Memoria de la Practica de Parchis

Jesús Losada Arauzo

31 de mayo de 2024

1. Analisis del problema

En esta práctica se debe hacer una variante del parchis. Se puede elegir los dados y el 3 no existe. Los dados se restauran una vez que se han gastado todos. Además, hay un dado especial con varios movimientos especiales.

El objetivo es ganarle a los 4 niveles (ninjas), implementando una Poda Alfa-Beta o Minimax, y una heurística eficaz.

2. Descripción del problema

2.1. Poda Alfa-Beta

Aquí se muestra la Poda Alfa-Beta implementada:

```
double AIPlayer::Poda_AlfaBeta(
    const Parchis &actual, int jugador,
    int profundidad, int profundidad_max,
    color &c_piece, int &id_piece,
    int &dice, double alpha,
    double beta,
    double (*heuristic)(const Parchis &, int)
) const {
    if (profundidad == profundidad_max || actual.gameOver())
        return heuristic(actual,jugador);

    bool Max_verstappen = actual.getCurrentPlayerId() == jugador;
    double valor;
    ParchisBros hijos = actual.getChildren();

    for (auto it = hijos.begin(); it != hijos.end(); ++it) {
        valor = Poda_AlfaBeta(*it, jugador, profundidad + 1, profundidad_max, c_piece, id_piece, dice,
             alpha, beta, heuristic);
        if (Max_verstappen) {
            if (alpha < valor) {
                alpha = valor;
                if (profundidad == 0) {
                    c_piece = it.getMovedColor();
                    id_piece = it.getMovedPieceId();
                    dice = it.getMovedDiceValue();
                }
            }
            if (alpha >= beta) return beta;
        } else {
            if (beta > valor) beta = valor;
            if (beta <= alpha) return alpha;
        }
    }
```

```
    return Max_verstappen ? alpha : beta;
}
```

3. Heurística

La heurística utilizada tiene las siguientes ponderaciones:

```
double AIPlayer::MiValoracion1(const Parchis &estado, int jugador) {
    int ganador = estado.getWinner();
    int oponente = (jugador + 1) % 2;
    const int CASILLASRECORRER = 68 + 7;

    if (ganador == jugador) return gana;
    if (ganador == oponente) return pierde;

    vector<color> my_colors = estado.getPlayerColors(jugador);
    vector<color> op_colors = estado.getPlayerColors(oponente);

    double puntuacion_jugador = 0.0;
    color max_goal_color;
    int max_pieces_at_goal = -1;

    for (int i = 0; i < my_colors.size(); i++) {
        color c = my_colors[i];
        int pieces_at_goal = estado.piecesAtGoal(c);
        if (pieces_at_goal > max_pieces_at_goal) {
            max_pieces_at_goal = pieces_at_goal;
            max_goal_color = c;
        }
    }

    for (int i = 0; i < my_colors.size(); i++) {
        color c = my_colors[i];
        puntuacion_jugador -= estado.piecesAtHome(c) * 5;
        int pieces_at_goal = estado.piecesAtGoal(c);
        puntuacion_jugador += pieces_at_goal * 100;
        if (pieces_at_goal == 2) puntuacion_jugador += 200;

        for (int j = 0; j < num_pieces; j++) {
            int distance = estado.distanceToGoal(c, j);
            if (distance > 0) {
                double factor = (c == max_goal_color) ? 0.2 : 0.1;
                puntuacion_jugador += (CASILLASRECORRER - distance) * factor;
            }
        }
    }

    if (estado.getCurrentPlayerId() == jugador) {
        if (estado.isEatingMove()) {
            pair<color, int> Comidas = estado.eatenPiece();
            puntuacion_jugador += (Comidas.first == my_colors[0] || Comidas.first == my_colors[1]) ? 10
                 : 50;
        }
        if (estado.isGoalMove()) puntuacion_jugador += 20;
        if (estado.goalBounce()) puntuacion_jugador -= 10;
    }

    double puntuacion_oponente = 0.0;
    color max_goal_color_op;
    int max_pieces_at_goal_op = -1;

    for (int i = 0; i < op_colors.size(); i++) {
        color c = op_colors[i];
        int pieces_at_goal = estado.piecesAtGoal(c);
```

```
        if (pieces_at_goal > max_pieces_at_goal_op) {
            max_pieces_at_goal_op = pieces_at_goal;
            max_goal_color_op = c;
        }
    }

    for (int i = 0; i < op_colors.size(); i++) {
        color c = op_colors[i];
        puntuacion_oponente -= estado.piecesAtHome(c) * 5;
        int pieces_at_goal = estado.piecesAtGoal(c);
        puntuacion_oponente += pieces_at_goal * 100;
        if (pieces_at_goal == 2) puntuacion_oponente += 200;

        for (int j = 0; j < num_pieces; j++) {
            int distance = estado.distanceToGoal(c, j);
            if (distance > 0) {
                double factor = (c == max_goal_color_op) ? 0.2 : 0.1;
                puntuacion_oponente += (CASILLASRECORRER - distance) * factor;
            }
        }
    }

    if (estado.getCurrentPlayerId() == oponente) {
        if (estado.isEatingMove()) {
            pair<color, int> Comidas = estado.eatenPiece();
            puntuacion_oponente += (Comidas.first == op_colors[0] || Comidas.first == op_colors[1]) ?
                10 : 50;
        }
        if (estado.isGoalMove()) puntuacion_oponente += 20;
        if (estado.goalBounce()) puntuacion_oponente -= 10;
    }
    return puntuacion_jugador - puntuacion_oponente;
}
```

4. Valoración de las piezas

Para el jugador:

- **Piezas en Casa**: Penalización de 5 puntos por cada pieza.
- **Piezas en la Meta**: 100 puntos por pieza, con un bono de 200 puntos si hay dos.
- **Distancia al Objetivo**: Bonificación basada en la cercanía al objetivo, con factores de ponderación de 0.2 y 0.1.
- **Movimientos Especiales**: Bonos y penalizaciones por capturas, movimientos a la meta y rebotes.

Para el oponente: Las valoraciones son idénticas.

5. Ventajas y desventajas

Las ventajas de la heurística son su equilibrio y la capacidad de hacer buenas jugadas, venciendo a los ninjas 0, 1 y 2. Sin embargo, no logra vencer al ninja 3 y a veces falla en capturar o evitar ser capturado.

6. Menciones a las otras heurísticas

Las otras heurísticas son evoluciones de las primeras. La segunda intentó un enfoque diferente y la tercera es una remodelación de la primera para intentar vencer al ninja 3.