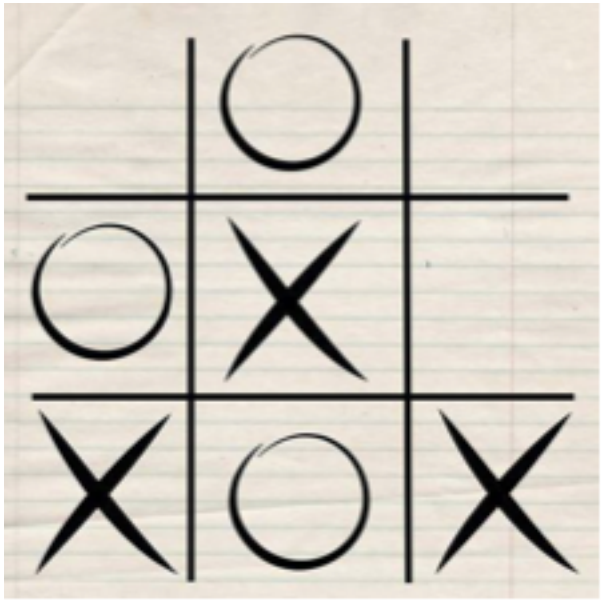
ECL – 2A - S7

2022-2023

Stratégies de Résolution de Problèmes

BE 2

Le jeu Morpion



I Plan

- Exposé et détails du Jeu Morpion
- Réalisation d'une version simple
- Introduction d'une fonction d'évaluation
- Réalisation de la version optimisée avec Heuristique Min-Max (et Alpha-Beta)
- Passage à une interface graphique (bonus)

• Ce BE nous permet d'aborder une stratégie de gain (**Best First** : choix du meilleur coupe local ou global) via différentes méthodes (fonction cout, Min-Max, Alpha-Beta...)

II Le jeu Morpion (TIC-TAC-TOE)

Le principe du jeu en quelques mots :

- Un plateau (grille, échiquier) carré de taille **Dimension** > 2.
- Un nombre égal à **Dimension** pions Noirs pour le joueur 1, autant de Blancs pour le joueur 2
- Départ : plateau vide
- But : un des joueurs arrive à placer ses pions sur une même ligne, colonne ou diagonale et gagne.

II.1 Le jeu (Cas Dimension = 3)

Notre jeu *Morpion* est composé d'une grille 3 x 3 et de deux joueurs chacun disposant de 3 pions de couleur noire (ou blanche).

Un des joueurs gagne à ce jeu s'il réussit à placer 3 pions de la même couleur (la sienne) sur une des 3 lignes, sur une des 3 colonnes ou sur une des 2 diagonales.

Déroulement du jeu : lorsque le jeu commence (sans aucun pion sur la grille), l'un des joueurs (soit N=noirs) place un pion noir, puis l'autre (B=blancs) place un pion blanc.
Lorsque les 3 pions blancs et les 3 noirs sont placés, si aucun des joueurs n'a gagné, chaque joueur à son tour déplace un des ses pions sur une case vide.

Ainsi, le joueur aura **permuté** un de ses pions avec une case vide. Cette interprétation simplifiera la mise en oeuvre de l'algorithme de ce jeu.

Le jeux va de cette manière se dérouler jusqu'à trouver un **gagnant**.

La figure 1 ci-contre montre un état non-terminal (sans gagnant) du jeu.

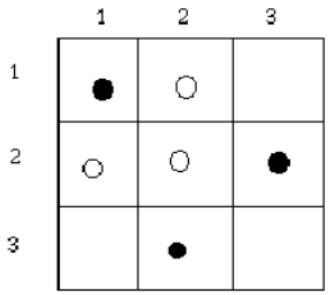


figure 1

Jeu déclaré nul : il arrive dans ce jeu qu'aucun des joueurs ne puisse gagner (répétition de permutation d'une case couleur avec une case vide qui revient sans cesse à un état précédent). Dans ce cas, après **k** tours (par exemple, k=3) pour chacun, si la grille se retrouve dans un état précédent, le résultat est déclaré nul et on recommence depuis le début. Ce cas sera traité plus tard dans les versions élaborées.

Pour éviter le cas d'égalité, il est possible de calculer la valeur des pions de chaque joueur en fonction des positions de ses pions (par exemple le centre valant plus que les autres cases) pour déclarer un gagnant.

II.2 Aspects pratiques

La figure 2 ci-contre représente une numérotation de la grille par le couple **Ligne/Colonne** (noté (L, C)). Chaque pion placé (ou case vide) aura une coordonnée de la forme L/C.

Du point de vu pratique (Python), le couple L/C peut être représenté par une *paire* (L, C) ou par une liste à 2 éléments. Privilégiez plutôt la structure de données "tuple" de Python. Dans le reste de ce document, on conserve la notation L/C. Voir section III page 5.

1/1	1/2	1/3
2/1	2/2	2/3
3/1	3/2	3/3

figure2

Une variante du jeu : dans une version de débutant et pour **Dimension** > 3, on accepte la fin d'une partie (état de gain) si l'un des joueurs a placé *Dimension -1* pions dans les mêmes dispositions que ci-dessus. Par exemple, pour une Dimension=4, un joueur aura gagné s'il a placé 3 pions sur une même ligne, une même colonne ou une même diagonale. Voir la section XVI.1 en page 28

☞ Comme pour tous les BEs, il est **important** de suivre une méthode et ne pas simplement "essayer à tous prix de faire marcher" votre code.

- Le but de ces BEs est d'étudier, critiquer et apprendre des algorithmes.
- Utiliser les algorithmes proposés tout en restant critiques ; ne les changez que si le votre est plus simple et/ou plus efficace tout en restant lisible et compréhensible. Évitez de **bricoler** !

II.3 L'algorithme de principe du jeu

Il est important d'apprendre comment mettre en place un algorithme simple et clair. Vous avez une proposition ci-dessous. Ne bidouillez pas, please ! Si vous réfléchissez à votre propre solution, revenez à cet algorithme pour comparer.

```
Procédure jouer :
  Entrées : Grille G ;
  Sortie : rien

Début
  Choisir une couleur Color parmi {B, N} ;
  bool Fini=faux ;
  Répéter jusqu'à (Fini = vrai)
    placer_un_pion(G, Color)          // C.à.d. placer un pion ou permuter un pion avec une case vide
    Fini ← gagnant(G, Color) ;
    Si (Fini = Faux)
        Alors Color ← l'autre_couleur(Color) ; // C-à-d. si (Color=B) alors Color ← N sinon Color ← B
finsi
    Finsi ;
  Fin Répéter
  Color est le gagnant
Fin jouer
```

N.B. : une *procédure* correspond à une *action ;* elle est assimilée à une *fonction* qui ne renvoie rien (rien = None en Python).

La fonction **gagnant** ci-dessous permet de savoir (par un booléen) si la grille actuelle G est un état gagnant pour la couleur *Color* :

```
Fonction gagnant : booléen
  Entrées : Grille G, Couleur Color          // Color représente un des 2 joueurs N,B
  Sortie : un booléen                        // la Couleur Color gagne-t-elle dans la Grille G

Début
  S'il y a une ligne/colonne/diagonale de Couleur Color sur la grille G
  Alors renvoyer Vrai              // succès = gagnant
  Sinon
    Si l'état de la grille a été rencontré il y a 3 tours
    Alors finir le jeu             // terminer brutalement !
    Sinon renvoyer Faux    // échec = non gagnant
    Fin si
  Fin si
Fin gagnant
```

L'action de **placer_un_pion** place un pion de couleur *Color* sur la Grille G :

```
Procédure placer_un_pion :
  Entrées : Grille G , Couleur Color ;
  Sortie : rien

  S'il y a déjà 3 pions de Couleur Color sur la grille G
  Alors retirer un pion Color de la Grille G
  Fin si
  Positionner un nouveau pion de Couleur Color sur la grille G
Fin placer_un_pion
```

☞ L'intérêt de ce découpage est qu'on peut implanter une stratégie dans l'action de placer un pion : que ce soit un simple placement ou un retrait suivi d'un placement.

N.B. : La fonction **placer_un_pion** ci-dessus est neutre. Vous serez amené à tester la couleur (Color) et d'adapter un comportement différent selon que *Color* correspond à la machine et/ou à l'humain.
➛ Par exemple, quand ce sera à l'humain (vous) de jouer, vous ne ferez peut-être pas un tirage aléatoire pour choisir votre prochain coup.

III Représentation de la grille

L'efficacité de nos algorithmes dépend des structures de données choisies. A ce titre, un élément important est *le choix de la prochaine case à jouer*.

On remarque également des appels fréquents à la procédure **gagnant**. Elle mérite d'être optimisée.

La section suivante décrit les structures de données qui tiennent compte de ces considérations.

III.1 Représentation des cases et placements

Étudions le choix de la prochaine case à jouer (l'objet de la procédure **placer_un_pion**) qui peut représenter un coût important et nécessite d'être optimisé par un choix judicieux des structures de données.

On peut représenter la grille par 3 vecteurs (ou listes) :

VB : vecteur des blancs placés (Blanc=B)
VN : vecteur des noirs (N) placés
VV : vecteur des cases vides (cases non occupées)

Supposons que la **Dimension**=3.
Nous adoptons la représentation de chaque case par un couple $(Ligne/Colonne)$ (figure 2 ci-dessous).

Au départ, VB = VN = vide et le vecteur VV contient les 9 cases de la grille = [1/1, 1/2, ... , 3/3].

Si le joueur B (Blanc) doit jouer, il prendra une des cases vides de VV qu'il insère dans VB. Le joueur N fera de même.
Lorsque chaque joueur aura placé ses 3 pions, les 3 vecteurs VB, VN et VV contiennent chacun 3 éléments.

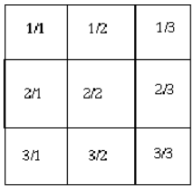


figure2

Un exemple : pour la figure 1 (rappelée ci-contre), le contenu des 3 listes sera :

VN=[1/1, 2/3, 3/2] le vecteur VN pour les Noirs
VB=[1/2, 2/1, 2/2] pour les Blancs et
VV=[1/3, 3/1, 3/3] pour les cases vides.

N.B. : sous Python, utiliser un couple **(a,b)** pour la case **a/b**.

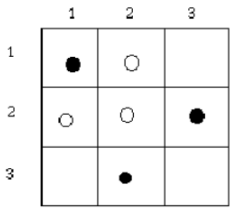


figure 1

A partir de ce moment dans le jeu (chacun a placé 3 pions), chaque joueur **permutera** un des éléments de son vecteur (VB pour les Blancs, VN pour les Noirs) avec un des éléments de VV. Le non déterminisme du jeu viendra des choix des pions ou des cases à placer/permuter.

IV Travail à rendre 1

Réalisez cette première version et faites jouer la machine vs. la machine (d'abord).
Pour faire jouer la machine contre la machine, faites la jouer bêtement (d'abord)! A chaque fois que vous avez un choix (choix d'un pion à retirer, choix d'une case vide, etc.), utilisez un tirage aléatoire.

Modifier ensuite votre réalisation pour faire jouer Machine vs. Humain.

