UNIVERSIDAD
DE GRANADA

GRADO EN INGENIERÍA INFORMÁTICA

Práctica 3: Búsqueda con Adversario
El Parchis

Autor:
Yeray López Ramírez

Curso:2ºC
Asignatura: Inteligencia Artificial
Fecha: 5 de Junio de 2022

Índice

1. Introducción

El objetivo de esta práctica es programar en C++ un jugador de parchís dotado de una cierta inteligencia artificial. Existen varias modalidades a la hora de jugar al parchís, en nuestro caso es una modificación de un parchís con 4 colores y dos jugadores, de manera que cada jugador controla una pareja de colores: amarillo/rojo para el jugador 1 y azul/verde para el jugador 2.

Se trata de un problema de búsqueda con un adversario. Dado que el parchís de base es un juego NO determinista, no existen reglas exactas que nos lleven de forma segura a un resultado óptimo. De hecho, en los jugadores humanos podemos observar que cada uno aplica unas determinadas reglas en función de su estilo de juego. Las reglas elegidas son lo que llamamos Heurísticas.

Nuestro objetivo en esta practica es diseñar un algoritmo de decisión: Minimax o Poda Alfa-Beta que generarán nuestro arbol de decisión del juego. También tenemos que crear una heuristica efectiva que sea capaz de vencer a tres agentes (Ninjas) creados por los propios profesores del departamento.

2. Desarrollo

2.1. Estrategia MiniMax

La estrategia mínimax consiste en elegir el mejor movimiento para el jugador (MAX) suponiendo que el adversario (MIN) escogerá el mejor para sí mismo.

El algoritmo general consta de los siguientes pasos:

1. Generar el arbol de juego, alternando los movimientos de MAX y MIN, asignando el valor apropiado.
2. Calcular la función de utilidad de cada nodo final, recorriendo el arbol recursivamente hasta volver al nodo inicial.
3. Elegir como jugada final aquel movimiento que mayor puntuación otorge según la función de utilidad.

La implementación del mismo para nuestro problema sería:

1. Generamos el arbol de juego a través de la función generateNextMove(), alternando entre MAX y MIN con un bool MAX que se calcula con la función actual.getCurrentPlayerId(). Si es igual a jugador, significa que nos toca a nosotros en esa iteración y sino le toca al oponente.
2. Calculamos la puntuación en los nodos terminales a través de la heurística. Un nodo terminal es una profundidad de iteración igual a la profundidad_max que en nuestro caso es 4 o la partida en el estado actual ha acabado que se calcula con la función estado.gameOver();
3. Se elige como jugada final los valores de id_piece y dice que más puntuación otorge según la heurística.

Mi implementación del mismo en código C++:

```
double AIPlayer::MiniMax(const Parchis &actual, int jugador, int profundidad, int profundidad_max, color &c_piece, int &id_piece,
        int &dice, double (*heuristic)(const Parchis &, int)) const
{
    bool MAX = (actual.getCurrentPlayerId() == jugador);
    int last_id_piece = -1, last_dice = -1;
    double valor, mejor = menosinf;

    //Si es hoja, devuelve valor
    if(profundidad == profundidad_max or actual.gameOver()){
        return heuristic(actual, jugador);
    }

    if(MAX)
        valor = menosinf;
    else
        valor = masinf;

    //Para cada hijo del tablero
    Parchis hijo = actual.generateNextMove(c_piece, last_id_piece, last_dice);
    while(!(hijo == actual)){
        if(MAX)
            valor = max(valor, MiniMax(hijo, jugador, profundidad+1, profundidad_max, c_piece, id_piece, dice, heuristic));
        else
            valor = min(valor, MiniMax(hijo, jugador, profundidad+1, profundidad_max, c_piece, id_piece, dice, heuristic));

        //cout << " Valor:" << valor << " Mejor:" << mejor << endl;
        if(profundidad == 0 and valor > mejor){
            mejor = valor;
            id_piece = last_id_piece;
            dice = last_dice;
        }
        hijo = actual.generateNextMove(c_piece, last_id_piece, last_dice);
    }
    return valor;
}
```

2.2. Estrategia Poda Alfa-Beta

Sobre los árboles de juego se puede aplicar un tipo propio de poda, conocida como la Poda alfa-beta. Es una modificación del algoritmo MiniMax donde se establecen umbrales en los nodos para descartar aquellos nodos que no los cumplan. Al adaptar la forma general para nuestro caso concreto, nos quedan los pasos:

1. Si es un nodo terminal, devuelve el valor de la heurística. Para el algoritmo alfa-beta, la profundidad máxima del nodo terminal es 6 o menor si es gameOver(). En otro caso, generar el primer hijo. Si el hijo es un nodo MAX, ir al paso 2. Si es un nodo MIN ir al paso 5.
2. Hacer alfa = max(alfa, Poda_AlfaBeta(hijo, jugador, profundidad, profundidad_max, ...))
3. if(alfa >= beta) return beta; else {continuar}.
4. if(hijo==actual) return alfa; else {hijo = generatextMove()}.
5. Hacer beta = min(beta, Poda_AlfaBeta(hijo, jugador, profundidad, profundidad_max, ...))
6. if(beta <= alfa) return alfa; else {continuar}.
7. if(hijo==actual) return beta; else {hijo = generatextMove()}.

