BERTIN Luc – GROUPE B

RAPPORT DE DEVOIR

Structure de Données

Problème : TIC TAC TOE

Mon travail traitre entièrement des 3 résultats à obtenir.
L'objectif de problème est de développer une intelligence artificielle (IA), qui permettrait à l'ordinateur de jouer astucieusement contre le joueur à un jeu de Morpion ou Tic-Tac-Toe.

I/ Structures de données choisies :

a) La grille du jeu de Morpion

La grille du jeu de Morpion est une simple matrice d'entiers, désigné par un pointeur int**. La matrice est de taille 3x3 pour obtenir les 9 cases à jouer. Comme j'ai commencé le travail à Noël j'ai appris à la première séance de la rentrée qu'un tableau de 9 cases contiguës aurait été plus efficace mais ai gardé le choix de poursuivre mon implémentation initiale. Dans l'interface utilisateur on propose de remplir une case en désignant un numéro de ligne et de colonne, qui iront de 1 à 3 inclus (et non pas de 0, l'utilisateur n'étant pas censé savoir que la première ligne par exemple serait la ligne 0). ***Les entiers mis dans la matrice en lieu et place des symboles conventionnels (la « croix » et le « rond ») sont 1 et 2, 1 représentant le symbole du joueur, 2 celui de l'IA***. On veille à concevoir une fonction pour initialiser la matrice à 0 !

b) L'arbre n-aire des possibilités

Pour concevoir la structure de l'arbre, je me suis aidé de la modélisation d'un arbre n-aire dessinée dans le document PDF donné en annexe du devoir à effectuer.
Pour chaque nœud de l'arbre on a donc un pointeur vers un fils, un père et un pointeur vers un frère. Chaque nœud de l'arbre contiendra aussi l'état actuel du jeu de Morpion représenté par la matrice associée int** ainsi qu'un entier qui désigne le score (le score de chaque état vis-à-vis du joueur donné a son importance dans l'exploitation de MinMax car il permettra de trouver « le chemin » menant a la victoire ⇔ les coups à jouer pour gagner).

Cette structure est à mon sens la plus simple et intuitive, et la plus flexible. En effet si on avait conçu une structure constituée que de fils, on n'aurait pas pu prévoir le nombre de fils pour un nœud donné. Pour contourner ce problème on aurait pu faire un tableau de pointeur de fils, mais alors la flexibilité aurait été amoindrie vis à vis d'un pointeur frère pour chaque nœud (notre méthode) qui s'apparente alors un peu à une liste chaînée.

II/ Fonctions et procédures menant aux sous-résultats imposé dans l'énoncé

Résultat 0 :

« Menu permettant de démarrer le jeu, de savoir qui joue en premier, et à chaque tour demandant où souhaite jouer le joueur et l'informant où l'IA a joué, choix aléatoire pour l'IA, et affichage de l'état du jeu à chaque tour. »

-menu permet à partir d'un choix que rentre l'utilisateur (1 ou 2) de laisser le joueur ou l'IA jouer en premier. Attention : l'utilisation de scanf(« %d ») impose qu'un entier doit être rentré, sous peine de bug inhérent à la fonction dans le cas d'une saisie autre qu'un entier.

-fonction_periodique permettra de changer de joueur en faisant dans les appels récursifs un fonction_periodique(num_joueur), ainsi si le numéro de joueur est 1 il devient 2 et inversement jusqu'à la fin du jeu.

-viennent ensuite les fonctions de création, d'initialisation (très important !), d'affichage des matrices (grille du jeu).

-3 fonctions supplémentaires vérifient s'il y a ou non un coup_gagnant, c'est à dire si trois entiers sont alignés en lignes, colonnes, ou diagonales pour une matrice3X3 (on aurait pu l'adapter sur des matrices de taille quelconque mais il s'agit d'un jeu de Morpion simple). On veille bien à ce que les 0 (=case vide) ne soit pas comptés comme des entiers joués par les joueurs. Les fonctions retournent 1 s'il y a bien une réussite, 0 si aucun coup gagnant.