V A propos du test 'gagnant' pour la Couleur Color

On peut envisager de tester un état gagnant (**fonction gagnant**) de plusieurs manières :

I- D'une manière plus classique : pour tester si la grille représentée par les 3 vecteurs correspond au gain des Color (par exemple, les B), il suffit de tester le contenu du VB. Ci-dessous, une même lettre qui se répète veut dire la même valeur.

VB = $[(L_,), (L_,), (L_,)]$; // une même ligne L$\in$ 1..3 gagnante ('_' veut dire n'importe)
VB = $[(,C), (,C), (,C)]$; // une colonne C$\in$ 1..3 gagnante ('_' veut dire n'importe)
VB = $[(L_1, C_1), (L_2, C_2), ((L_3, C_3)]$; // première diagonale gagnante (3 couples avec $L_i = C_i, i = 1..3$)
VB = $[(L_1, C_1), (L_2, C_2), ((L_3, C_3)]$. // 2e diagonale gagnante (L_i et C_j différents, voir ci-dessous)

P. ex., le joueur B possède la 2e diagonale si VB contient les 3 couples 1/3, 2/2 et 3/1 (au sens ensembliste).

Attention : ces vecteurs ne sont pas *a priori* ordonnés. Donc, dans le cas spécifique du test de la 2nde diagonale, un test d'égalité de VB avec le vecteur [1/3, 2/2, 3/1] dans cet ordre peut échouer car VB peut contenir les mêmes 3 cases dans un ordre différent (par exemple, dans l'ordre [3/1, 2/2, 1/3]). Point à surveiller :

Une première solution est de tester, pour ce dernier cas, l'appartenance des cases 1/3, 2/2 et 3/1 à VB à l'aide d'une fonction de test d'appartenance. L'autre solution est d'ordonner VB sur les numéros de ligne (L_i).

II- Mémorisation du dernier placement :

On peut accélérer le test du gain (la fonction **gagnant**) en mémorisant la case qui a été jouée en dernier. Ce qui permet de diminuer le nombre de tests.
Par exemple, si un jouer B a joué la case 3/3 en dernier, il suffit seulement de vérifier si la ligne 3, la colonne 3 ou la première diagonale contiennent 3 pions B.
➙ Ainsi, les tests sur d'autres lignes / colonnes / diagonale seront évités.

III- Par le carré magique : la figure 3 représente un carré magique pour la Dimension= 3 où la somme de chaque ligne=somme de chaque colonne=somme de chaque diagonale=15.
◦ Cette notation permet d'identifier rapidement un gagnant.
◦ Il suffit de tester si les pions d'une Couleur Color sont placés dans des cases dont la
somme des valeurs dans le carré magique donnent une somme = 15.

2	7	6
9	5	1
4	3	8

figure 3

Vous trouverez plus loin les carrés magiques pour les autres dimensions d'échiquier.

☞ Évitez dans tous les cas un traitement lourd car cette fonction est appelée très souvent.
Pour cela, modifier l'algorithme *placer_un_pion* par :

Procédure placer_un_pion :
Entrées : Grille G , Couleur Color ;
Sortie : rien

Si Color = couleur_humain()
Alors **placer_un_pion_humain**(Color, G)
Sinon **placer_un_pion_machine**(Color, G)
Fin si
Fin placer_un_pion

L'action de **placer_un_pion_machine** utilisera (pour l'instant) les tirages aléatoires. Elle sera bientôt munie d'une intelligence !
En attendant, si vous retirez un pion en (L, C), évitez de le remettre sur la même case (L, C) !

Procédure placer_un_pion_machine :
Entrées : Grille G , Couleur Color ;
Sortie : rien

S'il y a déjà 3 pions de Couleur Color sur la grille G
Alors retirer aléatoirement un pion Color de la Grille G
Fin si
Positionner aléatoirement un nouveau pion de Couleur Color sur la grille G
Fin placer_un_pion_machine

Pour l'humain, on lira les coordonnées des cases au clavier.
☞ Si vous retirez un pion en (L, C), évitez de le remettre sur la même case (L, C) !

Procédure placer_un_pion_humain :
Entrées : Grille G , Couleur Color ;
Sortie : rien

S'il y a déjà 3 pions de Couleur Color sur la grille G
Alors
    Lire au clavier les coordonnées ($Ligne/Colonne$) d'une case à libérer
    Contrôler et libérer la case $Ligne/Colonne$
Fin si
Lire au clavier les coordonnées ($Ligne/Colonne$) d'une case à occuper
Positionner un nouveau pion de Couleur Color en $Ligne/Colonne$
Fin placer_un_pion_humain

Pour la lecture au clavier (rappel du BE-1) :

```
def Lire_une_case() :
    """ Lecture de la case (sur la mm ligne) en commençant à 1 (avec controle)"""
    while (True) :
        print("Donner x puis y (sur la mm ligne, en commençant à 1) ? ",end=' ')
        try :
            x,y = input().split()    # 'input' ne peut lire que des chaines
            x,y=int(x),int(y)        # On convertie
            if x in {1,2,3} and y in {1,2,3} : pass
        except :  # L'exception / erreur est passée ici
            continue
        break
    return (x,y)
```

☞ N'oubliez pas ensuite de vérifier qu'une case à libérer/occuper appartient bien à la couleur adéquate !

☞ **Avant d'aller plus loin, lire la section 'A propos' suivante.**

VI A propos

VI.1 Déterminisme et le Hasard

En utilisant les tirages aléatoires (avec le même *seed*), votre programme donnera les mêmes résultats si le choix d'une case initiale à jouer est toujours le même. Ce qui empêche de tester tous les cas.

Pour introduire un degré de hasard, on pourra disposer d'un choix initial aléatoire.

Le plus simple est le suivant : au lieu d'initialiser VV à [1/1, 1/2, ..., 3/3] toujours dans le même ordre, on pourra introduire un choix aléatoire qui initialise VV dans un ordre aléatoire (9 valeurs, 9 ! possibilités). VV n'étant pas le même à chaque exécution, les résultats devraient être différents.

VI.2 Complexité du problème

Cette complexité est assez négligeable car on sait que $Dimension$ ne sera jamais très grand. Vérifier que ce jeu est de complexité $O((Dimension^2)!)$, pour *Dimension* pions pour chaque joueur.

VI.3 Différents carrés magiques

Pour une taille 3, on pourra utiliser le carré magique de la figure 3 ci-dessus.
- Pour Dimension=3, la somme magique est 15,
- Pour la dimension 4, elle est 34,
- Et somme=65 pour Dimension=5,
- Et somme=111 pour Dimension=6,

Les différents carrés magiques (pour les dimensions 3 à 6) :

Pour Dimension = 3, somme=15 :

2	7	6
9	5	1
4	3	8

Pour Dimension = 4, somme = 34 :

1	2	15	16
12	14	3	5
13	7	10	4
8	11	6	9

Pour Dimension = 5, somme=65 :

1	2	13	24	25
3	23	17	6	16
20	21	11	8	5
22	4	14	18	7
19	15	10	9	12

Pour Dimension = 6, somme = 111 :

1	2	3	34	35	36
4	18	28	29	5	27
10	26	30	8	23	14
31	25	13	21	15	6
32	16	20	12	22	9
33	24	17	7	11	19

☞ il faut noter que pour une taille $N > 3$, les tests seuls des valeurs du carré magique spécifique à N ne suffisent pas. On aura aussi besoin des carrées de taille inférieures.

Par exemple, pour la taille 4, il est possible que la somme (34) soit atteinte sans que l'on soit dans un état gagnant. On note que dans le carré magique pour Dimension=4 ci-dessus, les cases 2/1, 3/1, 1/2 et 3/2 donnent aussi la somme 34 sans être alignées (sans être gagnant).

➙ Pour être gagnant, il faut que 3 parmi les 4 pions d'une même couleur soient **alignés** avant de vérifier que la somme convient.

Pour y parvenir, il y a plusieurs solutions (cas Dimension = 4) :

Solution I- vérifier que 3 des 4 pions d'une Couleur Color $(L_1/C_1, L_2/C_2, L_3/C_3, L_4/C_4)$ sont alignées en testant :

1. au moins trois des lignes $L_i, i = 1..4$ sont identiques
2. au moins trois des colonnes $C_j, j = 1..4$ sont identiques
3. au moins dans trois des couples $L_i/C_j, i, j = 1..4$, $L_i = C_j$ sont identiques (1e diagonale)
4. les cases de la Couleur Color sont parmi $\{1/4, 2/3, 3/2, 4/1\}$ (2e diagonale)

Solution II- utiliser le carré magique de la taille 3 permettant d'imposer que 3 des 4 pions soient déjà alignés (sur une ligne/colonne ou diagonale) avant de tester l'état du 4^e pion. De même, pour Dimension=5, on a besoin du carrée magique de taille 4.

Pour la suite de ce document, on suppose la représentation par 3 vecteurs VB, VN et VV.

VI.4 Trucs pour bien jouer quand Dimension = 3

Vu sur la toile (chaque joueur a 4 pions en main ! Vérifier pour Dimension=3 et faites profiter tout le monde) :

- si B joue au centre, il est sûr de ne pas perdre.
- si N joue ensuite dans la croix (du signe '+'), il perdra.
- si N joue dans une diagonale, vous devez jouer dans la même diagonale !

VII Travail à rendre 2

Modifier votre réalisation précédente et réaliser une version graphique (*tkinter*).
☞ Vous pouvez permuter l'ordre Travail-2 et Travail-3.

☞ Un fichier de création d'interface Tk est fourni.

VIII Heuristique Best-First de base

L'algorithme du jeu de Morpion est un algorithme **irrévocable** : on ne revient pas sur un placement pour en tenter un autre.

VIII.1 Déplacement intelligents

Nous pourrions envisager une stratégie (intelligente) en favorisant les les lignes, colonnes ou diagonales où
- nous avons déjà deux pions et une case libre.
- l'adversaire n'a qu'un pion et deux cases libres
- toute autre combinaisons (favoriser le centre, les coins, etc.)

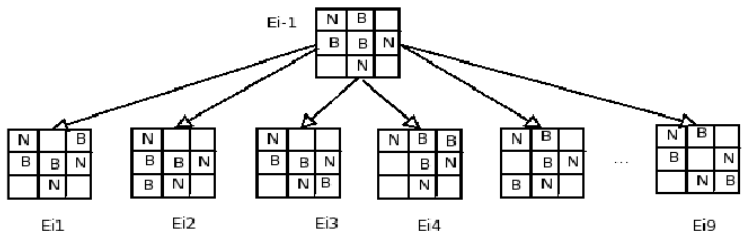
Ce types de stratégie (*algorithmique* et non *équationnelle*) au cas par cas a l'avantage d'être rapide. La stratégie développée ci-après est plutôt équationnelle.

Au lieu d'un choix aléatoire à chaque étape, on peut sélectionner la case la plus prometteuse. Ce qui donnera une stratégie du type **Best-First** dans le cadre de la méthode générale **Branch and Bound** *(développer et évaluer)*.

La clé du succès d'une telle stratégie est le choix de la **bonne** fonction d'évaluation d'un état.

