Rapport de projet de jeu :Aligne 4

Avant toutes première fonction ou programme il convient de transcrire les consignes en objectifs :

-Programmer la mécanique du jeu
-Saisit mécanique ou à la souris
-Plusieurs adversaire donc plusieurs modes de jeu, joueur contre joueur, joueur contre aléatoire, joueur contre AI etc. (cela implique une interface de jeu qui permet de choisir le mode de jeu, obvious…)
-l'interface utilisateur avec au choix : le terminal, ncurses ou SDL.

Après les consignes générales passons aux détailles spécifiques à notre jeu :

-Le jeu se déroule sur une grille 4x4.
-Il y a 16 types de pièces (chaque pièce est unique)
-Chaque pièce possède 4 caractéristiques binaires :
- sa couleur (noire ou blanche) ;
- sa taille (grande ou petite) ;
- son type (animal ou végétal) ;
- sa forme (ronde ou carrée).

-Condition de victoire : 4 alignement de pièces ayant une caractéristique commune
-horizontalement OU verticalement OU en diagonale

-Condition de match nul : la grille est remplie + aucun alignement gagnant

---

Partie élémentaire et vérification des conditions de victoire

J'ai donc commencé d'abord par définir une structure pièces qui contient 4 entier : couleur, taille, type et forme qui prendront les valeurs 0 ou 1 (et puis -1 aussi mais je n'en suis jamais servis, c'est juste une précaution que j'ai pris pour ne pas avoir des pièces qui contiennent des variables indéfinies).

Ainsi on peut définir désormais la structure ttableau qui va nous servir de grille. Cette structure est aux faites composées d'un tableau de structure pièces d'une taille de 16 cases, et puis d'un autre tableau d'entier de 16 cases qui va représenter l'état d'occupation d'une case de la grille.

Donc pour illustrer un peu, notre grille va ressembler à cette matrice 4*4, mais il n'est pas plus avantageux et facile d'utiliser un double tableau 4*4 donc autant rester sur un seul tableau de 16 cases.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Après quelque fonction élémentaire basique du type effacer_tableau(), placer_piece() etc.
On arrive à la fonction cases_occupe_de_la_table() qui mérite quelque explication.
Inévitablement nous allons faire des fonctions qui va parcourir toute la grille et y faire des tests logiques, mais a-t-on besoin de faire absolument tous les tests logiques sur chaque case ?
Notamment quand on sait qu'on va effectuer des tests sur les cases seulement occupées de la grille,
Ainsi j'ai défini cette fonction qui prend un pointeur de structure tableau et un tableau d'entier.
Plus tard vous verrez que j'ai défini de nommer ce tableau d'entier par « resultat_intermédiaire »,
Avec la fonction malloc je vais donner à ce tableau l'emplacement mémoire de 17 entiers.
Pourquoi 17 ? Parce qu'il se peut que notre grille soit entièrement remplie donc il nous faut déjà 16

cases, mais aussi faut prendre en compte le fait que ce tableau sera quasiment toujours à moitié remplie, je vais introduire une case en plus à la position n°0 qui va nous servir d'indice de taille.
Par exemple les numéros 1, 3, 5 de la grille sont remplie, on s'attend que le tableau resultat_intermédiaire soit définit comme :
[3], [1], [3],[5],[A]….[Z]
la case n°0 va indiquer le nombre de cases utiles, ainsi quand on veut parcourir les cases occupées de la grille , il nous suffiras de faire un boucle for (i = 0, i < resultat_intermédiaire[0], i++), les cases [A]-[Z] existe mais peuvent contenir n'importe quoi car non initialisé, on va les éviter par ce mécanisme d'indice n°0.

Bref c'est juste une petite détaille d'optimisation qui va rendre les fonctions futures un tout petit peu plus rapide.

On arrive maintenant à une fonction cœur de mon projet check_aligne_annexe(), qui prend en argument le tab resultat_intermédiaire, un entier tête, un entier indentation du pas ou marche et enfin un entier nombrealigné.

