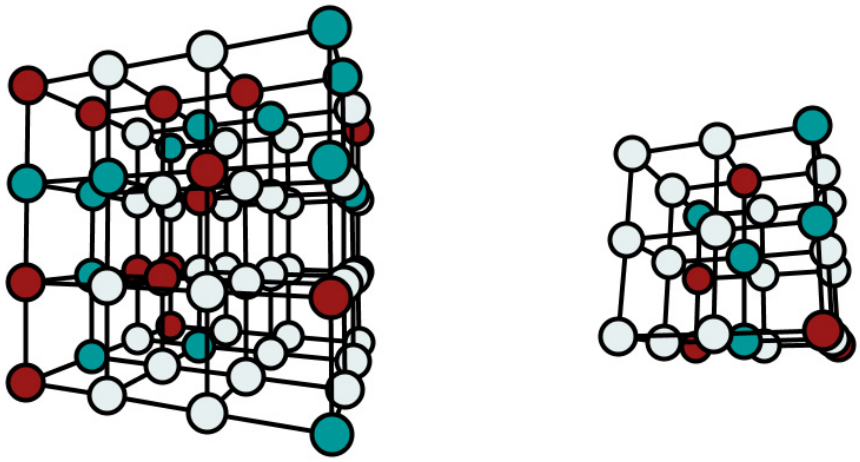
Nom: Walid KRICHENE
Groupe: 1
Date: March 14, 2008

Morpion 3D



1 Analyse, principaux algorithmes

Le projet consiste à programmer un jeu de morpion en 3D avec une IA qui utilise un algorithme minimax (simple ou amélioré avec la méthode alpha-beta) avec limite de profondeur.

Le jeu se compose d'un cube pricipal d'arête a, pavé de $(n-1)^3$ petits cubes d'arête $a/(n-1)$. On place aux sommets des petites cubes des sphères initialement de couleur blanche. Les joueurs colorient tour à tour une sphère chacun d'une couleur (joueur1: vert, joueur2: rouge). Le premier joueur qui aligne n sphères de sa couleur gagne la partie. Tous

les alignements sont possibles, y compris les grandes diagonales du cube. Possibilité de match nul si toutes les sphères sont colorées sans alignements.

1.1 Architecture et Listing des classes

Les classes sont présentées ici brièvement (seules les méthodes les plus importants sont cités). Pour plus de détails veuillez vous reporter au code source commenté.

On peut découper le programme en deux modules principaux :

- La représnetation d'une partie et l'Intelligence Artificielle
- L'affichage 3D et l'interface (fenêtres, interaction avec l'utilisateur...)

La représentation d'une partie et l'Intelligence Artificielle sont gérées par deux classes principales :

- La classe *Morpion*, dont l'attribut principal est game (de type *Game*) qui sert à représenter l'état du plateau de manière efficace (cf. Détection des lignes complètes).
- La classe *Game* : Cette classe permet de regrouper les informations nécessaires pour représenter une partie, ou un état d'un plateau de jeu. Elle fournit une méthode de mise à jour du plateau suite à un coup (update(Id id)), et la fonction d'évaluation du plateau qui est utilisée par l'algorithme minimax ou alpha-beta (eval()).

L'affichage 3D et l'interface sont gérés par trois classes :

- La classe *Interact* implémente la plupart des Listeners utilisés et appelle les fonctions de dessin fournies par la classe *DisplayableObject*
- La classe *Launch* permet de créer toutes les boîtes de dialogues, menus et labels utilisés, et gère la sauvegarde et le chrgement d'une partie
- La classe *DisplayableObject* permet d'avoir une représentation d'un objet 3D sous forme de graphe, et de fournir les fonctions de dessin adéquates.

Autres classes utilisées :

- La classe *Id*: permet de représenter les coordonnées dans un tableau à trois indices (exemple d'accès : tab[id.i][id.j][id.k] où id est de type Id)
- La classe *List* permet d'avoir une structure de pile avec les fonctions classiques *push(Object), pop(), is_empty, ...*
- La classe *Node*: permet de représenter un graphe (utilisée par la classe *DisplayableObject*)

- La classe *Line*: utilisée par *Game* pour représenter une ligne, permet de stocker le nombre se coups joués par chaque joueur sur cette ligne.