VIII.2 Stratégie Best-First

Un exemple de l'espace d'états est donné ci-dessous (Dimension = 3, B doit jouer en permutant une case vide avec un de ses pions). A partir de l'état au sommet de ce graphe, B aura les 9 possibilités représentées ci-dessous :



Dans cette stratégie, *l'espace d'états est développé en largeur (par niveau) puis le meilleur état est choisi pour être joué puis être développé à son tour ....*

Comment choisir "le meilleur prochain coup" ?
On peut utiliser une fonction d'évaluation, par exemple la fonction *f(.)* suivante (B va jouer) :

f(nbr. de 2 Bs sans aucun N sur une ligne / colonne / diagonale)

qui est un compromis permettant d'obtenir un meilleur coup à jouer via un calcul simple. Voir ci-dessous pour d'autres fonctions. Noter que seule la machine utilisera cette fonction pour jouer.

Explications : comme on peut le remarquer, l'évaluation a lieu une fois que B aura joué. C'est à dire, : "combien le plateau vaudra pour B s'il joue telle case" ? Et B jouera l'état qui maximisera **f**.
Dans la figure ci-dessus, s'agissant d'une permutation (il y a déjà 3 Bs joués), ce calcule intègre la libération d'une case suivie de l'occupation d'une autre.
Il y a ici 9 cas possibles car si on enlève un des Bs, on pourra le placer à 3 autres endroits différents (il ne rejouera pas au même endroit !). Et comme on a 3 Bs, il y a 9 cas possibles pour chaque configuration (état).

Encore une explication !
Si la machine (que l'on essaie de rendre intelligente par cette stratégie) a déjà placé 3 pions, il faut libérer une case et en occuper une autre. Il y a donc deux étapes : libérer puis occuper ailleurs.
- On libère la pire des cases (la moins bonne) : en fait, on évalue la situation de l'humain et on libère la case qui le favorise le moins.
- On occupe ensuite la case qui nous favorise le plus.

➛ Les deux calculs peuvent utiliser la même fonction d'évaluation.

Une recommandation triviale : privilégier l'occupation (pour la machine) de la case centrale (pour Dimension 3/5/...) si vous avez le choix ! En particulier, dès le début, si votre fonction d'évaluation ne la choisit pas.

N.B. : D'autres stratégies (P. ex. le schéma *min-max* qui prévoit davantage de coups) pourraient être employées. Voir le support du cours sur cette stratégie qui permet d'établir un plan de jeu avant tout mouvement.

VIII.3 A propos de l'évaluation

Une exploration exhaustive du graphe d'états des jeux (en particulier pour les jeux complexes) est quasi impossible. Par exemple, en Echecs, il y a 35^{30} états possibles à examiner pour choisir le meilleur prochain coup à jouer.
Pour le Morpion, la complexité est de l'ordre de $little\_oh(|D|^{|V|})$ où $|V|$ est le nombre de variables (ici 6 pions) et $|D|$ est la taille du domaine des variables (9 possibilités au pire pour chaque pion) : on aura 9^6 possibilités à envisager (ceci majeure D^2! vue plus haut). C'est le cas lorsque l'échiquier est vide et on cherche à savoir où placer notre premier pion et essayer de prévoir d'avance tous les coups possibles ! Les fonctions d'évaluation jouent un rôle central.

VIII.4 Meilleure fonction d'évaluation

Pour Dimension=3, soit à évaluer la valeur de la case L/C pour la Couleur Color. On note
NL1 : nombre de pions de Couleur Color sur la ligne L, NL2 : ceux de l'adversaire ;
NC1 : nombre de pions de Couleur Color sur la colonne C, NC2 : ceux de l'adversaire ;
ND1 : nombre de pions de Couleur Color sur les deux diagonales passant par L/C,
ND2 pour l'adversaire.

La valeur de la case L/C est calculée par la fonction
$g = (NL_1 - NL_2)^3 + (NC_1 - NC_2)^3 + (ND1_1 - ND1_2)^3 + (ND2_1 - ND2_2)^3$.
➛ (si la case en main est sur une/deux diagonales). [1]

On choisira le prochain état (P. ex. un parmi 9 ci-dessus) avec une valeur maximale de **g**.
La fonction g (qui est une *distance*) affecte une valeur élevée aux lignes, colonnes et diagonales où l'on occupe déjà 2 cases (pour une Dimension=3) et où l'adversaire est absent.

☞ Noter que la fonction cubique ci-dessus est un compromis : elle n'est pas infaillible ! (voir ci-après).
➛ Si plusieurs cases ont la même valeur d'évaluation, privilégier (occuper) les diagonales.

VIII.5 Variante de la fonction d'évaluation

(I) On peut souhaiter utiliser une fonction quadratique. Dans ce cas, il faut tenir compte du signe des soustractions :
- $Facteur\_NL = -1\ si\ NL_2 > NL_1\ sinon\ 1$
- $Facteur\_NC = -1\ si\ NC_2 > NC_1\ sinon\ 1$
- Idem pour les diagonales.

puis $g = Facteur\_NL * (NL_1 - NL_2)^2 + Facteur\_NC * (NC_1 - NC_2)^2 + Facteur\_ND1 * (ND1_1 - ND1_2)^2 + Facteur\_ND2 * (ND2_1 - ND2_2)^2$.

(2) Egalement, vous pouvez encore améliorer la fonction **g** précédente en donnant un poids plus élevé aux pions bien placés de l'adversaire.

Modifier la fonction **g** de la manière suivante :

- $Facteur\_NL = 2\ si\ NL_2 > 1\ sinon\ 1$
- $Facteur\_NC = 2\ si\ NC_2 > 1\ sinon\ 1$
- $Facteur\_ND = 2\ si\ ND_2 > 1\ sinon\ 1$

1. Si on choisit une fonction quadratique, il faudra tenir compte du signe de $(NL_1 - NL_2)$ (idem pour les colonnes et diagonales). Voir l aversion améliorée.

Puis multiplier **g** précédente par le produit de ces facteurs :

$$g = g * Facteur\_NL * Facteur\_NC * Facteur\_ND$$

Cette modification privilégie les configurations où l'adversaire est bien placé.

IX Travail à rendre 3

Votre réalisation peut maintenant faire jouer Machine vs. Humain.

1- Si vous avez déjà réalisé votre version graphique, alors appliquez la stratégie Best-First.
➛ Rendez la machine intelligente en la munissant d'une fonction d'évaluation (quadratique) qui lui permettra de décider du "meilleur" coup à jouer à chaque étape.

2- Sinon, vous pouvez appliquer la stratégie Best-First en console puis réaliser votre version graphique.

X Heuristique Min-Max

Dans le cas précédent, la stratégie Best-First choisit (**localement**) le meilleur coup à jouer. Il est évident que cette stratégie (optimum local) ne garantie par un gain final.

La stratégie Min-Max décrite ici tente d'aller vers une meilleure optimisation :
◦ soit en développant l'arbre du jeu jusqu'à une profondeur limitée (p. ex. 2),
◦ soit en développant tout l'arbre du jeu et atteindre un optimum **global** qui prévoit tous les états possibles.
Dans les deux cas, on développera un premier niveau puis les états successeurs de ce niveau en envisageant ce que l'adversaire pourrait jouer et ce que nous pourrions ensuite jouer ... pour choisir le meilleur coup.

La stratégie Min-Max (ou *MiniMax*) s'inscrit également dans le cadre de la stratégie générale *Best-First*. Comme dans toute stratégie *Best-First*, on calcule un score pour chaque coup possible **du prochain niveau** pour ensuite conserver le meilleur. Le parcours de l'arbre des états est donc un parcours *breath-first* (en largeur d'abord).

Cette section sera illustrée par les jeu Morpion et *Nimes* (voir plus loin).

X.1 Développement partiel de l'arbre

On développe l'arbre du jeu jusqu'à une profondeur paramétrable. Dans la section X.4, page 21, on verra la version qui développe la totalité de l'arbre du jeu.

On va donc calculer l'arbre de tous les coups possibles à partir d'un état donné **jusqu'à** k **niveaux suivants** pour sélectionner le meilleur coup. On commencera par $k = 2$ mais vous pouvez essayer $k = 3$ ou plus.

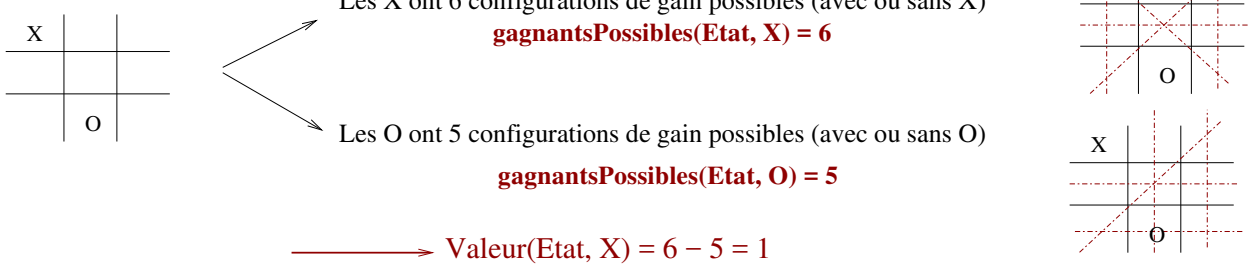
Nous aurons besoin d'une fonction **coût** (score) pour donner une valeur à chaque configuration (état du plateau). Cette fonction de coût doit être adaptée au développement partiel de l'arbre.

Nous pourrions parfaitement utiliser les fonctions coût étudiées plus haut (linéaire, quadratique, etc.). On décide cette fois d'utiliser la fonction d'évaluation suivante :

$Valeur(Color) =$ nombre d'états gagnants possibles pour $Color$ -
nombre d'états gagnants possibles pour l'adversaire.
$Valeur(Color) = +\infty$ si $Color$ gagne ;
$Valeur(Color) = -\infty$ si $adversaire(Color)$ gagne !

Évaluation d'un état

Nombre d'états gagnants possibles pour une couleur dans un État (plateau) donné :



Calcul de $gagnantsPossibles(Etat, Color)$ **:**
On sait que pour Dimension= 3, il y a maximum 8 états de gains possibles (3 en ligne, 3 en colonne et 2 en diagonale) sans tenir compte d'aucune couleur.
Pour calculer $gagnantsPossibles(Etat, Color)$, calculer la somme des 3 valeurs suivantes :
◦ le nombre de lignes où l'adversaire de $Color$ (c.à.d. O si $Color = X$) empêchera $Color$ de gagner
◦ le nombre de colonnes où l'adversaire empêchera $Color$ de gagner
◦ le nombre de diagonales où l'adversaire empêchera $Color$ de gagner

Puis $gagnantsPossibles(Etat, Color) = 8 - cette\ somme$.

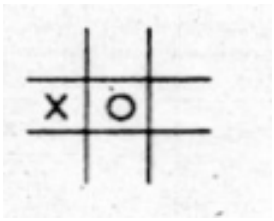
Par exemple, dans la figure ci-dessus, O empêche X de gagner en ligne 3 et en colonne 2 ; ce qui donne $8 - 2 = 6$ possibilités pour X.
A l'inverse, X empêchera O de gagner en ligne 1, en colonne 1 et la diagonale principale : $8 - 3 = 5$ possibilités pour O.