Il s'agit d'une boucle for qui va tourner 4 fois (on va aller sur 4 cases de la grille), et si la grille est occupée le compteur sera incrémenté, à la fin de la boucle, si le compteur atteint le nombre désiré de nombre d'alignement la fonction retourne 1 sinon 0 dans le cas contraire.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Il faut remarquer que peu importe l'alignement horizontale, verticale ou diagonale, ordinateur va exécuter la même démarche, il nous faut d'abord une tête, par exemple pour alignement horizontale de cases 0,1,2,3 la tête est 0, pour la diagonale 0,5,10,15 la tête est aussi 0. On fait intervenir maintenant la marche, qui va incrémenter d'une valeur constante, il vaut 1 pour une horizontale, 4 pour une verticale et puis enfin 3 ou 5 pour les 2 diagonales.
Par exemple pour la verticale 2,6,10,14 ; il suffit de donner la tête 2, le pas 4 et l'ordinateur va faire incrémenter le curseur d'un pas de 4.
2>> 2+4 >> 6>> 6+4>>10>>10+4>>14
Cela se resume par l'équation $A + xB = Z$ où A est la tête, B la marche, x un entier positif et Z le numéro de case souhaité.

Rq1 : j'avais d'abord fait une fonction check_aligne_annexe() spécifique au cas où on recherchait un alignement de 4 pièce et par la suite il m'a fallu avoir une fonction qui va rechercher un alignement 3 pièces puis 2 pièces d'où aspect générale de cette fonction.

Rq2 : j'ai décidé de séparer en 2 la vérification d'alignement et la vérification de caractéristique commune car nous n'avons pas forcément besoin d'aller vérifier les caractéristiques si on a même pas le nombre suffisant de pièce aligné.

Ce qui va relier ces deux parties c'est un tableau d'entier que j'appelle « resultat_finale » constitué de 5 cases :
[0] : contient un booléen qui indique si nous avons un aligne ou pas
[1] : contient la tête d'alignement
[2] : contient le pas d'alignement (donc donne aussi le type d'alignement hori, verti ou en diago)
[3] : contient la caractéristique commune aligné
- 0 si aucune caractéristique commune aligné

- 1 si couleur
- 2 si taille
- 3 si type
- 4 si forme

[4] : contient le sous type aligné, par exemple si la couleur est alignée alors 0 indique noir,1 indique blanche

Passons maintenant à la partie des vérifications de caractéristique commune :
Les fonctions check_couleur, taille, type, forme sont quasiment identique, ils supposent que nous avons un nombre de pièces alignés puis grâce à un boucle for et du tableau resultat_finale défini précédemment, on va parcourir la rangée 4 fois, et grâce à la tête et au marche de pas, on accède aux pièces alignées puis on va faire la comparaison de sous type, si le test est valide le compteur est incrémenté. Enfin à la sortie de la boucle, on compare le compteur et le nombre aligné souhaité afin de retourner une réponse booléenne.

La fonction check_finale_aligne4 va traiter les cas de succès mais aussi d'échecs d'alignement

Et si toutes les tests ont été validé, on peut supposer que le tableau resultat_finale est entièrement remplie et on peut faire appel à la fonction validation pour vérifier le tout une dernière fois

Les fonction check_horizontale4,verticale4,diagonale4 sont quasi identique, on faire d'abord passer le test d'alignement et s'il est valide, le tableau resultat_finale est remplie avec les valeurs correspondante puis on lance la vérification de caractéristique via check_forme_finale() .
Enfin si la fonction validation() est validé on peut afficher un message de félicitation et lancer la procédure_de_fermeture_du_jeu() qui va libéré la mémoire des tableaux resultat_intermédiaire et resultat_finale et fermer le programme via la fonction exit(EXIT_SUCCES) ;

Si toutes les tests précédent sont refusé il ne nous resteras plus que deux cas possible :
- Soit il reste de la place sur la grille et on peut passer la main au joueur/AI suivant.
- Soit la grille est remplie et il y a un match nul, s'il on est dans ce cas, on affiche le message correspondant et on peut fermer le programme.