- La classe *SaveData* : implémente *Serializable* et permet de sauvegarder une partie en enregistrant les données qui permettent de recréer la partie (niveau de difficulté de l'IA, qui correspond concrètement à la profondeur d'exploration *ply*, taille du plateau (*size*), adversaire humain ou ordinateur (*pl2_is_human*), ainsi que la liste des coups joués dans l'ordre (*moves*). Pour reconstituer l'état du plateau (suite à un chargement), le programme rejoue les coups stockés dans *moves*.

1.2 Intelligence Artificielle

Le but de la partie Intelligence Artificielle a été d'obtenir une méthode qui puisse, suite à une exploration des coups possibles jusqu'à une profondeur limite, choisir un *meilleur coup possible*.

J'ai implémenté, dans un premier temps, l'algorithme *minimax* qui effectue une exploration systématique des coups possibles, en utilisant un système d'évaluation pour choisir le meilleur coup : un état g du plateau de jeu est évalué par un entier $e \in [-N, N]$ où N est le nombre total de lignes du plateau. Une évluation positive correspond à un état favorable au joueur 1, tandis qu'une évaluation négative favorise le joueur 2. Le joueur 1 (aussi noté Max) cherchera donc à choisir le coup qui maximise l'évaluation alors que le joueur 2 (aussi noté Min) cherchera à la minimiser. Considérons le cas où Max est un joueur humain et Min est une Intelligence Artificielle. L'IA suppose pendant la recherche du meilleur coup que l'adversaire choisit systématiquement le coup qui le favorise le plus, ce qui n'est pas forcément le cas en pratique. Cela permet de rechercher le meilleur *pire des cas*.

Lorsque Min doit évaluer un état donné du plateau, il appelle la fonction *max* sur tous les coups possibles (donc "demande à Max d'évaluer tous les états fils possibles") et prend le minimum de toutes ses évaluations. Max évalue d'une manière analogue (appelle min et prend le maximum de toutes les valeurs obtenues)

Dans un second temps, j'ai modifié l'algorithme pour utiliser l'élagage alpha-beta (*alpha-beta pruning*) qui introduit deux paramètres supplémentaires : α qui stocke l'évaluation du meilleur coup calculé pour Max, et β qui stocke l'évaluation du meilleur coup calculé pour Min. α est mis à jour dans les fonctions *min* et β dans la fonction *max*, et dès que $\beta \geq \alpha$, j'arrête l'exploration de la branche (je fait un cutoff ou coupure) puisque dans cette branche l'adversaire peut m'obliger à être dans une situation moins avantageuse pour moi que ce que je peux avoir dans une autre branche déjà explorée !

Se reporter à la section Performances pour une analyse plus détaillée de l'IA.

1.3 Fonctions de dessin

Une partie du projet consiste à afficher un objet en 3D représenté par un graphe de points de l'espace. L'affichage en 3D est obtenu en projetant chaque point de l'objet sur le plan de l'écran de la manière suivante: on dispose d'un repère orthonormé (O, i, j, k) de l'espace, chaque point M étant déterminé par 3 coordonnées (x, y, z). On a un point I qui représente la position de l'oeuil de l'observateur, et un plan $P \perp \overrightarrow{OI}$ (qui représente l'écran d'affichage).

On associe alors au point M le point $\{M'\} = (MI) \cap P$ (de cette manière le point "réel" se trouve sur le prolongement de la droite qui joint l'oeuil au point projeté sur l'écran).

Pour appliquer une rotation à l'objet, il suffit de multiplier chaque vecteur par la matrice de rotation correspondante avant de le dessiner.

Gestion des recouvrements : La gestion des recouvrements des spères nécessite un tri des noeuds à dessiner selon une profondeur décroissante (les noeuds les plus profonds sont dessinés en premier, pour que les recouvrements se fassent dans le bon sens : un objet plus proche recouvre un objet plus profond). Cette technique est souvent désigné par l'algorithme du peintre. J'ai donc implémenté cet algorithme pour le dessin des sphères en comparant simplement les profondeurs des centres des sphères. Par contre, le recouvrement sphère/arête est plus dur à gérer, puisque je ne savais quelles profondeurs comparer. Je me suis donc contenté, dans un premier temps, de gérer les recouvrements de sphères uniquement, et je traçais les sphères par dessus les arêtes qui étaient donc toutes recouvertes, ce qui donnait une perspective peu réaliste.