-est_remplie_matrice vérifie s'il n'y a aucune place vide (contenant un 0 donc) et retourne 1 si la matrice est complètement remplie.

-est_gagnee_partie retourne 1 si au moins l'une des réussites (coup_gagnant suivant les 3 directions) est effectuée et donc si l'un des joueurs a gagné en alignant 3 entiers, qqe soit la direction.

-est_occupee_place retourne 1 si la case est déjà occupée par un entier d'un joueur.

-remplissage_matrice est une sous-fonction de remplissage. Pour une ligne et une colonne donnée entre 1 et 3 (pas 0, l'utilisateur n'étant pas censé savoir que la colonne 0 existe), ainsi qu'un entier joueur elle remplie la case matrice[num_ligne-1][num_colonne-1] de l'entier.

-remplissage_joueur renvoie la matrice après un remplissage opéré par le joueur, la fonction se sert d'un entier servant comme booléen pour demander à l'utilisateur de remplir une case tant que la ligne et la colonne soit correctement renseignée ET que la place ne soit pas occupée. Elle se sert de remplissage_matrice pour remplir cette-dernière.

-remplissage_IA renvoie la matrice après remplissage aléatoire opéré par l'IA grâce à la fonction rand avec srand(time(NULL)) dans le main pour généré un nombre aléatoire compris ici entre 1 et 3. Même consigne que pour le joueur, tant que la case est déjà occupée d'autres nombres aléatoires seront renseignés. On n'oublie pas un affichage des colonnes et lignes jouées pour renseigner l'adversaire, puis affichage de la matrice.

-jeu est la fonction principale de ce sous-résultat. Elle est récursive. Je me suis astucieusement accordé à attitrer 1 comme l'entier du joueur et 2 celui de l'IA, ceci coïncidant avec le choix de laisser ou non jouer le joueur dans la fonction menu. Ainsi si choix == 1, alors l'homme est le premier joueur à jouer, c'est aussi le numéro désignant l'homme-joueur, et on remplit la matrice avec remplissage_joueur. On affiche la matrice puis on appelle récursivement la fonction pour faire jouer cette fois le joueur 2 (l'IA donc) grâce à return jeu(matrice, fonction_periodique(choix_premier_joueur)). Si la matrice est remplie mais qu'aucun coup-gagnant n'a été détecté on renvoie 3 (un nombre quelconque différent de 1 ou 2). Sinon on renvoie le dernier joueur ayant joué le coup_gagnant (avec fonction_periodique(choix_premier_joueur).

-enonciation_gagnant affiche un message de victoire pour le joueur si le numéro de joueur en paramètre == 1, de victoire pour l'IA si gagnant ==2, de match nul dans le cas autre (gagnant == 3) si la matrice est remplie sans aucun coup-gagnant donc.

Résultat 1 :

« La génération de l'arbre de toutes les possibilités pour un joueur donné à partir d'un état donné du jeu (donc non nécessairement la grille vide ; e.g. grille page précédente) (b) L'affichage des successions d'états du jeu menant à une victoire pour ce joueur (IA ou non), et à défaut ceux à menant à une partie nulle. (c) L'utilisation de l'arbre par l'IA pour déterminer son choix à chaque tour de jeu. »

-creer-element(int** matrice) permet de créer un élément de type nœud qui va contenir une matrice d'état du jeu, le fils et le frère sont initialisés à NULL, le score n'est pas initialisé délibérément pour voir afficher des scores inexacts *avant d'employer* plus tard la fonction d'inscription des scores suivant MinMax pour chaque état du jeu.

-dupliquer_matrice renvoie une copie de la matrice envoyée un paramètre, ceci à une importance car vue comme un nouveau pointeur (int**) à part entière, et éviter des conflits au niveau des pointeurs.

