ΠΛΗ 311 – Τεχνητή Νοημοσύνη – 2024
Πρότζεκτ Προγραματισμού

Dimitrios Keramidas AM: 2020030028

8 Φεβρουαρίου 2024

Αλλαγές σε guiServer

Δεν θα σχολιαστούς εφόσον έγιναν για λόγους local testing και δεν αλλάζουν την λειτουργία του

Αλλαγές σε client

Αρχικά προστέθηκαν επιλογές argument στο cli

[-n name (set custom name)]
Το όνομα που φαίνεται στον server όταν συνδέεται το client

[-r (use random instaid of minimax)]
Ο αλγόριθμος που χρισιμοποιείται για να βρεθεί η κίνηση

[-t (disable pruning)]
Απενεργοπειεί το ab pruning στον minimax αλγόριθμο, έγκυρο μόνο αν δεν υπάρχει το -r.

Implemented steps in series

Minimax algorithm

Advanced evaluation function

ab pruning

Minimax algorithm

```
int minimax(Position *pos, int depth, char maximizingPlayer, char originalPlayer) {
    if (depth == 0 || !canMove(pos, WHITE) && !canMove(pos, BLACK)) {
        return evaluatePosition(pos, originalPlayer,1,0,0);
    }

    if (maximizingPlayer == originalPlayer) {
        int maxEval = -100000;
        for (int i = 0; i < ARRAY_BOARD_SIZE; i++) {
            for (int j = 0; j < ARRAY_BOARD_SIZE; j++) {
                if (isLegal(pos, i, j, maximizingPlayer)) {
                    Position newPos = *pos; Move move = {{i, j}, maximizingPlayer};
                    doMove(&newPos, &move);
                    int eval = minimax(&newPos, depth - 1, getOtherSide(maximizingPlayer), originalPlayer);
                    maxEval = (eval > maxEval) ? eval : maxEval;
                }
            }
        }
        return maxEval;
    } else {
        int minEval = 100000;
        for (int i = 0; i < ARRAY_BOARD_SIZE; i++) {
            for (int j = 0; j < ARRAY_BOARD_SIZE; j++) {
                if (isLegal(pos, i, j, maximizingPlayer)) {
                    Position newPos = *pos; Move move = {{i, j}, maximizingPlayer};
                    doMove(&newPos, &move);
                    int eval = minimax(&newPos, depth - 1, getOtherSide(maximizingPlayer), originalPlayer);
                    minEval = (eval < minEval) ? eval : minEval;
                }
            }
        }
        return minEval;
    }
}
```

Παραπάνω φαίνεται ο βασικός minimax αλγόριθμος που πάει πίσω μπρος μεταξύ των παικτών και στο τέλος επιστρέφει ένα value με βάση το initial state που του δίνεται. Value παράγουν μόνο τα φύλλα του δέντρου καταστάσεων (τερματικές καταστάσεις) με τήν evaluatePosition και αυτή μετά συγκρίνεται με τις υπόλοιπες τιμές ενώ διαδίδεται πρός τα πάνω.

Έπειτα το παρακάτω επιλέγει την κίνηση που έχει το μεγαλύτερο value και πέζεται μέσα απο τις μεθόδους που μας δόθηκαν.

```
void playMinmax( void ) {
    int bestValue = -100000;
    Move bestMove = {{NULL_MOVE, NULL_MOVE}, myColor};

    for (int i = 0; i < ARRAY_BOARD_SIZE; i++) {
        for (int j = 0; j < ARRAY_BOARD_SIZE; j++) {
            if (isLegal(&gamePosition, i, j, myColor)) {
                Position newPos = gamePosition; Move move = {{i, j}, myColor};
                doMove(&newPos, &move);
                int moveValue = minimax(&newPos, 3, getOtherSide(myColor), myColor);
                if (moveValue >= bestValue) {
                    bestValue = moveValue;
                    bestMove = move;
                }
            }
        }
    }

    myMove = bestMove;
}
```