Ensuite j'ai pu résoudre ce problème en remarquant qu'une arête rattachée à une sphère de profondeur z est recouverte par toute sphère de profondeur $> z$. Donc if suffit procéder comme suit : on traite les sphères dans l'ordre décroissant de profondeur, le traitement d'une sphère S consiste à tracer toute arête SS' qui relie S à une autre sphère S', à condition que celle-ci ne soit pas traitée (auquel cas l'arête SS' aura déjà été tracée), ensuite on finit par dessiner S et on passe au traitement d'une sphère moins profonde.

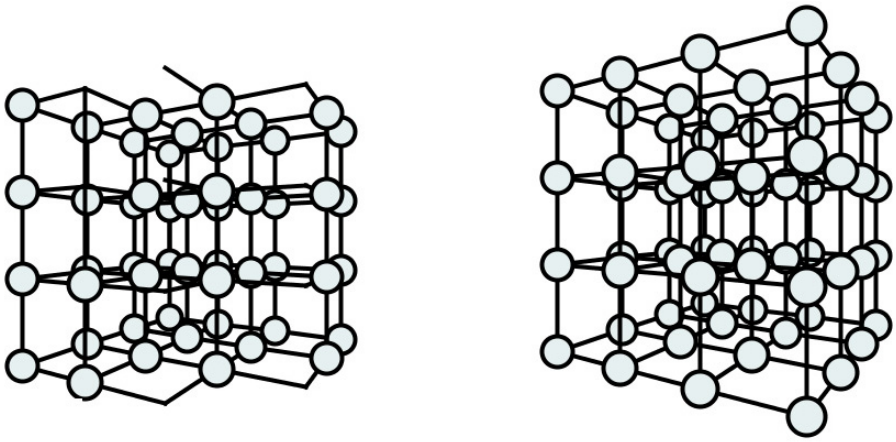


Figure 1: Etape intermédiaire et étape finale du dessin.

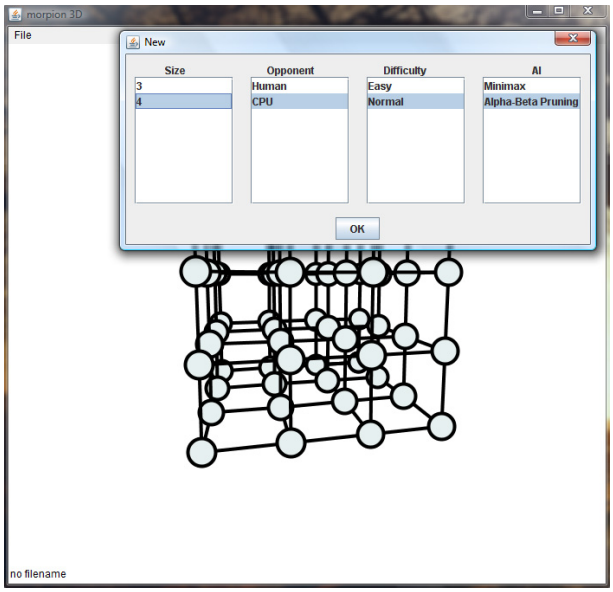
1.4 Détection des lignes complètes

Ma première approche a été de représenter le plateau de jeu par un tableau d'entiers (un entier pour chaque sphère qui vaut 0 si elle est libre, 1 si elle est occupée par le joueur 1 et

2 si elle est occupée par le joueur 2) mais cela m'obligeait à faire de nombreux accès à ces états : pour vérifier si une ligne est complète, il faut faire des parcours dans les 13 directions possibles (une sphère peut appartenir à 3 lignes non diagonales, 6 petites digonales, et 4 grandes diagonales), il est à noter que ces détections sont faites un grand nombre de fois par l'ia (une fois par coup exploré). J'ai donc réfléchi à une deuxième manière de faire cette détection, moins coûteuse en temps : il suffit de représenter toutes les lignes possibles d'une partie (de manière suffisamment structurée pour pouvoir identifier facilement les lignes qui passent par une sphère donnée, se reporter au code sources commenté de la classe Game pour une explication de la méthode que j'utilise), et de stocker pour chaque ligne le nombre de coups joués par chaque joueur sur cette ligne. Il suffit alors de mettre à jour ces nombres à la suite de chaque coup (on accède une seule fois à chaque ligne qui passe par le coup joué) et de vérifier si une ligne est complète (on compare le nombre total de coup à la taille *size*)

2 Notice d'utilisation

Après avoir lancé le programme, appuyez sur (Ctrl+n) pour lancer une nouvelle partie (une nouvelle partie se lance automatiquement avec les paramètres suivants : jeu de taille 4, contre l'ordinateur, difficulté normale avec l'algorithme alpha-beta).