-nbr_cases_vides renvoie l'entier correspondant au nombre de cases vides d'une matrice d'état du jeu.

-test est une fonction très importante. Elle va permettre de remplir une matrice (déjà partiellement remplie ou non) de juste **un** entier (int symbol) dès qu'on trouve une case vide. On commence pour se faire à la ***ligne 0, colonne 0*** et s'il y a un 0 à l'endroit désigné on insère l'entier puis on retourne la matrice. Si la case est déjà occupée on balayera la matrice par colonne puis on passe à la ligne suivante le cas échéant pour y trouver un 0. Les indices de ligne et colonnes i et j sont des pointeurs car on aura besoin de cette information dans la fonction suivante. Une fois la matrice remplie **d'un** entier à la case vide, on incrémente les indices i et/ ou j pour qu'au prochain appel de la fonction la recherche d'une case vide se fasse à la case juste après ; et on retourne la matrice.

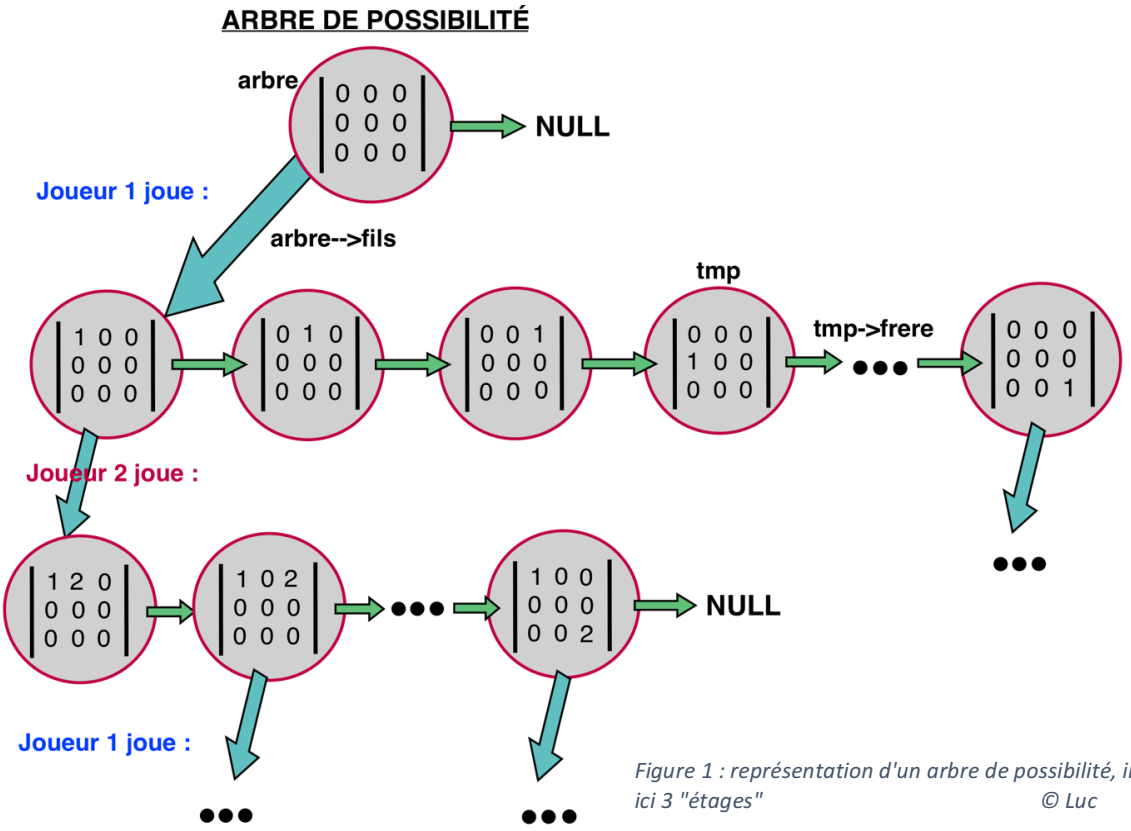

Figure 1 : représentation d'un arbre de possibilité, il y a ici 3 "étages" *© Luc*

