KERNEVES Théo

LAVEDRINE Aymeric

Projet IA

D2 – Q3

Compte rendu projet Morpion

I – Morpion 3x3

Nous avons commencé par réaliser le morpion 3x3 en implémentant les fonctions suivantes :

- `gagnant(morpion jeu, char player)` : renvoie 1 si le joueur donné en paramètre a gagné, 0 sinon. Renvoie –1 s'il y a égalité (9 cases remplies sans gagnant)
- `jouer(morpion jeu, int i, int j, char player)` : qui renvoie 1 si la case demandée n'est pas valable, qui renvoie 0 sinon et qui affecte la valeur de la variable joueur
- `initGame(morpion jeu)` : qui initialise la grille avec des –1, indiquant les cases vides sur la grille
- `afficherGrille(morpion jeu)` : qui affiche la grille de jeu, nous avons changé les -1 en espace vide, les 0 en 'O' et les 1 en 'X'.

Nous avons également défini un type `char **morpion`.

Ensuite, nous avons utilisé ces fonctions dans notre main pour demander aux joueurs de saisir les coordonnées où ils souhaitent jouer et ceci tant que les 9 cases ne sont pas remplies ou qu'il n'y a pas de gagnant. On affiche ensuite quel joueur a gagné ou s'il y a eu égalité.

II – Morpion NxN

Il est maintenant temps de généraliser notre morpion sur des grilles de NxN avec N, la taille, saisi par l'utilisateur. Nous avons donc modifié la structure *morpion* en conséquence, on donne maintenant la taille de la grille en paramètre aux fonctions précédentes.

Pour l'implémentation de la fonction *gagnant,* nous avons hésité quant à la manière de procéder. Nous ne voulions pas écrire une fonction qui ferait trop d'itérations pour vérifier chaque ligne et colonne, au final nous n'y avons pas échappé.

La fonction *gagnant* parcourt ainsi toutes les lignes pour vérifier s'il y a une ligne qui correspond à une victoire, puis les colonnes et enfin les diagonales, et ce pour le joueur du tour actuel. Pour la vérification du cas de l'égalité, nous vérifions qu'il n'y a plus de cases vides sur la grille et qu'il n'existe pas de gagnant.

La complexité de cette fonction est de $O(n^2)$.

III – Intelligence Artificielle MinMax

L'implémentation des fonctions *min*, *max* et *Minimax* fut légèrement compliquée car dans un cas particulier, notre algorithme ne distinguait pas le cas terminal "égalité" et ne jouait donc pas à l'endroit qui lui permettait de ne pas perdre. Le problème était simplement dû à une condition de cas terminal mal formulée. Une fois rectifié, l'algorithme s'avérait imbattable.

Au moment où l'IA doit jouer, l'algorithme *MinMax* va appeler la fonction *max* qui va ensuite appeler la fonction *min* jusqu'au moment où un cas terminal est rencontré. La fonction MinMax prend également deux pointeurs en argument pour pouvoir renvoyer les meilleures coordonnées disponibles pour que l'IA puisse les placer.

Soit b le nombre de coups possibles par situation et m la profondeur maximale de l'arbre, MinMax a une complexité de : $O(b^m)$

IV – Alpha Beta

L'implémentation des fonctions *min_AB*, *max_AB* et *MinMax_AB* ont pris du temps car le pseudo-code fournis pendant la dernière séance du projet ne semblait pas fonctionner. Nous avons donc recherché plusieurs solutions différentes et finalement trouvé une façon d'implémenter le principe d'alpha beta qui nous semble correct. Nous avons également inclus un élément de profondeur pour préciser à quel niveau de profondeur l'algorithme devrait aller chercher.

Ainsi, lors de l'appel de *MinMax_AB,* les nœuds dont leur heuristique est compris entre alpha et beta et ceux dont alpha serait supérieur à beta ne seront pas pris en compte. Pour le formaliser, l'algorithme ne prend pas en compte tous les nœuds tel que : $\text{alpha} \leq h(n) \geq \text{beta}$ et $\text{alpha} \geq \text{beta}$.