Advanced evaluation function

```
int evaluatePosition(Position *pos, char color,
                     int difweight, int mobilityweight, int stabilityweight) {
    int score = 0;
    int opponent = getOtherSide(color);

    // Basic score difference
    score += (pos->score[color] - pos->score[opponent]) * difweight;

    // Mobility
    int myMobility = 0;
    int opponentMobility = 0;
    for (int i = 0; i < ARRAY_BOARD_SIZE; i++) {
        for (int j = 0; j < ARRAY_BOARD_SIZE; j++) {
            if (isLegal(pos, i, j, color)) { myMobility++; }
            if (isLegal(pos, i, j, opponent)) { opponentMobility++; }
        }
    } score += (myMobility - opponentMobility) * mobilityweight;

    // Stability (corners and edges)
    int stability = 0;
    int c = 10, e = 5, m = 3; // Weights for diferent positions in the board
    int stabilityWeights[ARRAY_BOARD_SIZE][ARRAY_BOARD_SIZE] = {
        {  0,  0,  0,  0,  0,  0,  0,  c,  e,  e,  e,  e,  e,  e,  c },
        {  0,  0,  0,  0,  0,  0,  e, -c, -e, -e, -e, -e, -e, -c,  5 },
        {  0,  0,  0,  0,  0,  e, -e,  m,  m,  m,  m,  m,  m, -e,  5 },
        {  0,  0,  0,  0,  e, -e,  m,  m,  m,  m,  m,  m,  m, -e,  5 },
        {  0,  0,  0,  e, -e,  m,  m,  0,  m,  m,  m,  m,  m, -e,  5 },
        {  0,  0,  e, -e,  m,  m,  0,  0,  0,  m,  m,  m,  m, -e,  5 },
        {  0,  e, -e,  m,  m,  0,  0,  0,  0,  0,  m,  m,  m, -e,  5 },
        {  c, -c,  m,  m,  0,  0,  0,  0,  0,  0,  0,  m,  m, -c,  c },
        {  e, -e,  m,  m,  m,  0,  0,  0,  0,  0,  m,  m, -e,  e,  0 },
        {  e, -e,  m,  m,  m,  m,  0,  0,  0,  m,  m, -e,  e,  0,  0 },
        {  e, -e,  m,  m,  m,  m,  m,  0,  m,  m, -e,  e,  0,  0,  0 },
        {  e, -e,  m,  m,  m,  m,  m,  m,  m, -e,  e,  0,  0,  0,  0 },
        {  e, -e,  m,  m,  m,  m,  m,  m, -e,  e,  0,  0,  0,  0,  0 },
        {  e, -c, -e, -e, -e, -e, -e, -c,  e,  0,  0,  0,  0,  0,  0 },
        {  c,  e,  e,  e,  e,  e,  e,  c,  0,  0,  0,  0,  0,  0,  0 }
    };
    for (int i = 0; i < ARRAY_BOARD_SIZE; i++) {
        for (int j = 0; j < ARRAY_BOARD_SIZE; j++) {
            if (pos->board[i][j] == color) { stability += stabilityWeights[i][j]; }
            else if (pos->board[i][j] == opponent) { stability -= stabilityWeights[i][j]; }
        }
    } score += stability * stabilityweight;

    return score;
}
```

1^o: Mobility

Πόσες valid κινήσεις έχει ο κάθε παίκτης. Σέ ένα τέτοιο παιχνίδι το να έχουμε επιλογές για κινήσεις είναι πολύ σημαντικό. Για κάθε valid κίνηση δικιά μας προσθέτουμε, για αντιπάλου αφαιρούμε.

2^o: Stability

Πόσο πιθανό είναι να μην αλλάξουν τα κελιά που κατέχει ο κάθε παίκτης. Με βάση το πώς λειτουργεί το παιχνίδι κελί που είναι σε γωνίες η άκρες είναι απίθανο να το κλέψει ο αντίπαλος. Για αυτό είναι σημαντικό να παίρνουμε τα ακριανά ή γωνιακά κελιά. Επίσης πρέπει να μήν παίρνουμε τα 2α από το τέλος καθώς αυτό δίνει ευκαιρία στον αντίπαλό μας να πάρει τα ακριανά.
Για του παραπάνω λόγους δημιουργούμε τον μεγάλο πίνακα με weights για κάθε κελί.
Στο κέντρο δεν μας πειράζει πολύ και για αυτό έχει weights 0, 1
Γωνίες και άκρες είναι πολύ σημαντικές και για αυτό έχουν 10, 5
Δαχτυλίδι πρίν την άκρη του ταμπλό θέλουμε να μήν το επιλέγουμε και για αυτό έχει αρνητικό βαρος

