

JIA Yunfei

INFO 1

Tuteur enseignant :

M.CHAUSSARD

Matière :

**Programmation Avancée –
Structures de données**



Rapport du projet

Morpion AI (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 traiter les données, c'est donc les structures de données.

Le projet consiste à réaliser un jeu de morpion où l'adversaire est un AI qui s'apprend à chaque partie et devenir petit à petit un joueur professionnel.

Tout d'abord, je présenterai globalement le sujet de ce projet.

Puis, c'est le choix de structure pour stocker les données, selon moi, il y a deux structures possibles que je vous présenterai dans la suite et ainsi le choix que j'ai effectué.

Ensuite, je présenterai toutes mes démarches de la réalisation notamment les recherches, beaucoup de problème rencontré, des remarques intéressantes, du résultat, etc.

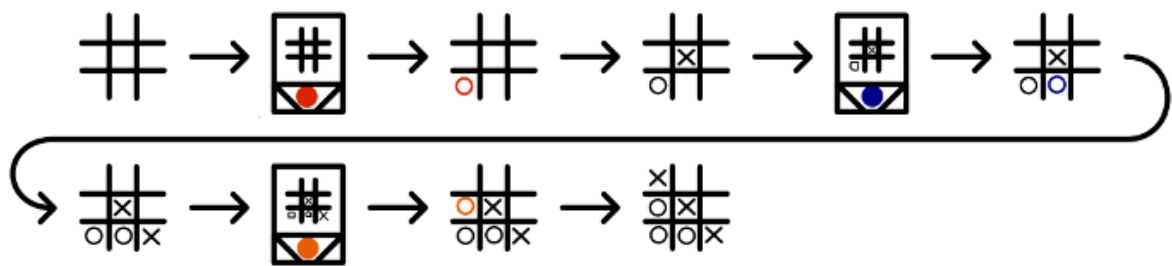
Enfin, je parlerai de ce que ce projet m'a apporté et mon avis personnel sur ce projet.

I / PRÉSENTATION DU PROJET

Ce projet consiste à coder un AI qui arrive à jouer au morpion et s'améliore en fonction du résultat de chaque partie pour devenir un joueur professionnel.

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 actuel et contient dans sa boîte des billes de couleur différente qui représente chacune un placement sur cette configuration.

Le déroulement de ce système est qu'on retrouve la configuration qui représente le jeu actuel et tire au sort une bille de la boîte puis jouer ce coup et répéter la même chose jusqu'à la fin du jeu.



Ici, le rond représente l'IA et en fonction du jeu actuel, tire au sort une bille et jouer 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 ce chemin gagne plus la bille de ce chemin auront une grande chance de tomber lorsque le même chemin reproduit.

II / PROGRESSION DU PROJET

A - Choix de la structure

Au début du projet, le grand problème est de savoir quelle sont les structures de données qui sont possibles et ainsi choisir celle qui est mieux adapter pour la mettre en pratique.

Pour cela, j'ai dû lister toutes les structures et finalement trier celle qui sont convenables dont la **liste chaînée** et **table hachage**.

L'idée qui correspondant a chacune des deux structures est :