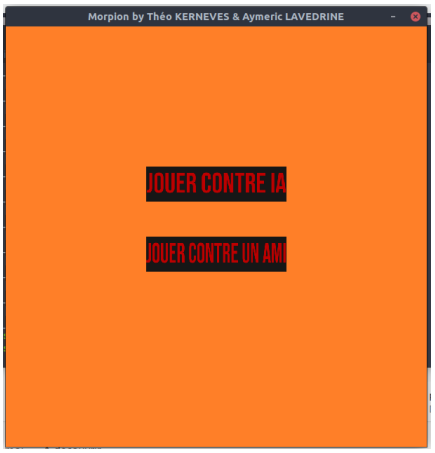
V – Organisation du programme

Nous avons décidé de garder la possibilité de jouer contre un ami ou de choisir le jeu contre une IA. Nous avons organisé les tours de jeu avec une boucle *for* qui tournera autant de fois qu'il faut pour remplir la grille. Dans cette boucle nous avons un boucle *do while* qui va vérifier si les coordonnées entrées par l'utilisateur sont correctes et redemander de jouer tant qu'ils ne le sont pas. Nous vérifions également à chaque tour s'il y a un gagnant et dans ce cas-là, nous sortons de la boucle et affichons un message de victoire.

VI – Interface graphique sur SDL

Pour créer l'interface graphique, nous avons décidé d'opter pour la bibliothèque graphique SDL (installer avec sudo apt-get install libsdl2-dev) car elle semblait beaucoup mieux documentée que ez-draw.

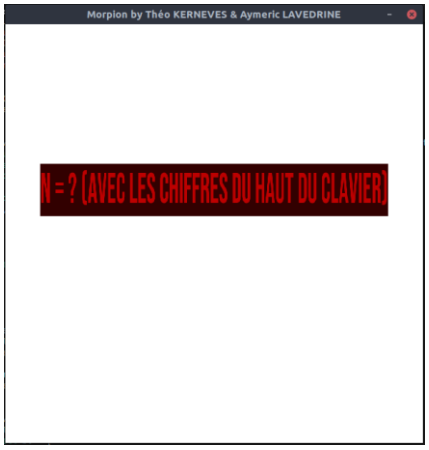
Nous avons commencé par essayer de créer le menu principal constitué de deux boutons : "Jouer contre IA" et "Jouer contre un ami".



Il n'y a pas de fonction déjà prévue dans SDL pour créer des boutons. Nous avons donc créé une fonction *drawButton* qui génère un bouton aux coordonnées (x, y) de taille w*h avec un texte rouge au milieu. Tout ça donné dans les paramètres. L'affichage de texte n'est pas nativement prévu dans SDL, nous avons donc dû installer et se servir de la bibliothèque "SDL_ttf" (installer avec `sudo apt-get install libsdl2-ttf-dev`).

Ensuite nous récupérons les coordonnées du clic de la souris et nous vérifions si ces dernières sont à l'intérieur des différents rectangles. En fonction du rectangle cliqué nous pouvons charger la partie.

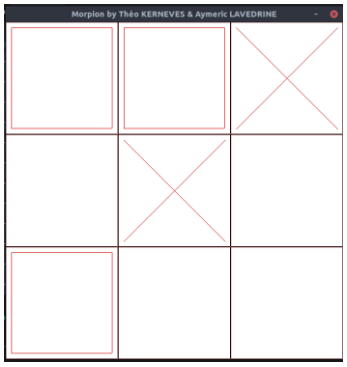
Maintenant, il faut demander à l'utilisateur sur combien de lignes et colonnes il veut jouer :



Cette partie de l'interface graphique était particulièrement difficile. Nous n'avons pas trouvé de méthode avec sdlib pour récupérer une entrée au clavier comme on aurait fait avec scanf. Nous avons donc adopté la méthode suivante :

-> On récupère le code de la touche et on utilise la fonction *SDL_GetKeyName* qui nous renvoie le nom de la touche. Par exemple la touche "&" a le nom "1", "é" le nom "2" etc... Et on convertit ce dernier en int. C'est pour cela que nous ne pouvons pas utiliser le pavé numérique dans notre interface graphique car ce sont des codes de touche complètement différents.

On peut maintenant afficher la grille avec *drawGrille* et jouer. Nous avons arbitrairement choisi de dessiner des carrés au lieu de dessiner des ronds pour le joueur n°1. La partie un peu difficile a été de convertir les coordonnées des clics de souris qui sont entre 0 et 600 (Taille de la fenêtre) en coordonnées i et j du morpion qui sont entre 0 et N.



VII – Conclusion

Pour conclure, nous avons trouvé ce projet très intéressant car nous étions vraiment en autonomie et le jeu du morpion est un cas concret sur lequel s'appuyer. Cependant, nous avons eu un manque de temps pour peaufiner le code. Nous gardons aussi quelques regrets quant à l'interface graphique qui s'est avéré beaucoup plus difficile que prévu et donc assez simpliste. Nous sommes néanmoins fiers d'être arrivés au bout du projet.