☞ Vous avez certainement remarqué que **le calcul de Valeur(État, Color)** peut être simplifié :

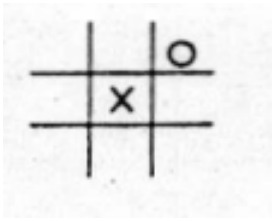
En fait, il suffit de calculer d'abord
- la somme des lignes, colonnes et diagonales bloquées par la couleur $Color$,
- faire la même somme pour $adversaire(Color)$;

puis de soustraire la 2e somme de la première.

D'autres exemples :



• Contre O, X bloque 2 chemins (1 ligne, 1 colonne) ; O bloque 4 chemins (1 ligne, 1 colonne, 2 diagonales) contre X ; ce qui donne $Valeur(Etat, X) = 2 - 4 = -2$
• Mais si on veut faire ce calcul au complet (cf. la figure ci-dessus) :
Les X ont 4 configurations de gain possibles (8-4) car $Somme(Etat, O) = 4$.
Les O ont 6 configurations de gain possibles (8-2) car $Somme(Etat, X) = 2$.
$Valeur(Etat, X) = 4 - 6 = -2$



Dans cet autre exemple, les X ont 5 configurations de gain possibles,
les O ont 4 configurations de gain possibles
$Valeur(Etat, X) = 5 - 4 = 1$
Calculé plus rapidement : X bloque 4 chemins contre O et O bloque 3 chemins
➛ $Valeur(Etat, X) = 4 - 3 = 1$

N.B. : une fonction d'évaluation similaire (et aussi triviale) dans le jeu de dames peut être la différence entre le nombre des blancs et le nombre des noirs. Pour le jeu d'échec, la fonction d'évaluation peut calculer la différence en nombre de points (associés aux pièces restantes) de chaque joueur, etc.

X.2 *Application de MinMax*

Munie de la fonction $Valeur(Etat, Color)$, si les X commencent, on développera l'arbre suivant (pour une profondeur=2) :

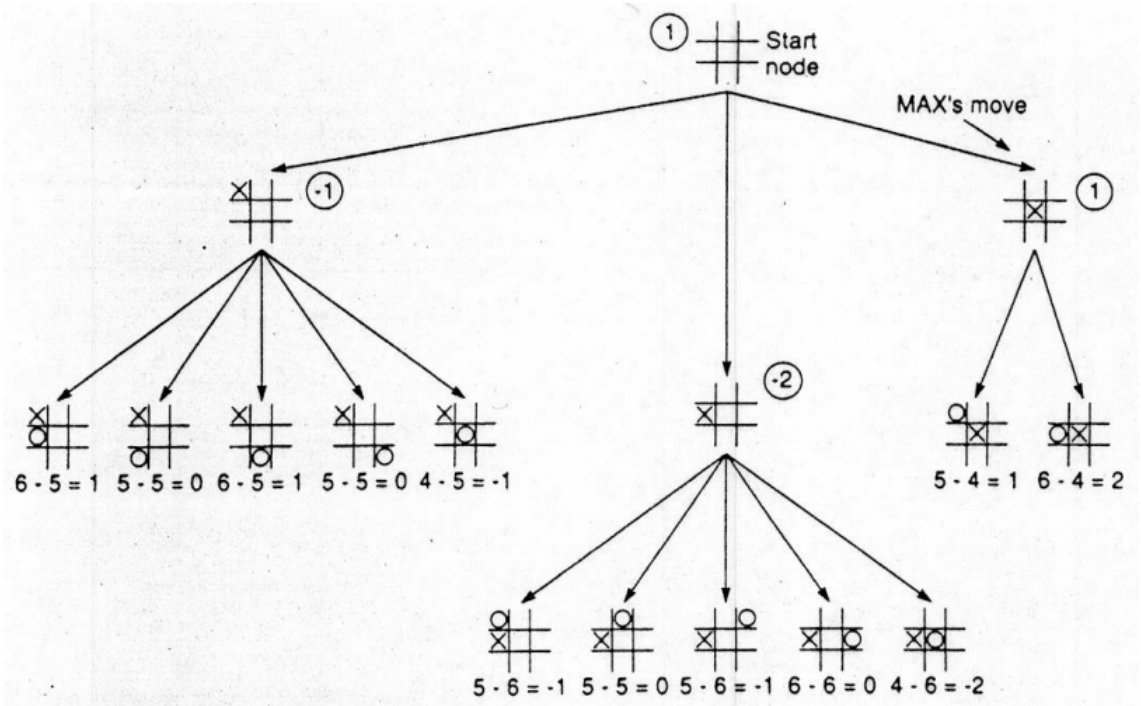


FIGURE 1 – Choix du 1e coup pour X (en $(1, 1)$)

Dans la figure 1, le développement des six autres cas n'apportent rien de plus. Empiriquement, on sait que le maximum de de la fonction *valeur(.,.)* est atteint quand X est au centre. Exploiter également la symétrie. Voir l'optimisation *Alpha-Beta* plus loin.

➛ X joue en $(2,2)$ et O en $(1,2)$.

Pour accélérer ce jeu, on propose également de ne développer qu'un nombre limité d'états successeurs. On peut par exemple décider d'un nombre aléatoires d'états à développer.
Ce nombre sera en général entre 1 et le nombre total d'état successeurs possibles.
Par exemple, pour le cas précédent, on choisira un nombre aléatoire entre 1 et 9.
Ceci affaiblit évidemment l'intelligence mobilisable dans le jeu. Un compromis entre ce nombre et la profondeur de l'arbre permet de fixer les paramètres de ce jeu.

X va maintenant jouer son 2e coup après avoir fait le meilleur choix pour son premier coup (max=1 et on place donc un X au centre en $(2,2)$). Il a été suivi par O qui s'est placé en $(1,2)$.

X joue son 2e coup

☞ La "racine" de l'arbre dans cette figure se trouve à droite où X n'a pas encore joué son 2e coup.

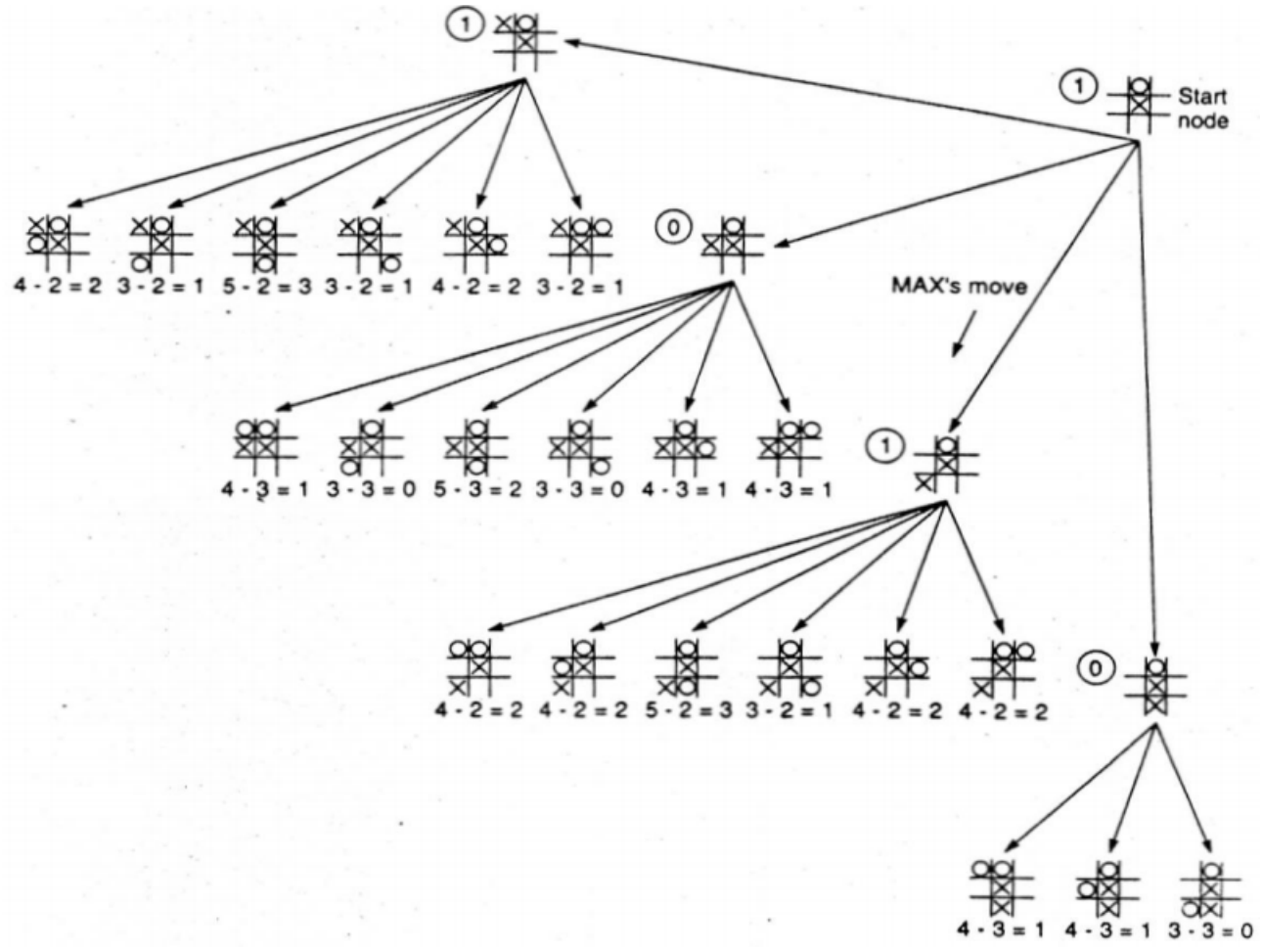


FIGURE 2 – Choix du 2e coup pour X (en $(3,1)$)

➛ X jouera donc en $(3,1)$ et O en $(1,3)$.
Le choix de X est donné par car *max(valeur(.,.)=1* (il y a au moins une alternative).

Ici également, le développement des autres branches ne modifiera pas le choix du meilleure case à jouer (pour X). Cette partie a été omise pour simplifier l'arbre d'états.
Vous pouvez également supposer qu'on ait décidé de ne développer que 4 successeurs!?