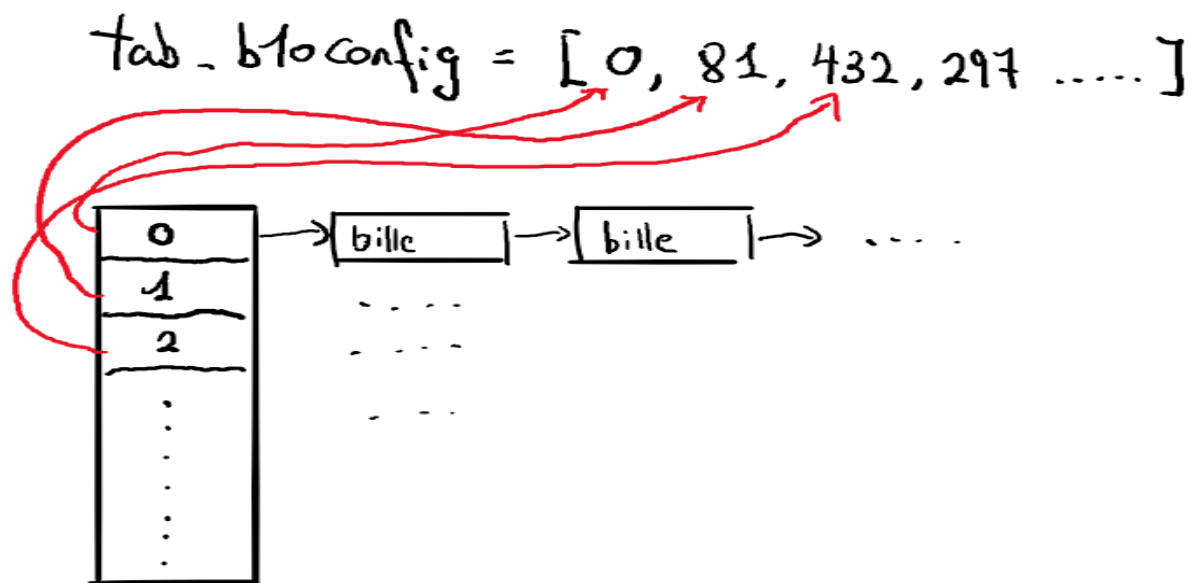
- Liste chaînée :

Une simple liste qui enregistre chaque configuration du jeu (après filtrage des cas impossible, etc.) dans un maillon.



- Table hachage :

Une table hachage qui enregistre toutes les billes de chaque configuration du jeu dans un maillon(data de chaque qui représente un emplacement de cette configuration) et en fonction d'un tableau de toutes les configurations en base 10, on pourra savoir quel indice de la table hachage correspond à quelle configuration.



J'ai choisi la structure de la table hachage qui me paraît plus simple à gérer les billes, cad lorsque l'IA a gagné, je peux directement ajouter des billes correspondant dans la liste de l'indice du tab_b10config en utilisant add_queue de cette liste.

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

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

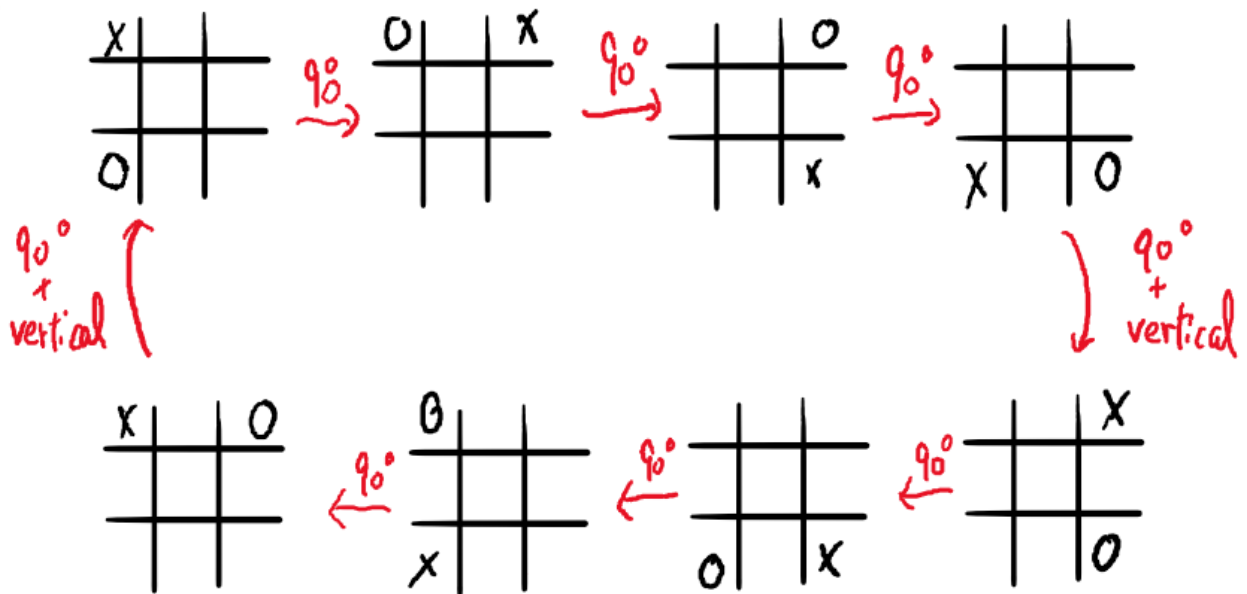
```
typedef struct
{
    box **tab;
    uint32_t taille;
}tab_box;
```

