bille d'une configuration) sera enregistré dans un tableau **indice_bille_joue**.



Exemple : `indice_bille_joue = [2, 3, 4]` pour pouvoir faire les modifications plus tard.

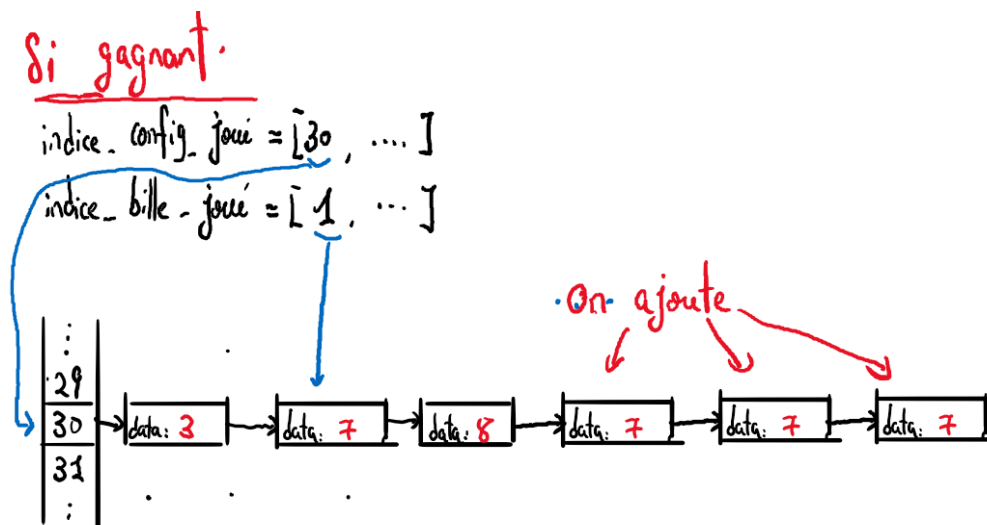
2. On enregistre les indices des configurations jouées durant le jeu de la même façon. Ici les numéros correspondent l'indice de la configuration dans le tableau des configurations défini avant.

Example : `indice_config_joue = [3,12,31]`. Configuration n°3, 12, 31, etc.

3. Mise à jour grâce aux deux tableaux précédents :

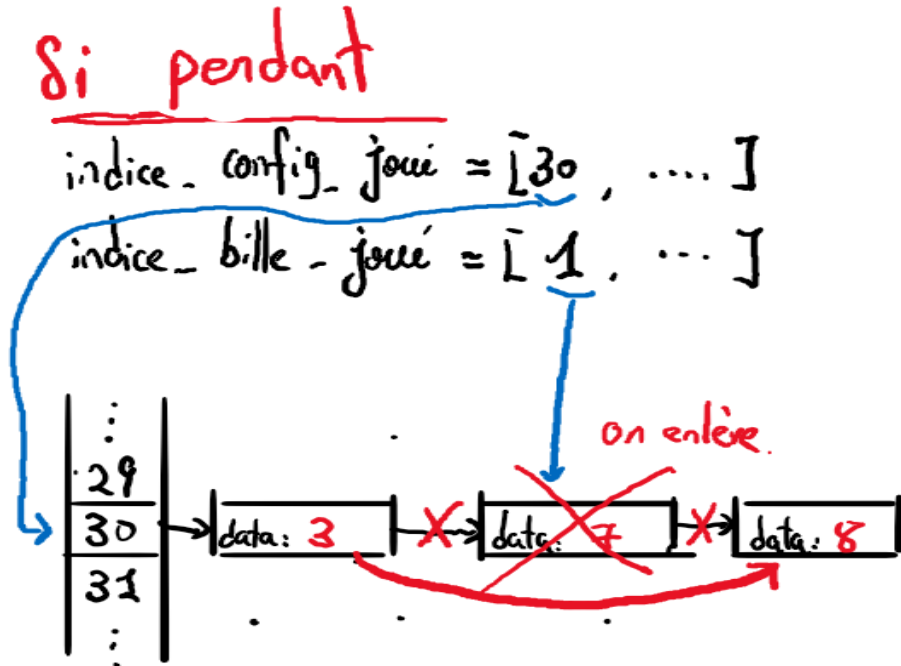
- S'il gagne ou nul :

En fonction des tableaux `indice_config_joue` et `indice_bille_joue`, on trouve la position de la bille dans la configuration correspondante, puis avec la fonction `data_bille_tabbox()` (qui retourne la data de la bille de la configuration) on ajoute dans la queue cette configuration trois billes(maillons) ou une bille de la même donnée.



- S'il perte :

item pour le début et on retire la bille avec la fonction **rem_position_tabbox()** qui retire la bille en fonction de l'indice de configuration et l'indice de la bille.