X joue son 3e coup

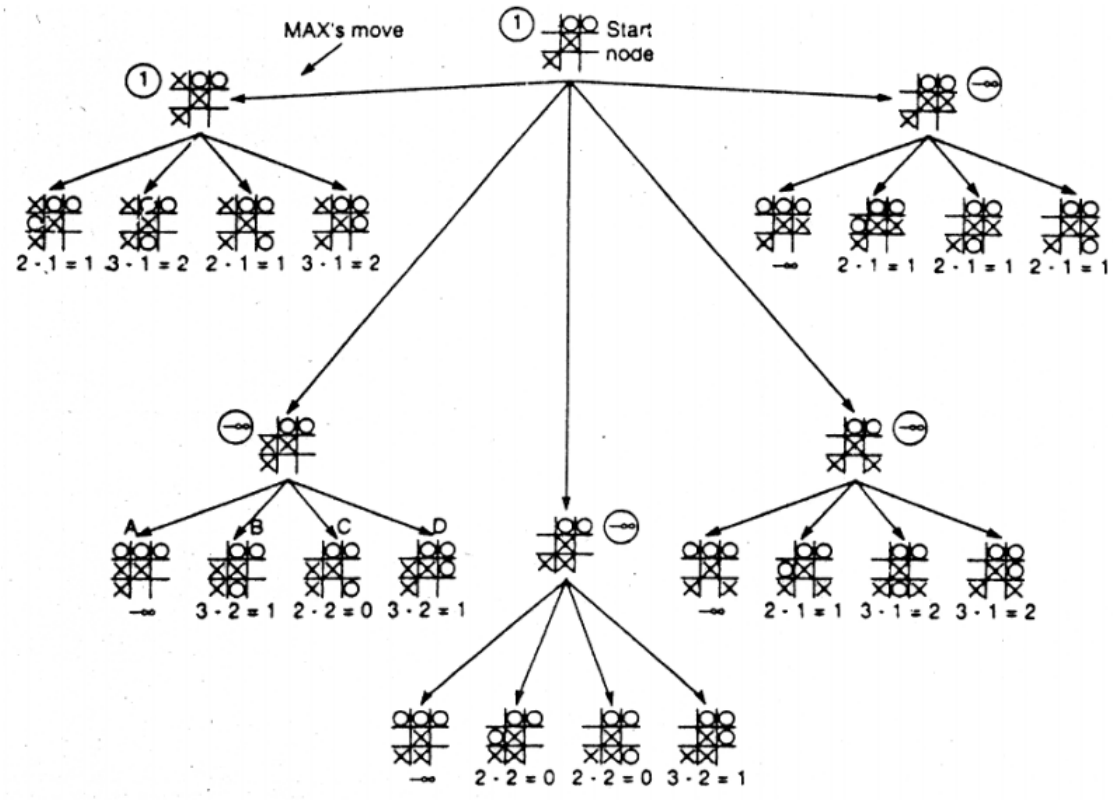


FIGURE 3 – Choix du 3e coup pour X (en $(1,1)$). Toutes les branches sont développées ici.

☞ Comme dans l'arbre de la figure 3, lorsqu'on étudie les coups possibles de l'adversaire (ici O) et que O gagne, on évalue l'état à $-\infty$ et puisque nous somme en état min (quand O joue) et en max quand X doit jouer.
☞ On constate dans ces cas qu'il n'est plus besoin de calculer les autres états de la même branche (voir plus loin l'optimisation *Alpha-Beta*)

➛ X jouera en $(1,1)$ et O jouera parmi $\{(2,1),(3,2),(3,3),(2,3)\}$.

Et ainsi de suite.

☞ On n'est pas certain de gagner puisqu'on ne développe que 2 niveaux (voire, un nombre limité de successeurs).

X.3 Algorithme de principe

Les pseudo-codes ci-dessous devraient vous aider à démarrer. Vous les adapterez à Python.

• Fonction de départ : **Jouer**
Initialisations et boucle principale du jeu jusqu'à un gagnant.

```
Fonction Jouer :
Début
    initialiser le Plateau à vides
    Soit 'B' la couleur de la machine
    Color  <- Décider qui commence ('B' ou 'N' ?)
    profondeur = 2     #(2 .. 6)

    Répéter jusqu'à (Fini = vrai)
       jouer_un_coup(G, Color, profondeur)
       # C.à.d. placer un pion ou déplacer  un pion (si 3 pions déjà placés)
       Fini <- gagnant(G,Color);
       Si (Fini = Faux)
       Alors Color<- l'autre_couleur(Color) ;
        # C-à-d. si (Color=B) alors Color <-- N sinon Color <-- B finsi
       Finsi ;
    Fin Répéter
    Color est le gagnant
Fin jouer
```

• Jouer un coup :
La machine (munie de l'I.A.) ou l'humain va jouer un coup.

```
# N'importe quelle couleur joue un coup, qq soit la configuration

Fonction jouer_un_coup(Plateau, Color_qui_doit_jouer, profondeur) :
    Si Color='N'  # Les humains
    Alors Humain_joue_un_coup(Plateau, Color_qui_doit_jouer)
    Sinon # Donc c'est la machine ('B')
        machine_joue_un_coup(Plateau, profondeur, Color_qui_doit_jouer)
    Finsi
Fin jouer_un_coup
```

• L'humain joue un coup : on lira au clavier le choix de la case

```
# L'humain joue (qq soit la configuration)

Fonction Humain_joue_un_coup(Plateau, Color_qui_doit_jouer) :
    Si 3 pions déjà  placés (Color_qui_doit_jouer)
        # Pour simplifier, Humain='N' mais laisser la souplesse
        # de choisir les couleurs (dans initialisation)
    Alors
        demander (x,y) d'une case à libérer (lecture au clavier + contrôle)
    Finsi
    demander (x,y) d'une case à occuper (lecture au clavier + contrôle)
    # DE préférence, éviter que l'humain rejoue la case qu'il vient de libérer
Fin Humain_joue_un_coup
```

• La machine joue un coup : elle profite de l'I.A. :

```
# La machine jouera à l'aide de l'IA (utilise min_max)
Fonction machine_joue_un_coup(Plateau, profondeur,  Color_qui_doit_jouer)
    # mode = 'max' (forcément)
    Si profondeur = 0 ou Profondeur > 6 :
    Erreur # il faut au moins une profondeur de 1 (et au plus 6)
    Finsi # On arrête le programme.

    plateau_to_play, sa_valeur =
        minmax(Plateau, profondeur , "max", Color_qui_doit_jouer)
    le Plateau du jeu =  plateau_to_play # MAJ du plateau global
Fin machine_joue_un_coup
```

• L'I.A. : Min-Max :

```
# La fonction minmax suivante joue un rôle centrale.
# Elle est appelée pour chaque coup à jouer par la machine et lui apporte
# l'intelligence artificielle adéquate.
# La fonction minmax conseille la meilleure case à la machine (par défaut 'B')
# selon la profondeur et la configuration
# Si mode='max', c'est la machine qui joue
# Si mode='min', c'est l'humain
# Corrélation avec   le param 'Color_qui_doit_jouer' :
# Si Color_qui_doit_jouer='B' (machine) alors mode="max" forcément
# MinMax est appellée initialement pour faire jouer la machine mais elle
# s'appelle récursivement à la fois pour faire jouer la machine ET
# l'humain (cf. arbre Minmax)

Fonction  minMax(Plateau, profondeur, mode,  Color_qui_doit_jouer)
#               renvoie (plateau, score)
    Si platau n'est pa svierge
    Alors
        Si profondeur =0 ou gagne(Color_qui_venait_de_jouer, plateau)
        Alors
            valeur = Score_d_un_plateau(plateau, Color_qui_venait_de_jouer)
            renvoyert (plateau, valeur)
        Finsi
    Finsi

    min_score = +infty
    max_score = -infty
    best_case_a_joeur = None
    liste_plateaux_successeurs=[]
    Si    Color_qui_doit_jouer possede  < 3 pions dans Plateau
    Alors remplir liste_plateaux_successeurs avec
          succ_Plateau_moins_de_3_pions_joués(plateau, Color_qui_doit_jouer)
          # Plateaux obtenus en plaçant un pion dans une case vide

    Sinon remplir liste_plateaux_successeurs avec
          succ_Plateau_deja_3_pions_joués(plateau, Color_qui_doit_jouer)
          # Plateaux obtenus en permutant une case vide et un pion déjà placé
    Finsi
```

• Suite Minmax ...

```
    Pour tout plateau_bis dans liste_plateaux_successeurs :
        # Appeler MinMax avec  l_autre(Color_qui_doit_jouer) car
        # Color_qui_doit_jouer a généreé ses plateaux successeurs
        plateau1, score = minMax(plateau_bis, profondeur-1, inverse(mode),
                l_autre(Color_qui_doit_jouer))
        Si mode = 'max' : # On sait que Color_qui_joue = 'B' (machine)
                    Si score > max_score :
                        best_score=score
                        best_plateau= plateau1
                        max_score = best_score;
                    Finsi

        Sinon : # mode= 'min': Color_qui_joue = Humain
                    Si score < min_score :
                        best_score=score
                        best_plateau = plateau1
                        min_score = best_score;
                    Finsi
        Finsi
    Fin Pour

    # ATTENTION : distinction importante ici :
    Si profondeur=profondeur_org #  Si c'est le 1er appel à Minmax
    Alors renvoyer best_plateau, best_score
    Sinon renvoyer Plateau, best_score # Celui recu en paramètre
    Finsi
Fin minMax
```

• Pour la fonction d'évaluation (la fonction `Score_d_un_plateau`), voir par exemple la section X.1 en page 13.

On peut également calculer la valeur d'un plateau par n'importe quelle fonction d'évaluation fournies dans ce BE (linéaire / quadratique / quadratique pondérée, etc.). Et en présence d'un plateau gagnant (par n'importe lequel des joueurs), affecter les bonnes valeurs.

• Les plateaux successeurs d'un plateau : cas $nbPions < 3$:

```
# générer tous les successeur du Plateau où on n'a pas encore joué 3 pions
# Les succ d'un plateau générés SANS IA sans aucune évaluation.
# C'est une simple génération de succ d'un plateau (le joueur n'intervient pas ici)

fonction succ_Plateau_moins_de_3_pions_joués(Plateau, Color_qui_doit_jouer) :
    liste = []
    # générer tous les plateaux possibles où "Color_qui_doit_jouer" en
    # placant un pion dans une des cases vides (bêtement !)
    pour toutes les cases vides C_vide disponibles
        Plat = placer un pion "Color_qui_doit_jouer" dans C_vide
        liste += Plat
    renvoyer liste
Fin succ_Plateau_moins_de_3_pions_joués
```

Suite ../..