- **struct** bille : représente les maillons d'une liste
uint8_t data : représente l'emplacement dans une grille 0-8
- **struct** box : représente une liste d'une table de hachage qui contient des billes
- **struct** tab_box : représente une table hachage qui contient des listes où chaque liste représente une boite d'allumette.

B - Déroulement du projet

Tout d'abord, concernant les données (les configurations à chaque étape durant le jeu), après de nombreuse recherche sur internet du projet MENACE, je constate qu'il doit y avoir au moins 304 boites qui représentent tous les configurations (en faisant des rotations, on aura 7 autres configurations donc en total, on a 304*8 configurations).

Ces 304 configurations traduites en base 10 seront écrits dans un fichier **b10Config.txt** et aussi traduits en base 3 dans un fichier **b3Config.txt**.



Pour les datas que je dois enregistrer dans chaque bille (maillon) est représenté sous forme du chiffre 0-8 qui représente les 9 cases du plateau de jeu, car cela est plus simple à traiter que les emplacements 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 l'idée que je voulais suivre, j'ai passé au codage de "traduire" une grille 3x3 (pour moi elle est sous forme d'un tableau 1d ayant 9 valeurs) en base 10 et l'inversement.

- ❖ **Pour passer de grille en base 10**, j'ai directement calculé avec les entiers présentent dans la grille qui sont déjà en base 3 :

```
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]%48 *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**, j'ai utilisé la division successive puis inverser le tableau :

```
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 du jeu 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 de transformer une configuration de base 10 en grille puis faire les rotations).

Ensuite, par préférence et la simplicité, j'ai choisi de manipuler seulement du 1d pour les grilles, pour cela je suis obligé de modifier la fonction suivant du fichier **aide_projet** en 1d (en raison de la longueur énorme du code je ne la mettrai pas ici) :

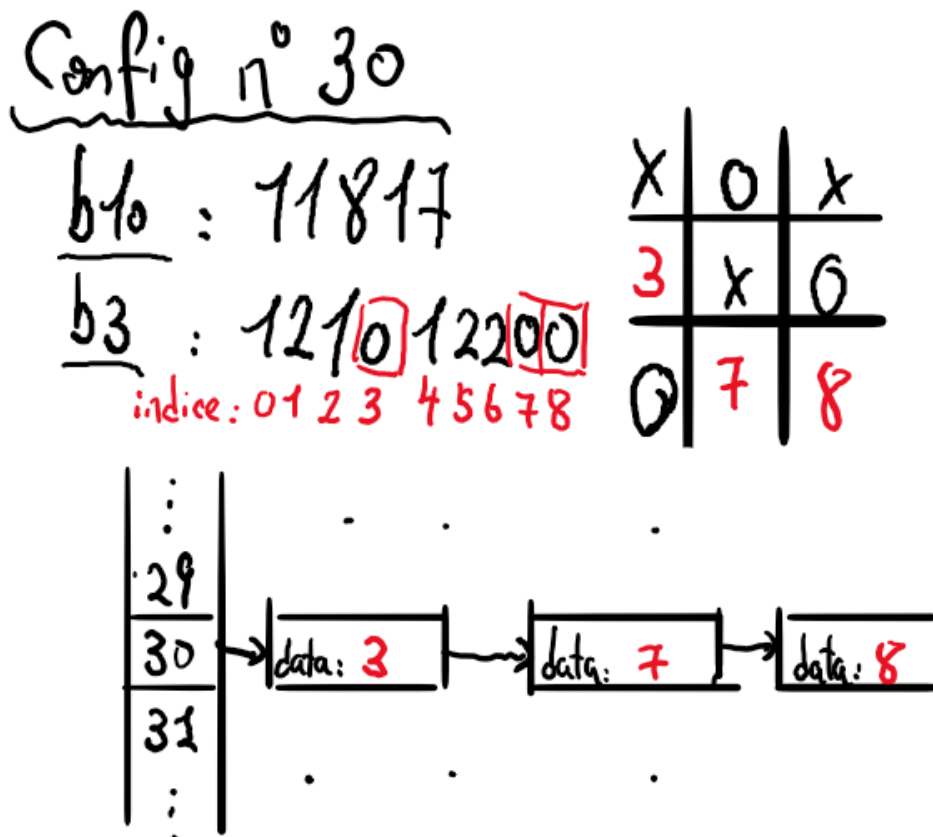
void appliquer_transformation_base(**uint8_t** grille[3][3], transformation tr)

