QUARTO

IPROG - Projet 2018

Justificatif technique

École Nationale Supérieure de Cognitique

MAHAUT Matéo
NOYERIE Constance

Bordeaux INP
ENSC

Sommaire

Sommaire **2**

Choix techniques **3**

Stockage des données 3

Précisions sur le stockage 3

Intérêts du binaire 3

Affichage 4

Déroulement **5**

Détails sur l'IA **7**

Choix de méthode 7

Fonctionnement 7

Contraintes 8

Choix techniques

Stockage des données

Le codage du QUARTO nécessite de stocker de nombreuses données. Certaines sont modifiées au cours de la partie - le plateau, les pièces encore disponibles - et d'autres non - les caractéristiques de chaque pièce (couleur, taille, forme, remplissage) et leur affichage.

Précisions sur le stockage

Pour les pièces, nous avons choisi de les stocker dans le tableau int[16] **piecesDispo** sous forme de binaire à 4 digits, un par caractéristique :

Attribut / Digit	couleur	taille	forme	remplissage
0	bleu	petit	rond	vide
1	jaune	grand	carré	plein

Les pièces sont classées par ordre croissant au regard de leurs 3 derniers digits mais 2 pièces dont seule la couleur diffère sont placées l'une après l'autre. Nous avons fait ce choix pour simplifier l'affichage des pièces : on n'utilise que les 3 derniers digits qui correspondent aux strings d'affichage stockées dans le tableau str[8] **affichage**.

> Remarque : **affichage** est une variable publique car il n'est jamais modifié et qu'on a besoin d'y accéder depuis les fonctions d'affichage, elles-mêmes appelées dans la plupart des fonctions.

L'affichage de la couleur est traité directement dans le sous-programme d'affichage **AfficherPiece**() avec le premier digit.

Lorsqu'une pièce est utilisée, sa valeur est remplacée par -1 dans **piecesDispo**.

La variable **plateau** est un tableau int[4][4] qui contient les binaires des pièces jouées.

Intérêts du binaire

L'utilisation du binaire pour les pièces présente deux avantages :

- accès immédiat au rang des pièces dans les tableaux **affichage** et **piecesDispo** via une simple conversion réalisée par **TraduireBinVersDec**()
- facilité des opérations de comparaison

Affichage

Le défi de l'affichage était de rendre le jeu le plus intuitif possible, afin qu'un utilisateur quelconque puisse jouer au QUARTO sans pré-requis autre que la connaissance des règles du jeu.

L'affichage des pièces est réalisé en 2D, assez grand pour être lisible et avec les quatre caractéristiques immédiatement apparentes.
Pour améliorer le confort visuel faciliter le repérage, le plateau se présente sous forme d'un damier dont les lignes et colonnes sont numérotées de 0 à 3 pour la saisie des emplacement.
Dans un souci d'utilisabilité, les consignes apparaissent à chaque étape. Ainsi, le joueur n'est pas perdu et n'a pas besoin de se référer à la documentation pendant sa partie.

Nous avons également mis en place un menu dynamique, dans lequel l'utilisateur se déplace avec les flèches du clavier, qui permet de choisir le niveau de difficulté de l'ordinateur et le tirage au sort pour savoir qui commence la partie.

Enfin, l'écran est régulièrement rafraîchi : dès qu'une nouvelle pièce est jouée, les plateaux précédents sont supprimés, ce qui permet au joueur de n'avoir que les informations utiles sous les yeux.

Déroulement

1. **LancerMenu**() : Fonction de lancement du programme. Permet de choisir le mode de jeu, et de recommencer à jouer une fois une partie terminée. Cette fonction peut appeler :
   JouerRandom(), **JouerIntelligent**(), ou **JouerHumain()** selon le mode choisi.
   Cette fonction ne prend aucun paramètre et ne renvoie rien

2. **JouerPileOuFace**() : Permet de choisir le joueur qui commence de manière aléatoire, tout en restant ludique.
   Ne prend aucun paramètre, renvoie un booléen, True si le joueur commence, false si l'ordinateur commence

On entre ensuite dans la partie à proprement parler.

JouerRandom() : Cette fonction lance une partie en faisant jouer le joueur contre l'ordinateur qui choisit ses coups aléatoirement.

Quand elle tourne, on observe dans l'ordre :

3. Une boucle qui fait jouer le joueur et l'ordinateur chaque tour (selon l'ordre déterminé à pile ou face). La boucle est coupée par une victoire ou une égalité (plateau plein)

4. tour du joueur :
   - l'ordinateur renvoie une pièce et la remplace par -1 dans **piecesDispo** avec **ChoisirPieceAlea**(). Cette fonction choisit aléatoirement une pièce disponible de la pioche, et l'en supprime. Elle ne prend pas de paramètres, et renvoie la pièce choisie dans l'encodage mentionné précédemment.