• Les plateaux successeurs d'un plateau : cas $nbPions \geq 3$:

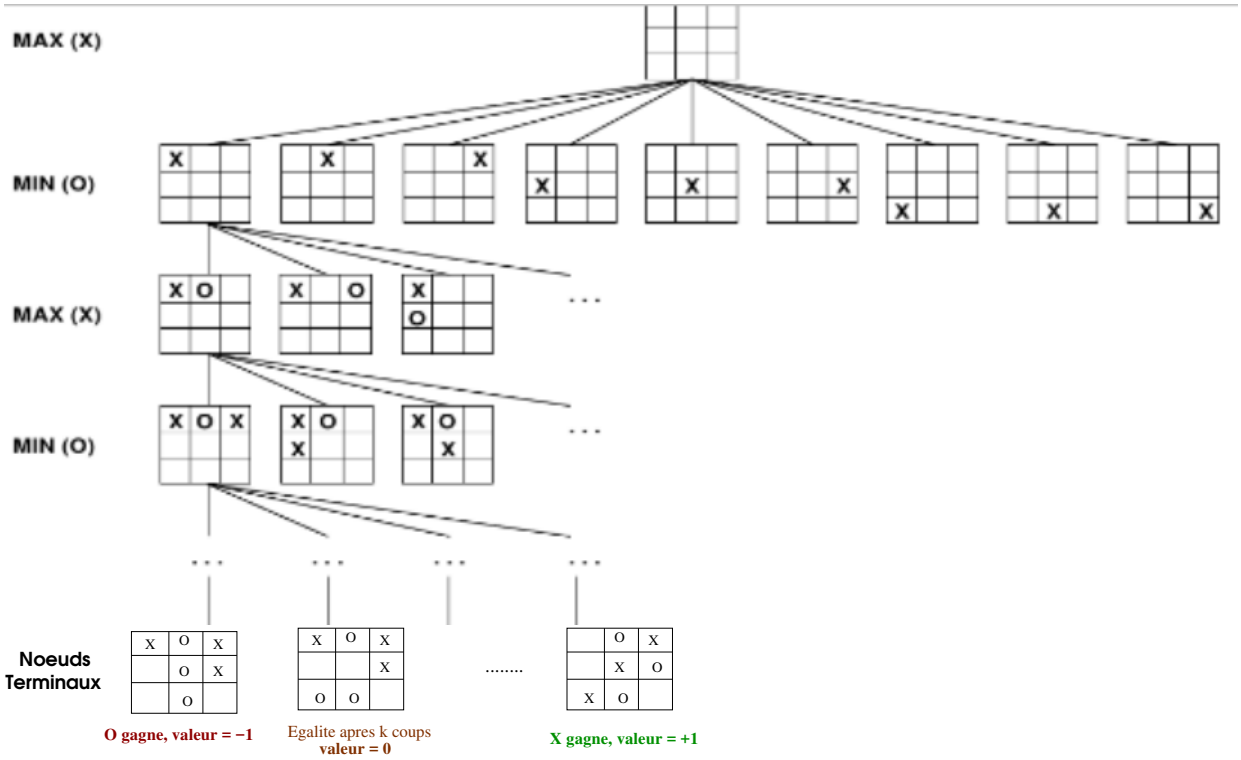
```
# Les succ d'un plateau générés SANS IA sans aucune évaluation
fonction succ_Plateau_deja_3_pions_joués(plateau, Color_qui_joue) :
    liste = []
    # Générer tous les plateaux possibles où on enlève un pion "Color_qui_joue"
    # et on le place dans une case vide (autre que celle qu'on vient de vider)

    pour toutes les case occupées C_Occup par Color_qui_joue
        Plat = enlever C_occup du "plateau"
        pour toutes les cases vides C_vide disponibles
                                            sauf C_occup qu'on vient de libérer
            Plat = placer un pion "Color_qui_joue" dans C_vide
            liste += Plat
    renvoyer liste
Fin succ_Plateau_deja_3_pions_joués
```

X.4 Développement total de l'arbre

Ici, on décide d'aller plus loin et développer tout l'arbre du jeu. Par bonheur, le graphe d'états développé pour ce jeu est fini et (raisonnablement) calculable (dit également traçable = *tractable*)

Ici aussi, avons-nous besoin d'une fonction **coût** (score) pour donner une valeur à chaque configuration (état du plateau).

Puisque l'arbre est totalement développé (les feuilles sont des état finaux), on peut par exemple et simplement donner la valeur +1 si un coup est gagnant et -1 s'il est perdant (et 0 en cas de match nul). D'autres valeurs (+100, -100 et 0) sont également possibles.



On suppose que l'humain (nous) n'avons pas besoin de conseil mais la machine, si ! On va continuer (travail-3 et 4 précédents) à donner une IA à la machine (dont nous sommes l'adversaire). Cette fonction sera appelée lorsque la machine doit jouer.

XI Réalisation avec interface graphique

Avec vos connaissances de *TkInter*, réaliser une interface graphique pour ce jeu en Python.

☞ Un fichier de création d'interface Tk est fourni.

XII Travail à rendre 4

Après avoir réalisé (d'abord) la version heuristique avec une fonction d'évaluation, introduire les éléments d'optimisation et d'heuristique Main-Max (pour la machine).

Délais : 4 semaines.

XIII Super-Bonus : Variante Alpha-Beta

Une optimisation de l'algorithme MinMax peut être décrite selon l'heuristique suivante : élaguer des branches qu'il est inutile d'explorer.
Si la valeur d'un noeud atteint un seuil , il est inutile de continuer à explorer les descendants de ce noeud
➛ La valeur d'un tel noeud n'interviendra pas / plus dans les calculs.

XIV Heuristique Monte Carlo Search Tree

L'heuristique de *la recherche arborescente de Monte Carlo* (Monte Carlo Search Tree = MCST) analyse les mouvements les plus prometteurs en développant **un arbre de recherche** par un échantillonnage aléatoire des coups à jouer dans **l'espace de recherche**.

L'heuristique MCST se base sur le développement d'un grand nombre de parties simulées (dit *playouts* ou *roll-outs*) qui vont jusqu'à un état final de gain/perte/nul. Les coups joués dans ces parties simulées sont sélectionnés au hasard.

Le résultat final du jeu de chaque partie simulée est ensuite utilisé pour pondérer les noeuds dans l'arborescence du jeu. L'idée est que les meilleurs coups (meilleurs noeuds) seront davantage susceptibles de mener à une victoire. Ils doivent donc d'être choisis dans les futures coups.

☞ Notons qu'après chaque vrai coup (coup légal) joué, le même nombre de simulations est effectué (et inscrits dans les noeuds simulés de l'arbre) puis on choisira le coup qui correspond au plus grand nombre de victoires.

L'efficacité de cette méthode augmente souvent avec le temps, car davantage de crédits sont attribués aux mouvements qui ont fréquemment abouti à la victoire du joueur actuel selon les tentatives précédentes.

Chaque cycle de recherche dans l'arborescente de Monte Carlo se compose de quatre étapes :

Sélection : Commencez à partir de la racine **R** de l'arbre et sélectionnez les noeuds enfants successifs jusqu'à ce qu'une feuille **F** soit atteinte.
☞ La racine est l'état actuel du jeu et une feuille est un noeud qui a un successeur potentiel à partir duquel aucune simulation n'a encore été lancée (donc la feuille n'a pas encore son propre sous-arbre). Voir ci-dessous sur la manière de biaiser le choix des noeuds enfants qui permet à l'arbre de jeu de s'étendre vers les mouvements les plus prometteurs.

Expansion : si **F** ne termine une partie par une victoire/perte/match-nul pour l'un ou l'autre des joueurs (voir ci-dessous pourquoi F n'est pas allé jusqu'au bout), on développe F en lui créant un (ou plusieurs) noeuds enfants puis on choisit le noeud **C** parmi eux. Ces noeuds enfants de F sont tous les mouvements valides à partir de la position de jeu définie par F.

Déploiement (ou Roll-out) : effectuez une partie aléatoire à partir du noeud C. La partie ainsi déployée consiste à choisir des mouvements aléatoires uniformes jusqu'à ce que le jeu soit terminé par un gain, une perte ou un jeu égal.

Rétropropagation : utilisez le résultat de la partie déploiement pour mettre à jour les informations dans les noeuds sur le chemin de R à C.

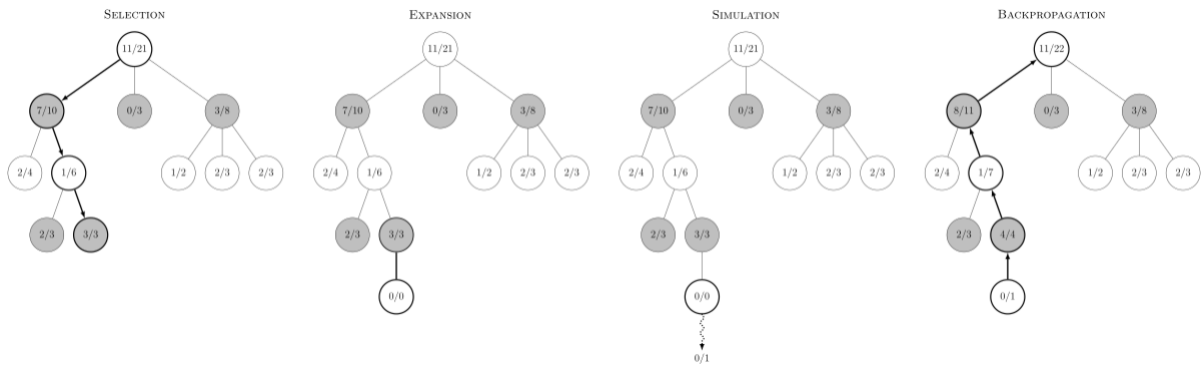


FIGURE 4 – Arbre de recherche MC (thanks to Wikipedia)

La figure ci-dessus montre les étapes d'une décision. Chaque noeud indique le rapport entre les gains et le nombre total de tentatives à partir de ce point de l'arbre de jeu pour le joueur que le noeud représente.

En étape de *sélection*, les noirs (la couleur grise des noeuds) vont jouer le prochain coup. Le noeud racine montre que jusqu'à présent, il y a 11 victoires sur 21 tentatives pour les blancs à partir de cette position (blancs car le noeud racine est blanc !).
Les noires totalisent 10/21 victoires noires affichées le long des trois noeuds noirs au dessous, dont chacun représente un mouvement noir possible.

Si les blancs perdent une *simulation*, on incrémente, pour tous les noeuds le long de la *sélection* , leur nombre de simulations (le dénominateur), mais parmi eux, seuls les noeuds noirs ont été crédités de victoires (le numérateur).
Par contre, Si le blanc gagne, tous les noeuds le long de la *sélection* augmenteront encore leur nombre de simulations, mais parmi eux, seuls les noeuds blancs seront crédités de victoires.

Dans les jeux où les matchs nuls sont possibles, un match nul entraîne une incrémentation du numérateur pour les noirs et les blancs de 0,5 et le dénominateur de 1. Cela garantit que lors de la sélection, les choix de chaque joueur s'étendent vers les coups les plus prometteurs pour ce joueur, ce qui reflète le objectif de chaque joueur de maximiser la valeur de son coup.

Les itérations de recherche sont répétées soit jusqu'à aboutir à un état gain/perte/nul (si pas limite de temps ou de profondeur pour jouer un coupe, comme c'est le cas en Echecs), soit tant qu'il reste du temps imparti à un déplacement ou bien la profondeur max n'a pas été atteinte.

Ensuite, le coup avec le plus de simulations effectuées (c'est-à-dire le plus grand dénominateur) est choisi comme le choix final.

XIV.1 Méthode Pure 'Monte Carlo game search'

La méthode ci-dessus peut être appliquée à n'importe quel jeu avec un nombre fini de coups en particulier dans les jeux de remplissage de plateau (Nimes, Morpion, Go, Echecs, Shogi, Puissance 4 ...).

Pour chaque position, tous les coups possibles sont déterminés : *k* parties aléatoires simulées sont joués jusqu'à la toute fin et les scores sont enregistrés puis le coup menant au meilleur score est choisi. Les égalités (matchs nuls) peuvent être brisées par une pile ou face.

La Méthode *Pure Monte Carlo game search* donne lieu aux résultats intéressants dans les jeux avec la possibilité de coups aléatoires. Les simulations permettent de converger vers un jeu optimal lorsque *k* tend vers l'infini.

☞ Notons que AlphaZero de DeepMind utilisent cette méthode mais remplacent l'étape de simulation par une évaluation basée sur un réseau de neurones.