ab - pruning

```
int minimax(Position *pos, int depth, int alpha, int beta, char maximizingPlayer, char originalPlayer) {
    if (depth == 0 || !canMove(pos, WHITE) && !canMove(pos, BLACK)) {
        return evaluatePosition(pos, originalPlayer,10,5,3);
    }

    if (maximizingPlayer == originalPlayer) {
        int maxEval = -100000;
        for (int i = 0; i < ARRAY_BOARD_SIZE; i++) {
            for (int j = 0; j < ARRAY_BOARD_SIZE; j++) {
                if (isLegal(pos, i, j, maximizingPlayer)) {
                    Position newPos = *pos; Move move = {{i, j}, maximizingPlayer};
                    doMove(&newPos, &move);
                    int eval = minimax(&newPos, depth - 1, alpha, beta, getOtherSide(maximizingPlayer), originalPla
                    maxEval = (eval > maxEval) ? eval : maxEval;
                    alpha = (alpha > eval) ? alpha : eval;
                    if (beta <= alpha && prune) { break; }
                }
            }
        }
        return maxEval;
    } else {
        int minEval = 100000;
        for (int i = 0; i < ARRAY_BOARD_SIZE; i++) {
            for (int j = 0; j < ARRAY_BOARD_SIZE; j++) {
                if (isLegal(pos, i, j, maximizingPlayer)) {
                    Position newPos = *pos; Move move = {{i, j}, maximizingPlayer};
                    doMove(&newPos, &move);
                    int eval = minimax(&newPos, depth - 1, alpha, beta, getOtherSide(maximizingPlayer), originalPla
                    minEval = (eval < minEval) ? eval : minEval;
                    beta = (beta < eval) ? beta : eval;
                    if (beta <= alpha && prune) { break; }
                }
            }
        }
        return minEval;
    }
}
```

Σε maximizing παίκτη ελέγχω αν β είναι μικρότερο από α και κάνω prune με βάση αυτό. Για minimizing παίκτη το αντίστροφο.

Ο αλγόριθμος ελέγχθηκε τρέχοντας τον αλγόριθμο με και χωρίς pruning μεταξύ τους για 2 παιχνίδια, μια με pruning στην άσπρη πλευρά και μια με pruning στην μαύρη. Και τα 2 παιχνίδια είχαν την ίδια τελική κατάσταση το οποίο γίνεται μόνο αν είναι ίδιοι, αφού στο ένα παιχνίδι παίζει ο παίχτης που κάνει prune πρώτος και στο αλλό αυτός που δεν κάνει.

Παρακάτω φαίνονται οι 2 τελικές καταστάσεις

```
/*
 *        B B B B W W W W         *        B B B B W W W W         *
 *       B B B B B B B B B        *       B B B B B B B B B        *
 *      B W W B W W W W B B       *      B W W B W W W W B B       *
 *     B W W W W W W W B W B      *     B W W W W W W W B W B      *
 *    B W B B B W W W B B W B     *    B W B B B W W W B B W B     *
 *   B B B B W B W B B W B W B    *   B B B B W B W B B W B W B    *
 *  B B W B B B W W B W W B W B   *  B B W B B B W W B W W B W B   *
 * W W W W W W W W B W W W W W B * W W W W W W W W B W W W W W B *
 *  B B B W W W W B W W W W W B  *  B B B W W W W B W W W W W B  *
 *   B W B W B B B W B B B B B   *   B W B W B B B W B B B B B   *
 *    B W W B B B B W B B B B    *    B W W B B B B W B B B B    *
 *     B W B B W B B W B B W     *     B W B B W B B W B B W     *
 *      B B B B B B B B B W      *      B B B B B B B B B W      *
 *       B B B B B B B B W       *       B B B B B B B B W       *
 */
```