et qui prennent maintenant des grilles de 9 entiers :

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

Une fois 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 tout d'abord les 304 configurations qui sont présentées dans le projet MENACE de base.

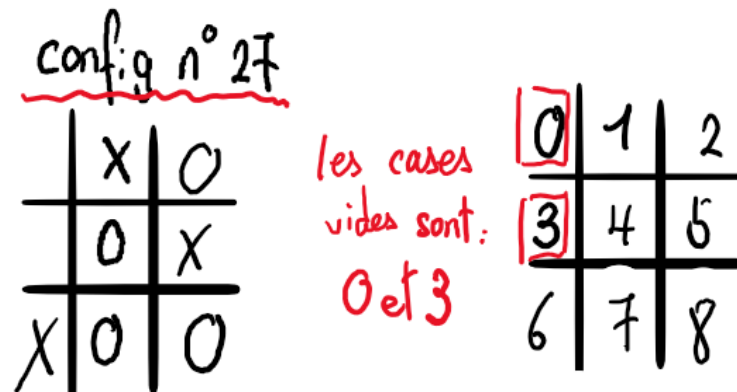
Pour commencer, je dois d'abord **charger dans un tableau les 304 configurations en base 10** à l'aide du fichier `b10Config.txt` et **initialiser les billes (la structure) de chaque configuration** à l'aide du fichier `b3Config.txt`. (détecter les cases vides en fonction des indices des 0 présents dans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



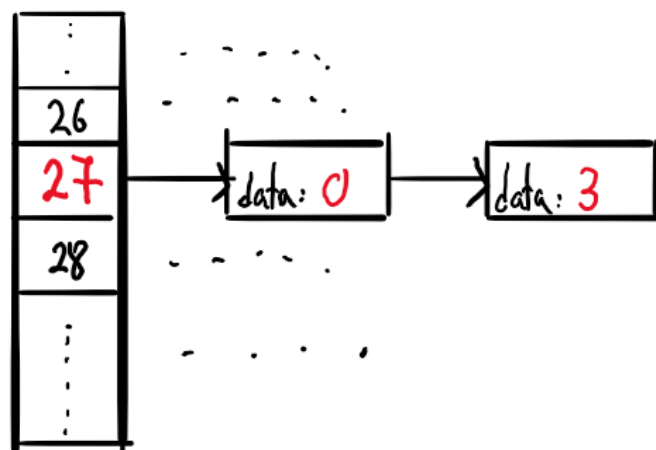
Pour le codage, j'ai suivi l'idée suivante :

- Durant le jeu, 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).