Vous pouvez alors choisir la taille du plateau et la nature de l'adversaire (humain ou ordinateur), et si vous choisissez de jouer contre l'ordinateur, vous pouvez choisir le type d'algorithme utilisé par l'intelligence artificielle (minimax ou alpha_beta) ainsi que la difficulté (une difficulté plus élevée correspond à une profondeur explorée *ply* plus grande).

Une fois la partie commencée, vous pouvez faire tourner le cube du jeu grâce à la souris (en maintenant enfoncé et en glissant). Vous pouvez utiliser la molette pour modifier le zoom. Pour choisir un coup, il suffit d'appuyer sur l'une des sphères. En fin de partie, le programme vous propose d'en démarrer une nouvelle.

A tout moment, vous pouvez sauvegarder la partie en cours (en appuyant sur Ctrl+s, ou Ctrl+a), ou charger une partie (en appuyant sur Ctrl+o)

3 Performances

L'opération qui nécessite le plus de ressources est le calcul du meilleur coup par l'ordinateur. Je me suis fixé comme objectif une durée maximale de 2 secondes.

L'algorithme minimax étant très peu efficace (exploration systématique), j'ai décidé d'implémanter l'algorithme alpha_beta qui permet d'aller à une profondeur plus grande pour le même coût en temps (pour une partie de taille 4 (plateau à 4^3 sphères), j'ai réussi à passer d'une profondeur maximale égale à 2 en minimax, à une profondeur égale à 4 en alpha-beta). Le temps de calcul reste néanmoins variable en fonction de la fréquence des cutoffs et de leur endroit. Il est donc intéressant d'essayer d'explorer les coups les plus prometteurs en premier (pour augmenter les chances d'un cutoff). J'ai donc décidé de trier les coups possibles au début de chaque exploration.

La manière la plus intuitive et la plus simple de trier consiste à utiliser la fonction d'évaluation (on met les coups dont l'évaluation est la plus élevée en tête de liste). Après avoir implémenté cet algorithme et fait quelques tests, je me suis aperçu qu'il n'améliorait pas vraiment les performances (pour obtenir l'évaluation d'un enfant, le programme doit d'abord jouer le coup correspondant et faire les mises à jour nécessaires par fonction *update(Id)*, ce qui ajoute beaucoup d'opérations et compense le gain éventuel d'une coupure plus rapide). J'ai donc décidé de l'enlever.

Une autre amélioration possible consiste à utiliser des threads pour profiter du temps de réflexion du joueur humain pour faire les calculs du meilleur coup (ce que je n'ai pas encore fait).

Evaluation de l'IA, pistes d'amélioration

L'efficacité de l'IA dépend grandement de la pertinence de la fonction d'évaluation, il est en effet inutile de pouvoir prédire plusieurs coups à l'avance si, arrivé à la profondeur maximale, on se trompe dans l'évaluation du plateau !

La fonction d'évaluation que j'ai utilisée, bien que simple à implémanter, me paraît suffisamment efficace. Elle renvoie la différence entre le nombre de lignes qui sont encore "ouvertes" pour chaque joueur (une ligne est ouverte si elle ne contient aucune sphère de l'adversaire). Plus la différence est grande, plus on a de chances de gagner.

Pour l'évaluation de l'efficacité de l'IA, j'ai joué plusieurs parties contre l'ordinateur pour observer son comportement face à différentes situations. Son comportement a souvent été rassurant, et correspond à des réflexes naturels. Par exemple, dans une partie de taille impaire, il commence par prendre la position centrale. Par contre, dans une partie de taille paire, il choisit de prendre un "coin", ce qui m'a d'abord surpris, mais il s'agit effectivement de la position la plus stratégique (c'est la position parlaquelle passe le plus de lignes, dont une grande diagonale du cube).