Partie joueur

Dans cette partie on va s'intéressé à aux mode de jeu joueur contre joueur, et plus précisément à la partie saisie de pièce au clavier.

Il n'a que 2 contraintes à respecter lors d'une saisie de pièce sur la grille :
- la case doit être vide
- la pièce ne doit pas être déjà placé (chaque pièce est unique)

Pour tout ce qui a en rapport avec la saisie au clavier, j'utilise la fonction boucle_de_saisie qui prend 2 entier a et b en argument, tant que l'utilisateur ne saisit pas un entier compris dans l'intervalle [a ; b], on va continuer à lui demander de saisir un entier.

Rq : pour tout ce qu'il est en rapport avec l'affichage des caractéristique j'utilise l'affectation conditionnel, il s'agit de test répétitif et booléen, cela permet réduire un peu l'usage d'if dans les fonctions.

La fonction verifier_piece_utilise vérifier si une pièce est déjà utilisée ou pas, il faut donner en argument les caractéristique correspondante et la boucle for va itérer dans la grille afin de comparer si cette pièce est déjà présente ou non, j'utilise ici le tableau resultat_intermédiaire car j'ai juste besoin de parcourir autant de fois qu'il a de pièce sur la grille et seulement les cases où une pièce est présente, bref c'est juste une détaille d'optimisation.

Jusqu'ici on a quasiment terminé l'action du joueur, le game mode joueur consiste juste à saisir la pièce, vérifier l'alignement gagnant et puis deux trois détaille d'affichage sans importance.

Partie Aléatoire

Toute la partie se résume à tirer au sort des entiers aléatoirement jusqu'à trouver une pièce non utilisé puis le placer sur la grille.

Pour cela on va utiliser les fonctions rand(), srand() et time(). D'où inclusion des bibliothèque <stdlib.h> et <time.h>.
Le principe de l'aléatoire n'existe pas vraiment en programmation, l'ordinateur va simplement donne des nombres fixes en fonction d'une seed, si on utilise toujours la même graine, on aura toujours la même suite de réponse. D'où utilisation de la fonction time(NULL) qui retourne le nombre de secondes passé depuis le 01/01/1970 (cf. https://www.tutorialspoint.com/c_standard_library/c_function_time.htm).
On a ainsi des graines toujours différente et donc une 'réponse aléatoire' toujours différente aussi.

Ainsi avant tout utilisation de la fonction tirage(),il faut d'abord tirer une graine avec srand((unsigned)time(NULL)).

La fonction précédente se résume à l'équation suivante :