Et pour chaque coup de l'AI, j'ai fait une fonction **random_bille()** qui tire au sort une bille dans la liste de la configuration correspondant. (dans l'exemple suivant, on tire une bille au hasard et son data correspond à l'emplacement vide de la configuration)

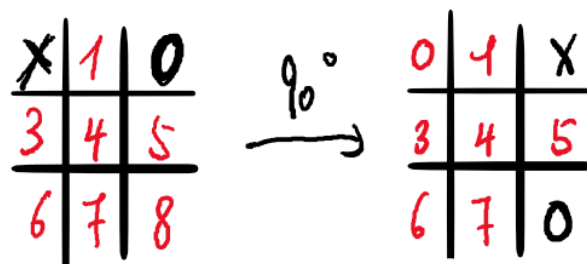


Donc dans la liste du config n° 27 :



Après le codage de la fonction AI vs Joueur (jeu de base) qui fonctionne seulement sur une game (cad sans la mise à jour des billes) pour tester si la détection de la configuration marche. Je remarque qu'il arrive à trouver la bonne configuration dans les 304, mais l'AI écrase chaque fois les cases qui sont non vide.

J'ai donc remarqué que le problème est qu'une configuration a 7 autres similaires, mais chaque similaire ont des emplacements des cases vides différentes. (Voici un schéma d'explicatif)

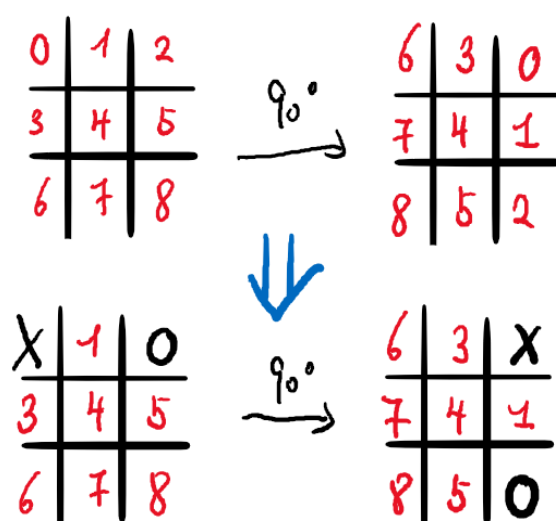


— : les entiers qui correspond à l'emplacement vide de la grille actuelle.

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

J'ai donc très vite changé l'idée et de mettre les 304*8 configurations dans le tableau, mais après réflexion, si je mets les 304*8 configurations, cela veut dire que chaque fois je dois chercher son similaire dans le tableau des configurations et modifier ses billes, autrement dit, jouer un coup est égal 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 dû reprendre mon première idée avec les 304 configurations et après de nombreuse réflexion, j'ai remarque qu'il faut aussi faire la rotation des emplacements des cases vide en même temps que les rotations de la grille.



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

Puis, lorsque je suis arrivé à la partie 4 du projet qui a indiqué que le jeu doit commencer aléatoirement par joueur ou AI.

Après la modification, des nouveaux problèmes survient. Il s'agit des configurations introuvables dans mon tableau de 304 configurations, je me suis donc lancer dans la recherche des détails sur le projet MENACE et j'ai remarqué que dans le projet MENACE, c'est toujours l'AI qui commence en premier, autrement dit, dans les 304 configurations du projet MENACE existent seulement des configurations où l'AI commence en premier.

Je me suis alors décidé de me lancer dans la recherche des configurations moi-même.

Pour ce faire, je dois utiliser la fonction **next_configuration()** du fichier **aide_projet** et la transforme en 1d pour générer toutes les configurations du jeu, puis, utiliser les fonctions **grille_valide()** et **end_game()** pour filtrer tous les cas impossibles et les cas gagnants, en fin, à l'aide de la fonction **tab_similaire()** pour enlever tous les similaires et 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 ce n'est pas un cas valide, 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 du jeu 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 )
```