-creer_tableau_de_matrices permet à partir d'une matrice renseignée de renvoyer un tableau de matrices (=pointeur de matrices) qui correspond en fait à toutes les possibilités de **remplissage d'une case supplémentaire** (utilisation donc de la fonction test) par un joueur de la matrice donnée en paramètre. On y voit donc déjà ici l'esquisse d' ***un étage*** de toutes les possibilités (⇔ les frères du fils d'un nœud). (Même si l'arbre n'est pas encore construit).

Si la matrice est déjà remplie ou qu'un coup gagnant a déjà été observé sur ladite matrice on ne crée aucun tableau, car inutile de donner les combinaisons possibles d'un coup supplémentaire quand l'un des joueurs à déjà gagné.

Dans le cas contraire on crée un tableau de la taille du nombre de cases vides (avec la fonction nbr_cases_vide), car il y aura autant de matrices/combinaisons/coup-possibles que de cases vides. Alors on va dupliquer la matrice donnée en paramètre pour que chaque « indice du tableau » répertorie la même matrice de départ. Puis pour chaque tableau[x] on va appliquer la fonction test en charge de remplir une case de la matrice dans l'ordre puis d'incrémenter i et/ou j (***indices inhérents à la fonction test***) pour qu'elle remplisse un endroit différent pour le tableau[x+1]. On a au final toutes les combinaisons possibles de placement d'un entier d'un joueur.

-la fonction ajouter_element ajoute un élément de type nœud comme frère d'un autre, comme à la manière d'une liste chaînée ou on ajoute en fin de liste un frère.

-**creer_descendance2 et créer_heritage2 sont les outils qui vont créer l'arbre** à partir du tableau de matrice des combinaisons pour un état.

On crée d'abord **une unique** génération suivante (**les enfants directs**) avec creer_descendance. Le pointeur de nœud donné en paramètre désigne la racine de l'arbre des possibilités, il peut s'agir d'une matrice partiellement remplie comme vide (exemple du schéma de la Figure 1). On génère alors le tableau des matrices/combinaisons possibles à partir de arbre->matrice passée en paramètre, donc ce que le joueur peut jouer à partir de la matrice initiale. On crée alors l'ainé (= premier fils, début de la lignée des frères ⇔ liste chainée) contenant la première matrice possible (donc la matrice contenue dans tableau_des_matrices[0]). Et on fait une boucle pour ajouter les suivants avec ajouter_element. On a donc à l'issue de creer_descendance la première matrice, père d'un aîné liés à ses frères.

Avec creer_heritage2 on va généraliser la fonction précédente à TOUTES les générations, et donc **créer les descendances directes de chaque frère** de la première famille créée avec créer_descendance2, **puis les descendances des frères de leur fils**, **encore et encore** jusqu'à obtenir tout l'arbre des possibilités. On utilise ainsi logiquement une fonction récursive. On veillera cependant à bien changer l'entier rentré par le joueur avec fonction_périodique (ce n'est pas tout le temps le joueur 1 qui place son entier dans la matrice).

La fonction d'affichage de l'arbre est affiche_arbre2, fonction récursive avec un paramètre de profondeur pour bien visualiser les nœuds de l'arbre et leur « parenté » (frère ou fils). L'affichage en ligne des matrices (avec la fonction affiche_ligne_matrice) permet d'améliorer cet affichage aéré.

Quelques fonctions agrémentent le programme comme nbr_etats_finaux_gagnant qui permet de connaître le nombre total de chemin qui mènerait à une victoire. (=nombre de possibilité de victoires possibles). (voir la capture d'écran ci-dessous).