Voici les fonctions qui permettent de réaliser cette idée :

- ❖ Fonction **humain_ai()**, permet de lancer une partie en joueur contre IA(en raison de la longueur du code, ça ne sera pas possible de le mettre dans le rapport +de 100 lignes).
- ❖ Fonction **ai_ai()**, permet de lancer des parties IA contre IA en lui donnant un nombre de parties à faire pour objectif de la laisser s'entraîner automatiquement (en raison de la longueur du code, ça ne sera pas possible de le mettre dans le rapport +de 100 lignes).
- ❖ Fonction **maj_tab_box()**, permet de mettre à jour les données après une partie.

```
tab_box* maj_tab_box(tab_box *tb, uint8_t result, uint32_t* indice_bille_joue,
uint32_t* indice_config_joue, uint8_t nb_coup_ai)
{
    uint32_t i;
    // si gagnant
    if (result == 1)
    {
        // nb_coup_ai est le nombre de coup joué par AI dans une partie
        for (i = 0; i < nb_coup_ai; i++)
        {
```

```

        //on donne une condition que les bille ne dépasse 100 pour ne pas surcharger la
        box
        //on considere que 100 est suffisant
        if (tb->tab[indice_config_joue[i]]->taille <= 100)
        {
            // on ajoute 3 bille de dans le config correspondant
            add_queue_tabbox(tb, data_bille_tabbox(tb, indice_config_joue[i],
            indice_bille_joue[i]), indice_config_joue[i]);
            add_queue_tabbox(tb, data_bille_tabbox(tb, indice_config_joue[i],
            indice_bille_joue[i]), indice_config_joue[i]);
            add_queue_tabbox(tb, data_bille_tabbox(tb, indice_config_joue[i],
            indice_bille_joue[i]), indice_config_joue[i]);
        }
    }
}

// si nul
else if (result == 2)
{
    for ( i = 0; i < nb_coup_ai; i++)
    {
        if (tb->tab[indice_config_joue[i]]->taille <= 100)
        {
            // on ajoute 1 bille de dans le config correspondant
            add_queue_tabbox(tb, data_bille_tabbox(tb, indice_config_joue[i],
            indice_bille_joue[i]), indice_config_joue[i]);
        }
    }
}

//si perdant
else if (result == 0)
{
    for ( i = 0; i < nb_coup_ai; i++)
    {
        // enlever si box n'est pas vide pour éviter erreur lorsqu'on tire une bille random
        dans un box vide
        if (tb->tab[indice_config_joue[i]]->taille > 1)
        {
            // on retire 1 bille de dans le config correspondant
            rem_position_tabbox(tb, indice_config_joue[i], indice_bille_joue[i]);
        }
    }
}
return tb;
}

```

- **tab_box** *tb : table hachage (structure tab_box) qui contient toutes les configurations(liste).
- **uint8_t** result : un entier entre 0-2 avec 0 est perdu, 1 est gagné et 2 est nul
- **uint32_t*** indice_bille_joue : un tableau qui enregistre tous les indices des billes tirées au sort aléatoirement.
- **uint32_t*** indice_config_joue : un tableau qui enregistre tous les indices des configurations apparus durant le jeu.(qui est en lien avec tableau indice_bille_joue)
- **uint8_t** nb_coup_ai : nombre de coup joué par AI pour parcourir les 2 tableaux d'indice.

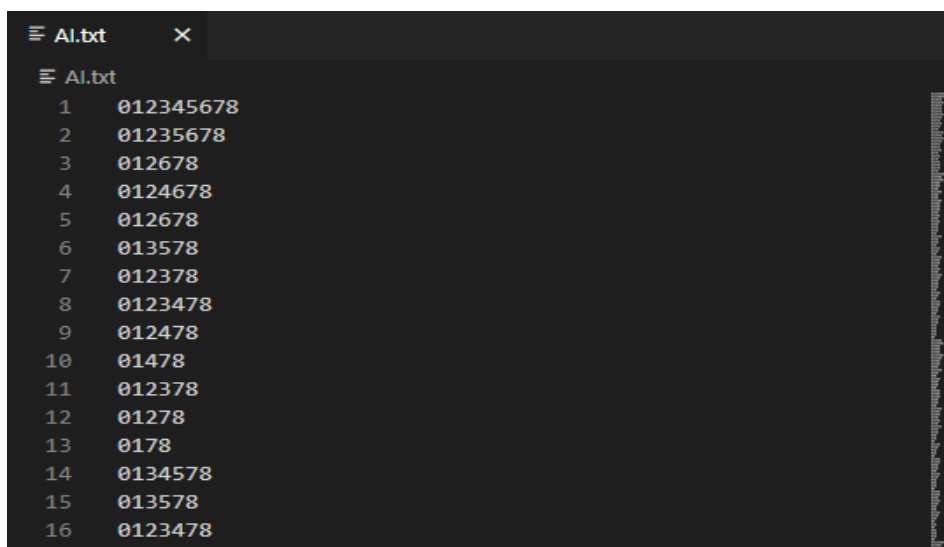
Comme on peut voir à travers le code **maj_tab_box()** que j'ai limité le nombre de billes à 100, car si on ne met pas de limite, cela posera sûrement des problèmes au moment de sauvegarde d'IA à cause d'une base de donnée trop chargée.