   - le joueur choisit l'emplacement avec **ChoisirEmplacement**(). Cette fonction permet au joueur de choisir où il veut placer sa pièce sur un plateau donné. Elle ne prend pas de paramètres et renvoie un couple de coordonnées.

   - **PlacerPiece**() range la pièce dans la variable plateau et vérifie s'il y a une victoire sur le plateau. Elle prend en paramètre le plateau actuel, les coordonnés, et la pièce à placer. Elle renvoie True en cas de fin de partie (victoire/défaite), et False sinon.

   - **AfficherPlateau**() affiche le plateau modifié sur la console. Prend le plateau actuel en paramètre.

5. le joueur choisit une pièce avec **ChoisirPiece**() : Cette fonction prend **PiecesDispo** en paramètre et renvoie la pièce choisie encodée sur quatre chiffres.

6. l'ordinateur choisit l'emplacement avec **ChoisirEmplacementAlea**(). Prend le plateu actuel en paramètre et renvoie un couple de coordonés.

7. **PlacerPiece**() puis **AfficherPlateau**() sont à nouveau appelés pour placer la pièce de l'ordinateur cette fois

8. Si on a une fin de partie, il y affichage du message de victoire ou d'égalité. Sinon, la boucle itère à nouveau.

JouerIntelligent()

ChoisirPieceAlea() et **ChoisirEmplacementAlea**() sont remplacées par **ChoisirCoupMinMax**(), qui s'occupe des deux choix.

Remarque : dans **JouerIntelligent**(), tous les coups joueur sont dans la même partie du code pour pouvoir regrouper tous les coups de l'ordinateur dans le même sous programme MinMax.

le déroulement des étapes est le même qu'avec **JouerRandom**()

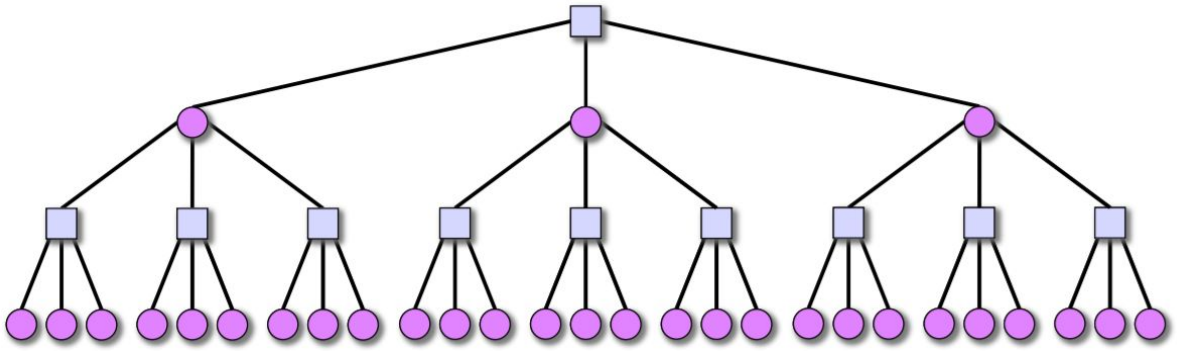
JouerHumain() :

ChoisirPieceAlea() et **ChoisirEmplacementAlea**() sont remplacées par **ChoisirPiece**() et **ChoisirEmplacement**(), qui sont donc appelés deux fois dans la fonction.

Entre les deux joueurs, un écran explicite clairement quand il faut changer de joueur.

Finalement, le sous-programme **TraduireBinVersDec**() est utilisé à l'intérieur des fonctions d'affichage pour permettre un accès plus rapide à la pièce voulue dans le tableau public string[] **affichage**.

Détails sur l'IA



Choix de méthode

Pour faire jouer l'ordinateur de façon "intelligente", nous avons choisi de mettre en place un algorithme min max. Ce type de méthode est pertinente dans un jeu comme celui-ci où 2 joueurs sont en compétition, chacun ayant toutes les informations sur la partie en cours (on peut aussi utiliser ce type d'algorithme pour aux échecs, au puissance4, …). De plus, nous avons déjà légèrement abordé cette méthode en prépa, même si c'était dans un autre contexte.

Fonctionnement

À chaque appel, l'algorithme min max teste toutes les possibilités de choix de pièce et d'emplacement et choisit le meilleur chemin vers la victoire. Dans notre cas, il s'agit du chemin qui utilise le moins de coup, l'autre option étant de mettre en place un système d'attribution de score à chaque grille, ce qui au QUARTO aurait été difficile car les deux joueurs partagent toutes les pièces. En effet ce qui est bénéfique pour l'un est donc bénéfique pour l'autre, et on a du mal à déterminer qui a l'avantage en cours de partie. Cet algorithme suppose que son adversaire joue lui aussi ses meilleurs coups.