$$\frac{rand(\ \ )}{RAND_{MAX}} * (Max + 1)$$

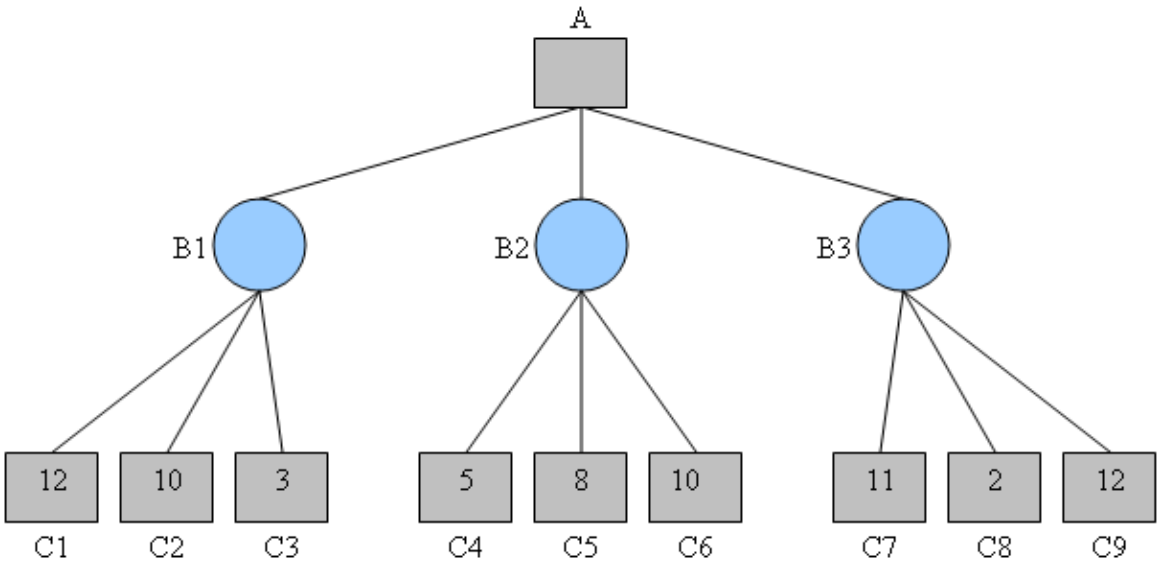
rand() va générer un nombre compris entre 0 et RAND_MAX, ainsi la partie rouge de l'équation va nous donner un nombre décimale positif compris entre 0 et 1, enfin on multiplie le tout par (max+1) qui va donner un nombre décimale compris entre 0 inclus et (max+1) exclus. Par la suite la fonction va se charger d'elle-même de tronquer cette valeur afin d'obtenir un entier compris entre 0 inclus et (max+1) exclus ou max inclus, c'est la même chose.
Exemple : 0.89 * 10 >> 8.9 >>(troncature par la fonction elle-même)>>8

à partir de tous ces éléments, on a quasiment fini la partie aléatoire.
saisie_piece_random() consiste simplement à générer les caractéristiques de la pièce jusqu'à trouver une pièce vierge et de la placer.

Partie AI lv1

Dans cette partie on va s'intéressé aux comportement d'AI, j'ai d'abord fait des recherche sur les AI d'autres jeu comme le morpion ou le puissance 4, ils utilisent plus couramment les algorithme MiniMax ou une variante AlphaBeta, mais après réflexion ces 2 algorithmes ne conviennent pas vraiment à notre projet car le jeu est beaucoup trop complexe, imaginons le cas où il reste 10 cases vide sur la grille, pour faire tourner algorithme, il faudra noter les 10 cases en plus de créer une note

annexe pour chaque pièce qu'on peut placer sur cette case, mathématiquement on se retrouve avec déjà 100 possibilité rien que pour un seul nœud, si on essaie de simuler l'action du joueur suivant, on atteint 9*9= 81 notes, ce qui nous fait 8100 actions de notation à réaliser avant même de comparer les notes pour y sélectionner la meilleur action.



(https://fr.wikipedia.org/wiki/Algorithme_minimax)

Par quelque approximation je suis arrivé à la conclusion qu'une partie moyenne se joue à environ 6-8 coups si chaque joueur joue parfaitement, car si 2 pièces qui ont 1 ou plusieurs caractéristiques sont aligné cela crée une case 'interdit' associé à 1 ou plusieurs pièces vierges.

Rq : le jeu n'est pas une puissance 4 ou morpion version 2.0 car il est exponentiellement plus complexe et puis surtout l'espérance du joueur 2 n'est pas nul, l'espérance des 2 joueurs sont quasiment identiques. Tout le monde peut espérer gagner ou pousser le match nul avec un comportement antijeu.

Du coup j'ai changé ma méthode de résolution plutôt de se compliquer à faire un AI qui met trop de temps à exécuter, je vais programmer non pas 1 mais plusieurs niveaux de AI, chacun hériterons des évolutions du niveau précédente et ainsi de suite.

Pour commencer donc, on peut déjà se servir du mode de jeu aléatoire comme base, car il peut arriver que AI ne trouve pas de coup particulièrement intéressant à jouer, autant laisser la main aux dés. Ainsi si AI ne sait pas quoi faire, il jouera aléatoirement.