En plus de ça, j'ai mis une condition pour le cas perdant, on n'arrête d'enlever la bille si la "boite" possèdent plus qu'une bille, parce que sinon, pendant l'entraînement de l'IA (jouer avec lui-même), des erreurs de "floating point exception" qui vient de la fonction **random_bille()** car il ne peut pas tirer au sort une bille dans une boite vide. Cela me paraît très étrange, car normalement, si on enlève le chemin complet, on ne pourra donc plus retomber sur cette même configuration.

Pour résoudre ce problème, j'ai essayé de faire des modifications sur la fonction **maj_tab_box()**, cad, au lieu d'enlever toutes les billes du chemin perdant, j'enlève seulement la dernière bille et lorsque cette configuration ne possède plus de bille, j'enlèverai la bille de celui qui précède, etc. (l'idée est d'enlever les billes un par un du bas vers le haut) Mais malheureusement le problème persiste et cela est devenu alors un problème que je n'arrive pas à résoudre, j'ai donc gardé la solution d'avant.

Enfin, je me suis enchainé par le codage des fonctions sauvegarde.

L'idée que j'ai utilisé est simplement de parcourir toutes les configurations à travers la structure et d'écrire la donnée de chaque bille de la configuration sur une ligne, autrement, chaque ligne du fichier **AI.txt** représente une configuration et les entiers de cette ligne représente les données de ses billes(tous les emplacements valide de cette configuration). Et le nombre des entiers variera en fonction de son apprentissage.(ici, c'est l'initialisation du jeu)



```

AI.txt
1  012345678
2  01235678
3  012678
4  0124678
5  012678
6  013578
7  012378
8  0123478
9  012478
10 01478
11 012378
12 01278
13 0178
14 0134578
15 013578
16 0123478

```

❖ **Fonction save_AI**, permet de sauvegarder l'IA actuelle dans le fichier AI.txt

```
void save_AI(tab_box *tb)
{
    FILE *tab_bille = fopen("AI.txt", "w");
    bille *bi;

    //parcour toutes les configurations
    for (int j = 0; j < tb->taille; j++)
    {
        // bi prends la tete de la j configuration
        bi = tb->tab[j]->tete;
        //parcour toutes billes de cette configuration
        for (int i = 0; i < tb->tab[j]->taille; i++)
        {
            // ecrire dans AI.txt
            fprintf(tab_bille, "%i", bi->data);
            bi=bi->suivant;
        }
        //saut de ligne à la fin de chaque configuration
        fprintf(tab_bille, "\n");
    }
    free(bi);
    fclose(tab_bille);
}
```

- **tab_box** *tb: Structure de donnée (table hachage) qui contient les configurations.

Pour charger l'IA qu'on a déjà sauvegarde avant lorsqu'on lance le jeu, j'ai tout simplement fait une vérification sur l'existence du fichier AI.txt et la charger si cela existe. Et pour suppression, je supprime simplement le fichier AI.txt et détruit la structure entièrement.

❖ **Fonction load_AI**, permet de remplir la structure avec le fichier de sauvegarde "AI.txt".

```
void load_AI(tab_box *tb)
{
    FILE *load = fopen("AI.txt", "r");
    // ici la taille varier en fonction de la capacité d'un box
    // qu'on a déclarer dans la fonction maj_tab_box
    char box[200];

    fscanf(load, "%s", box);
    // i est le numéro de config ici taille_config == 627
```

```

for (int i = 0; i < taille_config; i++)
{
    // parcourir la ligne qu'on a lue dans AI.txt
    for (int j = 0; j < strlen(box); j++)
    {
        // on le transforme en int et l'ajouter dans chaque tab de notre tab hachage
        add_queue_tabbox(tb, box[j]%48, i);
    }

    fscanf(load, "%s", box);
}
fclose(load);
}

```

- **tab_box** *tb : Ici la structure est vide (qui vient de se créer sans rien à l'intérieur) pour pouvoir charger la sauvegarde.

Et pour finir, j'ai réalisé une interface avec des simples affichages sur le terminal pour faciliter l'utilisation.

```

-----MENU-----
1:Entraîner AI
2:Jouer une partie contre AI
3:Enregister AI
4:Supprimer AI
5:Quitter
-----

```


CONCLUSION

Tout d'abord, le projet peut être considéré comme terminé dans le sens de la fonctionnalité du programme et du respect du cahier de charge, mais, loin d'être réussi, car il y a encore la présence du problème du côté apprentissage(l'AI n'apprend parfois pas correctement) et il y aura surement la possibilité d'optimiser le programme.

Ensuite, je n'ai pas réussi à faire le bonus pour changer les règles du jeu ou bien modifier la taille de la grille, car dès le début, j'ai codé pour seulement du 3x3 donc pour réaliser ces fonctionnalités supplémentaires, il y a donc la nécessité de faire des grandes modifications sur certaines fonctions existante.

Enfin, ce projet me parait très compliqué et long au début, mais plus je m'approfondis, plus ça me donne envie d'aller au bout, de plus, je trouve que ce projet est très intéressant qui m'a permis de réfléchir toujours plus et m'oblige de relire le cours et faire des recherches, et c'est exactement cela qui m'a permis de bien mémoriser les différents moyens de stockage des données et notamment la gestion de la mémoire, etc.