.

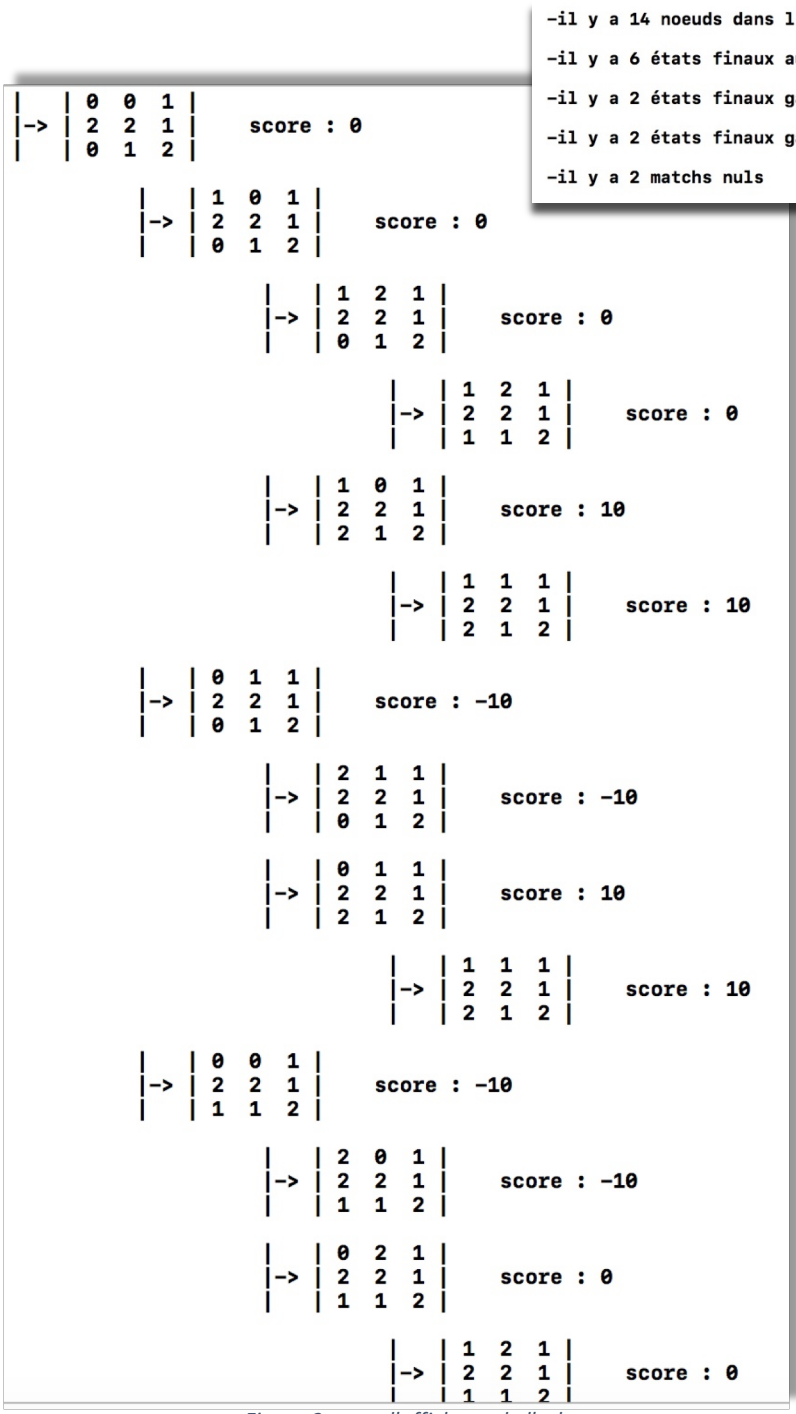

Figure 2 : test d'affichage de l'arbre

Développement de l'algorithme MinMax : MinMax sera utilisé en Résultat 1 et réexploité dans le résultat 2 pour calculer les scores dans le cas d'un arbre partiel.