D'autres comportements paraîssent incohérents, mais ne le sont en fait pas dans le cadre de l'algorithme: Il arrive par exemple que l'ordinateur ne bloque pas un coup gagnant de son adversaire et joue un coup "arbitraire". En fait, ce comportement est dû au fait que l'IA est "sûre de perdre", c'est-à-dire que quelque soit le coup joué par l'ordinateur, le joueur humain *peut* jouer une série de coups qui le fait gagner. Donc l'IA choisit indifféramment l'un de ces coups (le premier rencontré en fait). Cependant, si l'adversaire ne joue pas de manière optimale, l'IA peut avoir une chance de gagner.

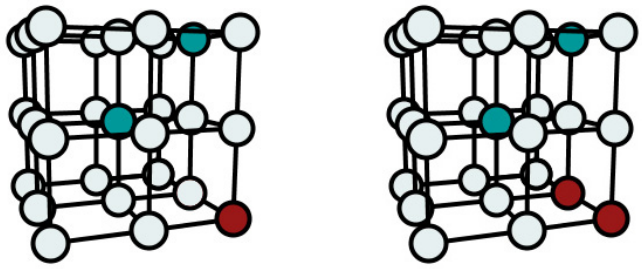


Figure 2: A ce stade du jeu, l'IA est "sûre" de perdre et choisit donc n'importe quel coup, puisque tous les coups ont la même évaluation.

Il est utile de noter que l'algorithme du minimax (et par extension alpha_beta) vise à maximiser le gain *dans le pire des cas*, c'est-à-dire que l'IA suppose que son adversaire choisit le coup qui l'arrange le plus, ce qui n'est pas forcément le cas.

Un problème similaire peut se poser, (mais qui ne compromet pas les chances de gagner de l'IA) : Lorsque l'IA est sûre de gagner, elle peut choisir de "retarder la fin de la partie" en ne jouant pas le premier coup gagnant qui se présente (ce qui est en fait frustrant pour l'adversaire humain qui retrouve l'espoir de gagner suite à ce qui semble être une erreur de l'IA, mais finit par perdre).

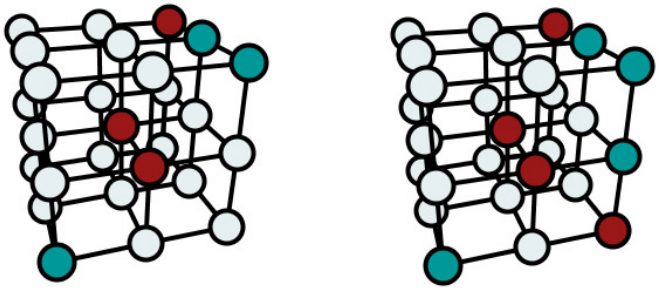


Figure 3: A ce stade du jeu, l'IA peut choisir entre plusieurs coups gagnants et ne choisit pas forcément un coup gagnant immédiat (victoire immédiate).

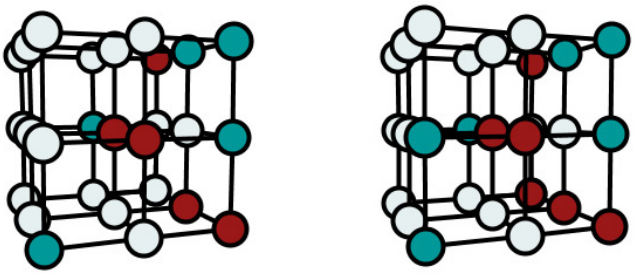


Figure 4: Ce n'est que quatre coups plus tard que la partie se termine.

Une amélioration possible serait donc de faire un système de classement supplémentaire des coups dans le cas particulier d'une défaite certaine (mais qui ne l'est en fait pas puisque c'est une défaite certaine *dans le pire des cas*) ou d'une victoire assurée, en privilégiant les *victoires les plus rapides* et les *défaites les plus retardées*.

Remarque : pour analyser le comportement de l'IA, la configuration la plus intéressante est un plateau de taille 3 (fins de parties rapides et existence d'une stratégie gagnante pour le joueur qui commence s'il prend la sphère centrale) avec une profondeur *ply* maximale (l'IA prédit très tôt une victoire ou une défaite et réagit donc de manière surprenante !)