La fonction commence par essayer de poser la pièce fournie par l'adversaire en chaque position possible sur le plateau. Si ce positionnement mène sur une fin de partie (victoire, déterminée par la fonction **PlacerPiece**()) la fonction renvoie ces coordonnées et s'arrête. Sinon elle continue, et simule l'action de donner chacune des pièces disponibles à l'adversaire en s'appelant elle-même avec ces nouveaux paramètres. Elle garde ensuite la position et la pièce qui lui ont permis d'avoir le meilleur score, c'est à dire de tomber sur une victoire le plus rapidement.

Pour les raisons expliquées dans la section suivante, parfois la fonction ne trouve aucune pièce qui lui semble intéressante. Dans ce cas, pour pouvoir continuer son exploration, elle prend la première pièce et le premier emplacement, en lui mettant le pire score possible afin qu'il soit remplacé par n'importe quelle autre option si elle se présente.

Lorsque la fonction simule son propre coup, ou quand elle simule le coup du joueur, elle doit se comporter différemment. Dans son propre coup elle veut maximiser le score possible (i.e. trouver la victoire la plus rapide), alors que dans le coup de l'adversaire elle cherche à le minimiser. Cette différence est gérée par la variable booléenne **minOrMax**, qui vaut True lorsque l'on maximise, et False lorsque l'on minimise. Elle détermine la valeur à prendre en fonction de la parité de profondeur (chaque joueur joue un coup sur deux) à chaque appel récursif en début de fonction.

Contraintes

L'algorithme min max peut poser quelques problèmes, notamment son temps d'exécution important pour une profondeur maximale, c'est-à-dire quand la partie est simulée jusqu'à la fin. En effet, au premier coup, il y a 16 cases vides et 16 pièces disponibles, au deuxième 15 de chaque. On a donc en tout 16! x 16! options différentes à essayer, ce qui donne un résultat de l'ordre de 10^{26}, qui au mieux, et avec un processeur de 3 Ghz, mettrait 100 Milliards de siècles à être calculé. Même en optimisant fortement la fonction, et en divisant le temps de calcul par deux, on reste hors des temps raisonnablement envisageables.

On propose donc comme solution de n'autoriser, dans les premiers coups, que l'ordinateur ne cherche à prédire qu'un certain nombre de coups, en introduisant le paramètre **observation**. La fonction continue de chercher les coups suivants, tant que la profondeur de récursivité reste strictement inférieure à cette variable. Si elle ne trouve aucune victoire ou défaite dans aucun des chemins, elle prend une pièce par défaut comme expliqué précédemment. Le paramètre **observation** correspondant à chaque tour de jeu est déterminé empiriquement (on part du principe que les ordinateurs de correction seront aussi performant que ceux de l'école), et stocké dans le tableau **proff**. Le temps de calcul de l'ordinateur ne dépasse donc jamais une minute, et reste donc agréable à jouer. Avec plus de temps, il aurait été intéressant de créer une fonction déterminant la profondeur maximale atteignable en un temps donné pour chaque machine en utilisant le temps du système par exemple.

Un autre problème posé par le choix de l'algorithme min max est qu'à moins de vraiment savoir ce qu'il fait, ou d'avoir beaucoup de chance, le joueur ne peut pas gagner. Il ne peut en effet pas prédire autant de coups que l'ordinateur, et risque ainsi d'être frustré après quelques parties que l'ordinateur ne propose que 2 niveaux. L'un aléatoire, deviendrait vite trop facile, l'autre prévoyant tout ses coups, serait donc quasiment impossible à vaincre et trop difficile. Pour résoudre ce problème, nous avons découpé le mode intelligent en deux niveaux, intermédiaire et difficile. Le niveau intermédiaire utilise la même fonction **ChoisirCoupMinMax**(), mais au lieu de fournir la plus grande profondeur possible comme limite d'observation, la profondeur d'observation est limitée à 1 constamment dans le tableau **proff**. Ainsi, le mode de difficulté intermédiaire ne jouera jamais un coup le faisant perdre, et ne ratera pas une occasion directe de gagner, mais restera incapable de prévoir plusieurs coups à l'avance. Le joueur a donc moyen de le piéger si il réfléchit bien. Il peut ainsi s'agir d'un mode de jeu plus satisfaisant.

Pour réduire le temps d'exécution, il aurait aussi été possible de mettre en place un élagage alpha-bêta qui permet de supprimer au fur et à mesure les branches inutiles de l'arbre récursif. On aurait ainsi pu utiliser une profondeur plus élevée en un temps plus court. Une autre méthode d'optimisation aurait été la mémoïsation, c'est à dire d'enregistrer toutes les branches de l'arbre de récursion déjà explorées pour ne pas avoir à les chercher à nouveaux. Ces deux méthodes n'ont pas été implémentées ici, le temps ayant été plutôt passé à travailler sur l'aspect, ou des fonctionnalités originales. En effet, même à la profondeur restreinte choisie, le mode intelligent joue assez bien pour présenter une vraie compétition au joueur.