Dans le Résultat 1, le score d'une feuille est donné par score_matrice : de +10, 0 ou -10 (si joueur 2 = IA gagne ou si match nul ou si joueur 1= homme gagne ⇔ IA perd).

Dans le Résultat 2 : les score peuvent prendre des valeurs entières entre -10 (échec) et +10 (victoire) avec score_matrice_etat_partiel. Ces valeurs sont arbitrairement définies. Ainsi la limite du Résultat 2 à une génération d'arbre global donnera les résultats cohérents au Résultat 1..

L'algo MinMax se découpe en 4 toutes petites fonctions dans un ordre précis :

-On initialise d'abord tous les scores de chaque nœud à -1000, une valeur « impossible. »

- scores_des_feuilles va, selon le résultat voulu (1 ou 2 de l'énoncé) entré en paramètre, inscrire toutes les feuilles de l'arbre en faisant appel à score_matrice (résultat 1) ou score_matrice_etat_partiel (résultat 2). ATTENTION : on entend par feuille, les éléments de l'arbre qui n'ont pas de fils. Sur l'exemple de la Figure 2 il y a très exactement 6 « sans-fils » = feuilles.

Ensuite, *Minmax()* s'occupe d'inscrire le score du père **Si et seulement Si** l'arbre n'est pas null, le père existe et score_inscrits_fraternite ==1 c'est à dire si tous les nœuds frères ont un score **bien définis** (différents de -1000) on peut **alors** remonter le minimum ou le maximum de ses scores selon quel joueur joue au niveau du père ! La fonction renvoie 1 ou 0 pour témoigner si oui ou non il y a eu une inscription de score vers un père.

Vient alors Minmax_entier() qui va faire un test de MinMax à partir de l'arbre renseigné en paramètre. Logiquement, on aura au début la condition dans le if qui sera à 0 (car la plupart des scores au niveau des étages supérieurs (vers la racine) sont égaux à -1000 ⇔ pas encore inscrits). On fait alors du récursif

jusqu'à obtenir un cas égal à 1. On comprend donc qu'à la fin de cette fonction on aura effectué une inscription vers un « étage » de père supérieur.

Enfin MinMax_global répète récursivement ces inscriptions tant que la racine de l'arbre est de -1000, quand la racine a une autre valeur que 1000, alors cela signifie que tout l'arbre a été inscrit de scores ;-) et MinMax fonctionne !
On notera que le minimum ou le maximum renvoyé selon que le joueur 1 ou 2 joue s'effectue par l'intermédiaire de fonction_périodique pour changer de joueur ☺

En finalité, pour l'algo de MinMax J'avoue être particulièrement fier d'avoir pu faire en seulement 4 petites fonctions de 4 lignes chacune un Algo de MinMax opérationnel.

//pour les fonctions affichage_etats_menant_vers_victoire, remplissage_par_MinMax et recherche_victoire_la_plus_rapide voir la dernière page de ce rapport.

Résultat 2 :

<u>NB sur les scores arbitraires du Résultat 2 :</u>
Pions_alignes_diagonales, Pions_alignes_lignes et enfin Pions_alignes_colonnes permettent de connaître le nombre de 1 ou 2 placés dans une colonne, diagonale ou ligne vide.
Ainsi si pions_alignes_colonnes(matrice, 1, 2) ==2 alors cela signifie qu'il y a **2 colonnes** dans lesquelles il y a **2 pions** du joueur **1** avec une case vide/libre dans ces 2 colonnes. Autrement dit : le joueur 1 pourra placer son pion dans une de ses colonnes car la case est vide, il a toute chance de gagner.
De même si pions_alignes_diagonales(matrice, 2, 1) ==2 alors cela signifie qu'il y a sur les **2 diagonales** de la matrice **un** 2 de placé et les deux autres cases sont alors vides.