XIV.2 Aller plus loin : Équilibre Exploration et Exploitation

Dans un jeu, lorsque chaque joueur a un temps fini pour jouer un coup (de développer son arbre de recherche, cf. Échecs, Go, ...), la principale difficulté dans la sélection des noeuds enfants est de maintenir un certain équilibre entre l'exploitation de grande profondeurs (avec un taux de gain moyen élevé) après un mouvement et l'exploration de mouvements avec peu de simulations.

Une façon d'équilibrer l'exploitation et l'exploration est la stratégie UCT (Upper Confidence Bound 1 applied to trees).

UCT (déjà appliquée dans les processus de décision muti-étages tel que le processus de décision de Markov) recommande de choisir dans chaque noeud de l'arbre du jeu le coup pour lequel l'expression suivante a la valeur la plus élevée :

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$$

Dans cette formule :
◦ w_i représente le nombre de victoires pour le noeud considéré après le ième coup
◦ n_i représente le nombre de simulations pour le noeud considéré après le i-ème déplacement
◦ N_i représente le nombre total de simulations après le i-ème mouvement exécuté par le noeud parent de celui considéré
◦ c est le paramètre d'exploration : théoriquement égal à $\sqrt{2}$ ou choisi empiriquement (comme c'est généralement pratiqué).

La première composante de la formule ci-dessus correspond à l'exploitation ; elle est élevée pour les mouvements avec un taux de victoire moyen élevé. La seconde composante correspond à l'exploration ; elle est élevée pour les coups avec peu de simulations.

La plupart des implémentations "poussées" de MCTS (*la recherche arborescente de Monte Carlo*) est basée sur une variante de l'UCT qui remonte à l'algorithme d'optimisation de simulation AMS (Adaptive Multi-stage Sampling) pour estimer la fonction de valeur dans les processus de décision de Markov à horizon fini (MDP) en Recherche Opérationnelle (optimisation).
AMS a exploré l'idée de l'exploration et de l'exploitation basées sur UCB1 (Upper Condidence Bound 1) dans la construction d'arbres échantillonnés / simulés (Monte Carlo) et a été le principale point de départ pour UCT. Voir ci-dessous pour un exemple sur Morpion.

Idées d'amélioration

Diverses modifications de la méthode MCTS (recherche arborescente Monte Carlo) de base cherchent à réduire le temps de recherche en utilisant par exemple des connaissances spécialisées spécifiques à un domaine.

La recherche arborescente de Monte Carlo peut utiliser des simulations plus ou moins simples ou élaborées. Les cas simple (vu plus haut) consistent en des mouvements aléatoires tandis que les plus élaborés appliquent diverses heuristiques pour influencer le choix des mouvements.

☞ Notons que paradoxalement, jouer de manière sous-optimale (aléatoire uniforme) dans les simulations rend parfois MCST plus fort que des simulations plus élaborées !

Néanmoins, les heuristiques élaborées peuvent améliorer le traitement des cas difficiles en utilisant par exemple les résultats de simulations précédentes (par exemple, l'heuristique de la dernière bonne réponse) ou d'utiliser les connaissances d'un expert sur un jeu (compliqué) donné.
Par exemple, dans de nombreux programmes de jeu de Go, certains motifs dans une partie du plateau influencent la probabilité de se déplacer dans cette zone. De même, dans le jeu Morpion, la case centrale représente une priorité.

(I) Des connaissances spécifiques du domaine peuvent être utilisées lors de la construction de l'arbre de jeu (élaboré) pour aider à l'exploitation de certaines variantes.

Une de ces méthodes attribue des valeurs a priori non nulles au nombre de simulations gagnées et jouées lors de la création de chaque noeud enfant. Ce qui entraîne des augmentations ou des réductions artificielles des taux de victoire moyens qui font que le noeud est choisi plus ou moins fréquemment lors de l'étape de sélection.

(II) Une méthode apparentée, appelée **biais progressif**, consiste à ajouter à la formule UCB un élément $\frac{b_i}{n_i}$ où b_i est un score heuristique du ième mouvement.

(III) **Une amélioration pratique (et raisonnable) :**

On constate que la recherche arborescente de Monte Carlo (méthode de base) recueille suffisamment d'informations pour trouver les mouvements les plus prometteurs seulement après plusieurs tours avec des simulations essentiellement aléatoires.

Cette phase exploratoire peut être significativement réduite dans une certaine classe de jeux en utilisant la stratégie dite **RAVE** (Rapid Action Value Estimation).

Dans ces jeux, les permutations d'une séquence de coups mènent à la même position. En règle générale, ce sont des jeux de société dans lesquels un mouvement implique le placement d'une pièce ou d'une pierre sur un plateau.
Dans de tels jeux (comme Go, Morpion, Puissance 4, Shogi...), la valeur de chaque coup n'est souvent que légèrement influencée par les autres coups.

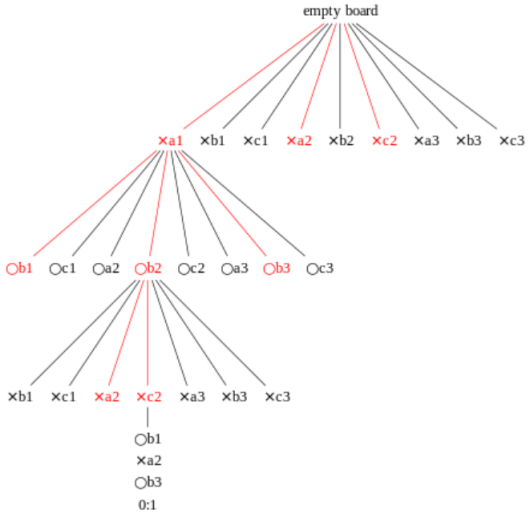
Dans la stratégie RAVE, pour un noeud d'arbre de jeu donné N, ses noeuds enfants C_i stockent non seulement les statistiques des gains dans les simulations commencées au noeud N mais aussi les statistiques des gains dans tous les simulations commencées au noeud N et en dessous, s'ils contiennent le coup i (également lorsque le coup a été joué dans l'arbre, entre le noeud N et la suite de simulation). De cette façon, pendant les simulations, le contenu des noeuds de l'arbre est influencé non seulement par les coups joués immédiatement dans une position donnée, mais aussi par les mêmes coups joués plus tard.

Exemple de la méthode RAVE en Morpion :

Dans la figure ci-contre du jeu Morpion *(Thanks to Wikipedia)*, les 'X' au début de chaque motif représentent les 'croix' et les 'O' les 'Ronds'.

L'échiquier est une matrice avec les lignes a,b,c et les colonnes 1,2,3.

➛ Donc, "Xa1" = les "croix" jouent la cases "a1" et "Ob3" = les 'ronds' jouent la case "b3".



Dans les noeuds rouges, les statistiques seront modifiées après la simulation *b1-a2-b3* : "Ob1, Xa2, Ob3".

Lors de l'utilisation de RAVE, l'étape de sélection choisit le noeud pour lequel la formule UCB1 modifiée

$$(1-\beta(n_i,\tilde{n}_i))\frac{w_i}{n_i}+\beta(n_i,\tilde{n}_i)\frac{\tilde{w}_i}{\tilde{n}_i}+c\sqrt{\frac{\ln t}{n_i}}$$

a la valeur la plus élevée.

Dans cette formule, $\tilde{w}_i$ et $\tilde{n}_i$ représentent le nombre de simulations gagnés contenant le coup i et le nombre de tous les simulations contenant le coup i.

La fonction $\beta(n_i,\tilde{n}_i)$ doit renvoyer une valeur proche de 1 pour un petit n_i et proche de zéro pour une grande valeur de $\tilde{n}_i$.

Une des nombreuses formules pour $\beta(n_i,\tilde{n}_i)$ stipule que dans des positions équilibrées, on peut prendre $\beta(n_i,\tilde{n}_i)=\dfrac{\tilde{n}_i}{n_i+\tilde{n}_i+4b^2n_i\tilde{n}_i}$, où b est une constante choisie empiriquement.

☞ Notons enfin que MCST (en particulier la version de base ou toute versions vues ci-dessus où les simulations sont indépendantes) peut être facilement parallélisée et gagner en temps total d'exécution.

XV Barème

- Travail 1 : Jeu de base (jeu aléatoire, Machine / humain vs. Machine) version console : 7 points
- Travail 2 (Bonus) : Ajout de l'interface graphique (tkinter) : +6 points
- Travail 3 : Heuristique Best-First pour Machine vs. Humain en version console : +3 points
- Travail 4 : Comme Travail-3 mais avec l'heuristique Min-Max : +10 points
- Travail 5 (Bonus) : Comme Travail-3 mais avec l'heuristique alpha-Beta : Bonus de 4 points
- Travail 6 (Bonus) : Comme Travail-3 mais avec l'heuristique MCST (dans sa version de base) : Bonus de 7 points.
  Dans une version plus élaborée, les points Bonus s'élèvent à 10 (voir MCTS améliorée)

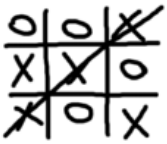
☞ L'ajout de l'interface graphique & le Travail 5 & 6 sont donc des bonus.

☞ Il est possible, à partir du Travail-4, de changer de jeu et de choisir (p.ex.) **Nimes**.

XVI Variantes du jeu Morpion

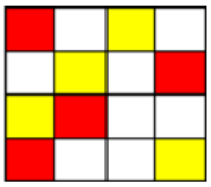
XVI.1 Nombre de pions

1. Grille de dimension D (par exemple 3) mais chaque joueur dispose de *D +1* (donc 4) pions.



2. Pour éviter toute permutation, pour une dimension D, on peut disposer de $D^2/2 + D/2$ pions pour chacun. On prend ici la partie entière dans les divisions.
3. D'autres variantes (par exemple avec *Dimension -1* pions pour chacun) existent.

Exemple avec Dimension =5, avec 4 pions pour chacun où toute rangée de 3 (ici les jaunes) est gagnante.



XVI.2 Jouer sur une bande

Pour Dimension =3, on a un tableau de 9 cases numérotés de 1 à 9.

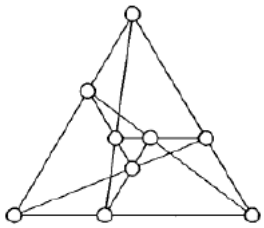
1	2	3	4	5	6	7	8	9
N_3	N_2	3	B_3	5	B_2	N_1	B_1	9

Le but est de placer ses pions (nombre de pions non limité) et de couvrir 3 cases dont la somme des chiffres = 15.

Par exemple (voir la 2e ligne du tableau), N joue 7, B joue 8, N joue 2 et B joue 6. Puis N joue 1 et B joue 4. Maintenant, N peut jouer 5 pour bloquer les Bs (éviter aux B d'avoir 4+5+6=15) mais alors, B jouera 3 et gagnera (3+4+8=15).