```

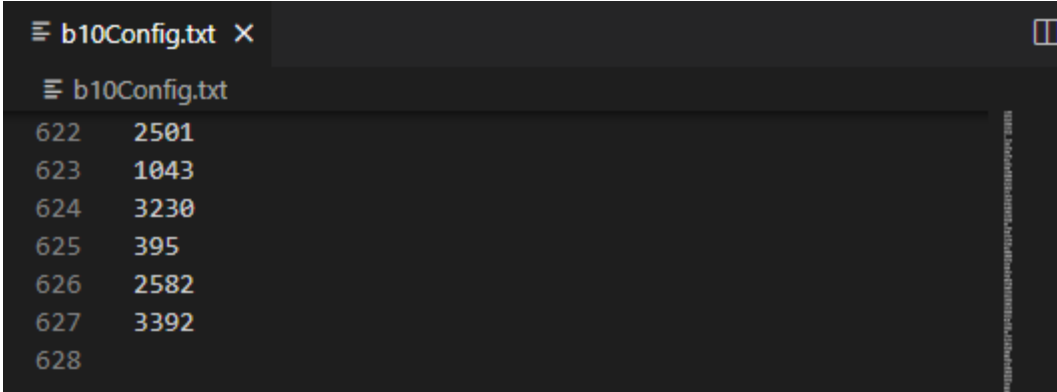
{

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.



```

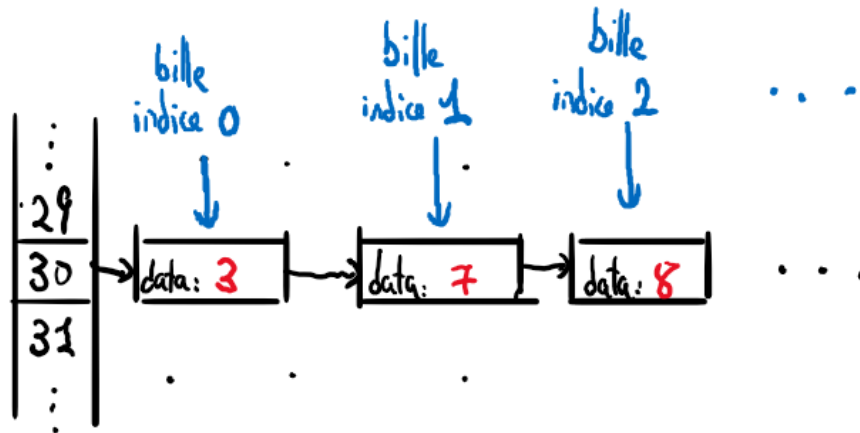
b10Config.txt
622 2501
623 1043
624 3230
625 395
626 2582
627 3392
628

```

Une fois le problème des configurations est résolu, je commence le codage du vrai jeu qui se mettre à jour après chaque game.

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

1. On tire au sort à l'aide de la fonction **random_bille()**, et en fonction du numéro qu'on a tiré (correspond à l'indice de la bille dans la liste de la structure), on cherche dans box (structure box) la bille pour laquelle a comme indice le numéro qu'on a tiré au sort et enregistrer 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.

Exemple : **indice_config_joue** = [3, 12, 31].

3. Mise à jour grâce aux 2 tableaux précédents :

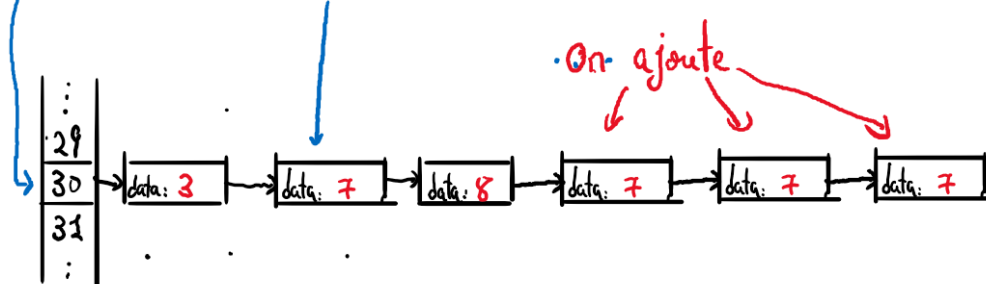
- S'il gagne ou nul :

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, enfin, ajouter dans la queue de la configuration (liste) 3 billes (maillons) ou 1 bille de la même data.

Si gagnant.