Ces fonctions sont alors beaucoup plus précises que la simple connaissance d'un alignement de 3 pions du même joueur et qui témoigne d'une victoire immédiate. Ces fonctions de connaissance du nombre de pions joueurs avec degré de liberté pour placer un autre pions nous donnent beaucoup d'infos sur l'état du jeu et permettent de prévoir au mieux le **jeu avec un arbre partiel généré**, à la base même de mon IA du résultat 2.

Instructions_score en donne ainsi une matrice d'état du jeu et un joueur donné le score obtenu :

//s'il y a sur une ligne, une colonne ou une diagonale 2 pions avec possibilité d'en mettre un troisième

//s'il y a au départ ou a un moment donne 1 pion au milieu ou dans les coins, avec aucun autre adverse place sur la diagonale du pion concerne

//s'il y a sur une ligne, une colonne ou une diagonale 2 fois 2 pions avec possibilité pour les 2 d'en mettre un troisième (la partie est alors gagnée quelque soit la défense opérée par l'autre joueur car il restera un endroit de dispo pour y mettre un pion)

//si le joueur concerné gagne (3 pions alignés).

score_matrice_etat_partiel fait appel à instructions_score pour nous en donner un savant calcul en faisant la différence des scores obtenus par chacun des joueurs. Un nombre résultant négatif est donc « arbitrairement plus défavorable à l'IA qu'à l'humain ». On exclue le cas où 3 pions sont alignés pour directement retourner le +10 ou -10 du joueur concerné (et pas faire la différence avec le score de l'ennemi si jamais il a placé des pions permettant de peut être gagner).

Pour l'interface : menu_total est le menu principal du jeu, c'est la première chose qui s'affiche à l'écran quand lance le programme, qui nous permet avant même de choisir qui joue en premier grâce à menu() de connaître quel jeu on joue (aléatoire, MinMax avec arbre totalement généré, MinMax avec arbre partiel). VoirArbreJeuEtPossibilites nous permet de voir s'afficher l'arbre des possibilités de l'IA à la demande.

Resultats_TD() s'occupe de rassembler la plupart des fonctions de jeu (menus, énonciation du gagnant) en coïncidence avec les résultats voulus par l'énoncé du TD, pour décharger un peu le main.

Remarques très intéressante sur la fonction de remplissage par l'IA grâce à MinMax : MinMax crée les scores de chaque nœud de l'arbre et le nœud de la racine peut être utilisé ensuite par remplissage_par_MinMax pour retrouver le « chemin de la victoire ». Hors il a été constaté après plusieurs expérimentations de mon jeu que MinMax préférait parfois éviter que j'aligne 3 cases consécutives plutôt que de jouer le COUP GAGNANT ! En plus de ceci, MinMax préférait parfois prendre plus de temps à gagner alors qu'il pourrait le faire en moins de coup. Ce « problème » a été corrigé et était inhérent au fait que pour le score d'un arbre (retourné par le max des scores de ses enfants), il peut y avoir **plusieurs mêmes scores** chez les enfants, ainsi le chemin suivant peut être plus long et moins efficace si on se contente de demander à MinMax de trouver un fils dont le score est égal.

Pour corriger ce problème, remplissage_par_MinMax faisant appel à recherche_victoire_la_plus_rapide j'ai décidé de demander à mon IA de jouer « l'enfant qui a le même score que le père » **MAIS aussi** tel que celui-ci ait un nombre de nœuds d'enfants et petits enfants minimal. Ceci signifie qu'en jouant, MinMax se rapprochera d'autant plus vite vers la victoire, et minimise les coups inutiles ;)

Voilà ! Tout est bien détaillé. Je me suis beaucoup donné sur ce devoir, il ne me reste plus qu'à vous souhaitez une bien heureuse partie de Tic-Tac-Toe ! ☺