Pour le niveau 1, j'aimerais apprendre à mon AI de jouer le coup gagnant, c'est-à-dire :
-trouver un alignement de 3 pièces qui ont une caractéristique en commun
-vérifier si la 4ème case est vide
-si oui, chercher si une pièce est jouable sur la 4ème case afin de remporter la manche

Pour la première partie nous avons quasiment le code déjà prête, il suffit d'utiliser encore une fois check_aligne_annexe() pour trouver un alignement de 3 cases, puis si le test est positif on lance check_forme_AI() qui utilise les fonctions de vérifications de caractéristique précédemment utilisé pour le cas de 4 alignement en combinaison avec check_nombre_caracteristique_utilise() qui retourne une booléen en fonction du caractéristique commun aligné et surtout si cette caractéristique peut encore être placé.

Par exemple pour la grille suivante nous avons la couleur rouge pour toute les cases en rouge, il n'est pas intéressant de placer une pièce en 0 car même si on a un alignement horizontale rouge, il n'existe plus de pièce rouge dans la réserve d'où intérêt de passer la main vers la vérification d'un autre caractéristique.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Par la suite si nous avons un tableau_finale valide, il va faire appel à la fonction generer_et_placer_piece_definie() qui est constitué d'une boucle infinie tant que la pièce adéquate n'a pas été générer. Enfin la pièce est placée.

Dans le cas où il n'y a pas de 3 pièces de même caractéristique aligné, on fera appel la saisie aléatoire.

Partie AI lv2 &&lv3

Notre AI sait désormais jouer le coup gagnant, maintenant on va s'intéresser aux cas où il n'existe pas de coup gagnant et optimiser son action quand il n'y a pas de coup dominant.

Je me suis basé une fois de plus sur check_aligne_annexe() et ses fonctions annexe comme check_horizontale_AI() par l'intermédiaire de la fonction recherche_2alignement_et_placement qui va chercher un alignement de 2 pièces ayant un caractéristique en commun, puis fera appel à la fonction Deux_alignement_placement_AI(), cette fonction est similaire à la fonction recherche_alignement() à la différence près qu'ici on plutôt chercher volontairement à bloquer l'alignement en y plaçant une pièce qui a le sous-type opposé.

Rq : je préfère employer ici une boucle for finie avec un entier permissivité en argument, c'est-à-dire qu'on va tenter de générer la fameuse pièce en question mais avec un nombre de tentative limité, à cela rajoute le cas où jouer une pièce d'antijeu nous fera perdre la partie comme c'est dans le cas suivant où choisir de bloquer l'alignement rouge avec une pièce bleu nous sera défavorable :

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Si le nombre de tentative atteint 0, on fera appel à notre bon vieux saisie_piece_random().

Enfin pour le niveau 3 on fera appel saisie_piece_antijeu() qui est une version un peu plus raisonnée mais beaucoup plus lente que la précédente, en effet on va chercher à générer une pièce aléatoirement sans provoqué une alignement de 3 pièces identiques.

J'utilise toujours une boucle for avec 32 comme le nombre de tentative, en soi je pourrais très bien m'aider de la notation binaire sur 4bits pour faire une boucle qui tourneras au maximum 16 fois, mais cela n'est pas très intéressant car j'aimerais programmer un AI qui joue le mieux possible sans que cela devienne trop statique avec un comportement trop antijeu, d'où ma préférence pour l'aléatoire même si cela alourdi considérablement le temps de calcul de l'algorithme.

Rq : un des critiques que je me ferais c'est d'avoir utilisé une structure ttableau comme variable local, si j'avais choisi de le définir comme une variable globale j'aurais évité de faire rentrer en argument la structure ttableau…

Bref j'ai trouvé le projet sympa à réaliser car il est plus complexe que je le croyais, de plus je me suis trop focalisé sur les fonctions élémentaires, en fin de compte cela aurais été plus fluide de définir d'abord les grandes parties puis faire chaque élément un par un plutôt que de faire l'inverse et faire l'assemblage à la fin…

Sinon le projet étais sur mon GitHub et mise à jour d'une façon assez régulière disons, voici le lien vers l'entrepôt : https://github.com/bk211/fac/tree/master/LearningC/TD_P8/aligne4