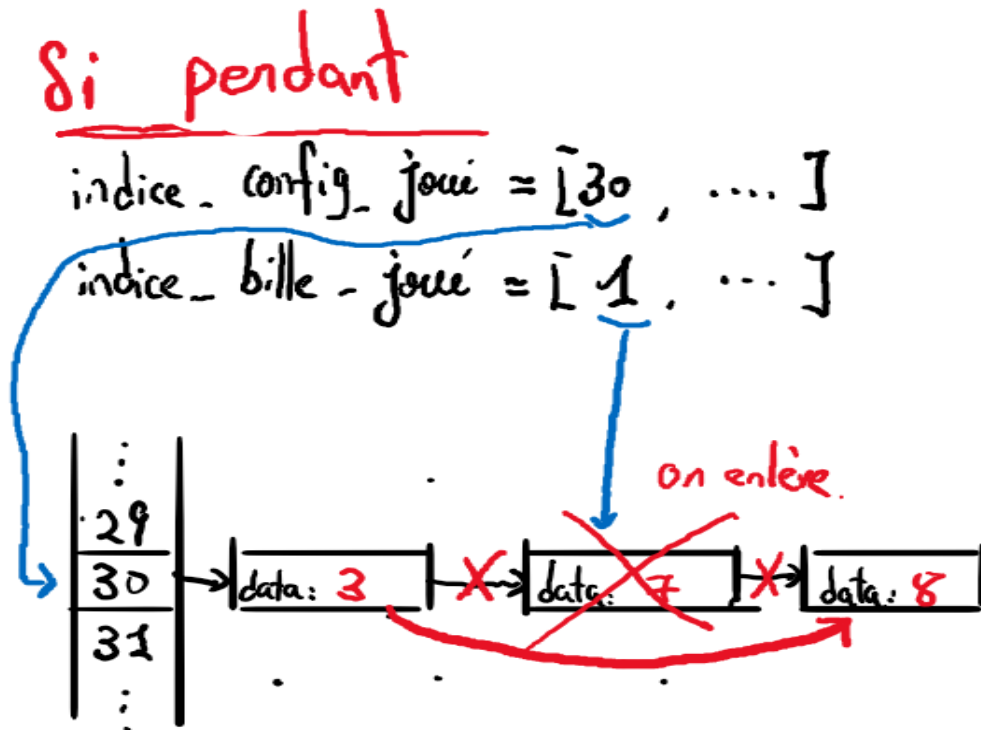
indice_config_joue = [30,]

indice_bille_joue = [1, ...]



- S'il y a une 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.



- ❖ Fonction **humain_ai()**, permet de lancer une partie en joueur contre AI (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 une partie AI contre AI en lui donnant un nombre de parties à faire pour objectif de le 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()**, vous remarquer surement que j'ai limité le nombre de billes en 100, car si on ne met pas de limite, au moment de sauvegarde d'AI posera surement des problèmes à cause d'une base de donnée qui est trop chargé.

En plus de ça, j'ai aussi mis pour cas perdant qu'on enlève la bille seulement si la "boite" possèdent plus d'une bille, parce que sinon, pendant l'entraînement de l'AI (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, ce qui est très étrange, car normalement, si on enlève le chemin complètement, 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 celui de la configuration jouée qui précède de cette configuration vide, 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.

Enfin, lorsque j'ai terminé le jeu, j'ai passe au codage des fonctions sauvegarde. L'idée que j'ai utilisé est tout simplement de parcourir toutes les configurations à travers la structure et écrire la data 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 la data de toutes ses billes(toutes les cases possibles de cette configuration). Et le nombre des entiers varie 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
17 012478

```

- ❖ **Fonction save_AI**, permet de sauvegarder AI actuel en écrivant toutes ses données dans un 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 et leurs billes

Pour charger le AI qu'on a sauvegarde à chaque relance du 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.

- ❖ **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 déclaré 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).

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, je peux dire que le projet est 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 n'est pas devenu "joueur professionnel" après l'entraînement) et 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 comme j'ai commencé pour seulement du 3x3, il y a donc la nécessité de modifier beaucoup de fonction existante.

Enfin, ce projet me paraît très compliqué et long au début, mais plus je m'approfondis plus ça me donne envie d'aller au bout, 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 ces connaissances sur les différents moyens de stocker les données et notamment la gestion de la mémoire, etc.