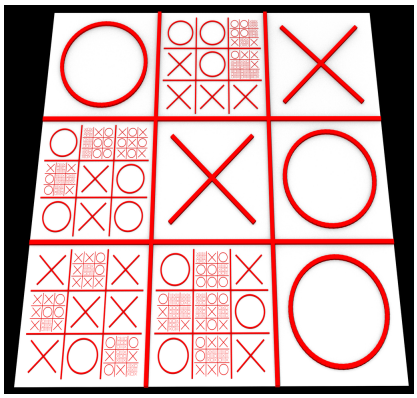
• Une autre variante plus simple de ce cas est de couvrir simplement 3 cases contiguës.

XVI.3 Un plateau curieux



Pour avoir une chance égale de gagner que l'on soit premier ou second à jouer, le plateau (Dimension =3) doit être de la forme ci-dessus [M. Gardner, Math Circus].

XVI.4 Un autre plateau curieux

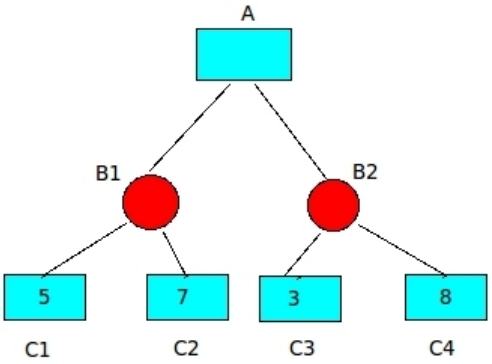


• Dans cette version, on choisit une des 9 cases du grand plateau (par exemple celle du centre) qui contient elle même un petit plateau de 9 cases ; supposons que B y joue son premier coup.
Suivant la case jouée (1/1 ... 3/3) dans ce petit plateau, disons par exemple 2/1, l'adversaire doit se reporter à la case 2/1 du grand plateau et jouer dans le petit plateau qui s'y trouve.
Et ainsi de suite jusqu'à ce que l'un des joueurs gagne au moins un des (petits) plateaux.
On peut aller plus loin et jouer les 9 petits plateaux. A la fin des 9 parties, compter qui a gagné le plus.

☞ Hyper-Bonus : réaliser cette version si vous en avez le courage !

XVII Supplément : retour sur le principe de Min-Max

Supposons que dans un (autre) jeu très simple, que l'on soit dans l'état A et qu'il n'y ait que deux états successeurs possibles $(B1, B2)$ pour A.
On se demande comment choisir entre les coups $B1$ et $B2$. La stratégie Best-First local évaluera les deux et choisira le meilleur.
Dans Min-Max, on va envisager les coups possibles que l'adversaire pourrait jouer : $C1, C2$ si on jouait $B1$ et $C3, C4$ si on décidait de jouer $B2$.
Le niveau A est dit le niveau **Max** (on choisira le maximum des scores des fils de A), le niveau B $(B1, B2)$ est dit du niveau **Min** et le niveau suivant (C_i) est labellisé **Max**, et ainsi de suite alternativement.



Celui qui utilise cette stratégie maximisera toujours ses chances ($max(B1, B2)$ donnera le prochain coup à jouer pour l'état A) et l'algorithme Min-Max suppose que l'adversaire jouera toujours le meilleur coup possible.

☞ On justifie la label **Min** des B_j car on se dit que l'adversaire jouera un coup C_i qui nous désavantagera (l'avantagera, lui) et donc on prendra le min des C_i sous un B_j donné.
On va choisir le **maximum** des valeurs des noeuds fils de A, sachant que la valeur de chaque noeud fils de A est lui-même le **minimum** de ses noeuds fils, et que chaque noeud fils des noeuds fils de A est lui-même le maximum des ses propres noeuds fils, etc... jusqu'à arriver aux feuilles qui symbolisent la fin de la partie.

Admettons que l'état $C1$ a un score de 5 et $C2$ un score de 7. Le noeud $B1$ récupère 5 (c.à.d. $min(C1, C2)$) et $B2$ aura 3 (c.à.d. $min(C3, C4)$). Le noeud A récupère à son tour le $max(B1, B2) = 5$. Ayant choisi la valeur finale 5 de $B1$, on jouera $B1$ et ce sera le noeud $C1$ qui devra être développé pour la suite (voir aussi $Alpha - Beta$)

On devine que certains jeux auront un arbre très complexe. On peut se limiter à une profondeur sachant que pour le jeu Morpion, la profondeur maximum pour un plateau 3×3 est 6. Cette valeur peut varier pour rendre la machine plus ou moins (profondément) intelligente.

XVIII Supplément : Min-Max sur Nimes

On considère les jeux à deux joueurs, avec :
- l'information complète (chaque joueur a la connaissance complète du jeu entier),
- sans hasard (pas de lancement de dés ou cartes par exemple),
- avec la somme nulle (la somme des gains des deux joueurs est zéro).

➛ Le jeu d'échec est un exemple typique.

Le joueur qui commence est dénoté par A et l'autre par B.
Considérons seulement les gains simples, c-à-d : 1 si A gagne (alors B perd), -1 si A perd (et victoire de B), 0 si égalité.

Pour ce type de jeu, il y a toujours une **stratégie de gain**, qui est une manière de jouer pour un des joueurs qui, indépendamment des mouvements de l'adversaire, s'assure qu'il ou elle ne perd pas. Ceci signifie que A a un $gain \geq 0$ (si A a une stratégie gagnante), B de gain $leq0$ (si B a une stratégie de gain mais qui perd devant A).
On peut montrer qu'il y a toujours une stratégie gagnante, mais il n'est pas possible de dire a priori lequel des joueurs la possède (dépend du jeu, et les deux cas se produisent).

Arborescence associée :
Comment une stratégie de gain (gagnante) peut être trouvée algorithmiquement ?
La démarche (algorithmique) de la mise en place constituera une preuve constructive de l'existence de cette stratégie.

Pour ce faire, on utilise une arborescence afin d'appliquer une recherche :
- La racine de cette arborescence est l'état initial du jeu.
- Les noeuds descendants de la racine sont tous états de jeux obtenus après le premier mouvement du joueur A.
- Les noeuds du niveau suivant s'obtiennent après un premier mouvement de B, et ainsi de suite, alternant les coups de A et de B.
- Les feuilles terminales (tout en bas de l'arbre) sont des états obtenu par une succession alternée (A et B) de mouvements qui représentent alors un match, après quoi il n'y a plus de mouvement (le jeu étant terminé).
- Pour chaque feuille, le gain est évalué comme ci-dessus :

➛ 1 si A gagne, -1 si B gagne et 0 pour une un match nul.

N.B. : Un état donné du jeu peut apparaître plusieurs fois dans l'arborescence.
C'est inhérent à ce modèle du jeu, puisqu'une situation donnée dans un jeu peut généralement être obtenue par des séquences différentes de mouvements.

XVIII.1 Exemple de Nimes

- L'arborescence d'un jeu peut vite devenir énorme.
- Pour présenter un cas accessible, considérons le jeu de Nimes.
- Sous sa forme classique (popularisée par un film français dans les années 60), il y a quatre piles d'allumettes contenant respectivement 1, 3, 5, et 7 allumettes.
- Alternativement, chaque joueur enlève autant d'allumettes (mais au moins un) de seulement une des piles.

Le joueur qui enlève la dernière allumette aura perdu.

Illustration

Limitons-nous à une version réduite de ce jeu avec seulement deux piles d'un et trois allumettes.
- ◦ La figure 5 : l'arborescence entière
- ◦ Chaque état : le couple **(allumettes_Pile1 - allumettes_Pile2)**

➛ Ex : (1-3) pour l'état initial (une allumette sur la 1e pile et 3 sur la seconde).

☞ N.B. : jeu simple mais son arborescence Min-Max est déjà un peu compliqué.

Rappel : un joueur peut enlever 1,2 ou 3 allumettes d'une seule pile. Le joueur qui joue doit enlever au moins une allumette. Le **perdant** est celui qui enlève la dernière allumette de la table (ensemble des piles).

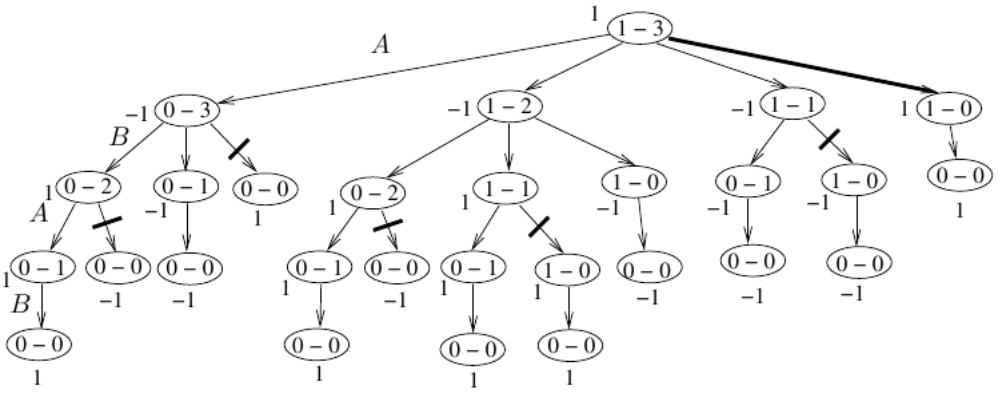


FIGURE 5 – Arbre de Nimes

XVIII.2 Explication de Min-Max sur Nimes

Trouver une stratégie de gain est plus difficile que l'optimisation simple d'une séquence de décisions (cf. Ci-dessus).
En effet, il y a une poursuite antagonique entre les joueurs : celui qui commence (A) recherche un gain maximum, puisque c'est directement son gain.
L'autre joueur (B) essaye de réduire au minimum ce gain final puisque son gain est l'opposé de celui de A.

Le principe essentiel de l'algorithme est celui de remonter les gains depuis les feuilles (où les valeurs sont connues) vers la racine, où la valeur retournée indiquera lesquels des joueurs bénéficie d'une stratégie gagnante.

Généralement le gain retourné pour un sommet de l'arborescence égalera 1 si c'est une position de gain pour le joueur A, -1 si c'est une position perdante pour A (et donc gagnante pour B), 0 si ce n'est pas une position de gain pour aucun des joueurs.
Cette dernière situation est celle d'un jeu nul, et se produit si aucun des joueur ne fait une "erreur" : c-à-d, quand chaque joueur évite, si possible, d'entreprendre une démarche menant à une situation de gain pour son adversaire.
Dans ce cas particulier, nous pouvons dire que chaque joueur a une stratégie de gain (une stratégie que l'on appellerait " non-perdante ").
Le principe décrit indique par lui-même comment calculer les gains.

Exemple
Soit le cas d'un état du jeu où c'est le tour de A, et supposons que les gains de tous les noeuds dans l'arborescence ont déjà été déterminés.

La règle à appliquer est que le gain retourné à un noeud est le maximum des gains de ses noeuds descendants. Cela détermine également le "meilleur mouvement " pour A.

Une formule semblable s'applique dans le cas où c'est le tour du B, avec le minimum au lieu du maximum. Ainsi, du fond jusqu'au sommet (des feuilles à la racine), il est possible de déterminer les valeurs de gain retournées pour chaque sommet de l'arborescence et, in fine, pour la racine. Cette technique de renvoyer alternativement un minimum et un maximum explique le nom min-max donné à cet algorithme.

Table des matières