La implementación del mismo en C++:

```
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
{
    bool MAX = (actual.getCurrentPlayerId() == jugador);
    int last_id_piece = -1, last_dice = -1;
    double valor = menosinf, mejor = menosinf;

    if(profundidad == profundidad_max or actual.gameOver()){
        return heuristic(actual, jugador);
    }

    //Para cada hijo del tablero
    Parchis hijo = actual.generateNextMoveDescending(c_piece, last_id_piece, last_dice);
    while(!(hijo == actual)){
        valor = Poda_AlfaBeta(hijo, jugador, profundidad+1, profundidad_max, c_piece, id_piece, dice, alpha, beta, heuristic);
        //Paso 2
        if(MAX){
            alpha = max(alpha, valor);
            //Paso 3
            if(alpha >= beta)
                return beta;
        }
        else{ //Paso 5
            beta = min(beta, valor);
            //Paso 6
            if(beta <= alpha)
                return alpha;
        }

        //Guarda el mejor valor
        if(profundidad == 0){
            if(alpha > mejor){
                mejor = alpha;
                id_piece = last_id_piece;
                dice = last_dice;
            }

        }
        hijo = actual.generateNextMoveDescending(c_piece, last_id_piece, last_dice);
    }

    //Paso 4 y 7
    if(MAX)
        return alpha;
    else
        return beta;
}
```

2.3. Heurísticas

La heurística es la parte más importante ya que es la que determina el comportamiento del agente que vamos a crear. En mi caso, se dispone de 3 heurísticas muy similares en código pero de comportamientos radicalmente distintos. Las llamaremos ganaJ1, ganaJ2 y valoracionDefinitiva.

2.3.1. Comportamiento general

A pesar de ser 3 heurísticas muy distintas, su código es extrañamente similar:

1. Si gana el jugador, se devuelve un valor $+\infty$. Si gana el oponente, se devuelve un valor $-\infty$. En otro caso, se establecen las puntuaciones según estados del tablero.
2. Se recorren todas las fichas del jugador y del oponente con dos bucles for respectivamente: for (i to color.size) for(j to num_pieces){ *puntuaciones* }
3. Se valora positivamente la posición de las fichas en el tablero, cuando más alto mejor. Aquí

difieren las heurísticas ya que la J1 valora positivamente cualquier posicion del tablero pero las otras 2 solo puntuan si se encuentra en la cola de meta.

4. Si está en una casilla segura, se valora positivamente. +1 para las J1 y J2, +10 para la definitiva.
5. Si tiene casillas en casa implica que han sido comidas por lo que se valora muy negativamente. Se resta -100 para cada ficha en home dando como resultado que intenta evitar ser comido y saca fichas cuando puede.
6. Cualquier casilla que llegue a la meta se puntua de forma muy positiva. Se suma +100 por cada ficha en meta. Al tener la misma puntuación que comer, sigue prefiriendo comer antes que meterla ya que se puntua la posición del tablero positivamente.
7. La puntuación del oponente es exactamente igual en las 3: Se le valora positivamente al oponente según la posición de sus fichas en el tablero. Además se le penaliza al oponente que sus fichas estén en casa dando como resultado que nuestro agente esté más ansioso por comerselo y bloquear sus casillas de casa (efectivamente, si 2 fichas ocupan la casilla de casa el jugador no puede sacar ficha).

2.3.2. Heuristicas: ganaJ1

Esta es la primera heurística y también la que gana de forma aplastante todos los ninjas como jugador 1. Es pésima como jugador 2, no le gana ni al ninja 1.

Su funcionamiento radica en maximizar el desplazamiento en el tablero, cuantas más fichas y mas cerca de la meta haya mejor. Esto le ayuda a alcanzar la meta rapidamente pero deja fichas rezagadas muy golosas para el oponente. Al ser jugador 1 no supone ningun problema ya que se escapa antes de ser comido, no ocurre lo mismo como jugador 2.

2.3.3. Heuristicas: ganaJ2

En esta heurística sin embargo les gana de forma aplastante a todos los ninjas como jugador 2. Como jugador 1 dispone de un comportamiento extraño, no vence ni al ninja 1 o 2 pero gana satisfactoriamente contra el ninja 3.

Su funcionamiento radica en valorar muy positivamente la cola de meta pero no el resto del tablero. Esto evita que las fichas queden rezagadas intentando maximizar el tablero, funcionando muy bien como segundo jugador pero no tiende a avanzar por lo que como jugador 1 no es muy buena.

2.3.4. Heuristicas: valoracionDefinitiva

Como bien indica su nombre es la que vence de forma satisfactoria a todos los agentes propuestos. Es una heuristica ganaJ2 vitaminada. No vence a los agentes tan sobradamente como las otras 2 por separado.

Es la más distinta de las 3 ya que se añade una funcionalidad extra: detecta enemigos cercanos a cada ficha. Es decir, calcula cuantas fichas enemigas tiene delante y detrás. Puntúa positivamente tener fichas enemigas delante en un rango de 7 casillas, +9. Puntúa negativamente tener fichas

enemigas detrás en un rango de 7 casillas, -9. Esta funcionalidad es muy efectiva contra los ninjas 1 y 2 aunque no tanto con el ninja 3 (penaliza tardando más en ganarle).

3. Conclusiones

Podemos sacar en claro varias cosas:

- En cuanto a comportamiento, a pesar de que las heurísticas son extremadamente simples los agentes desarrollan jugadas complejas. Entre ellas:
    - A veces un color obtiene el rol de "support" en el que simplemente se dedica a bloquear a las fichas enemigas y ser cebo para el otro color. Mientras que el otro color obtiene el rol de "ganador", comiendose toda ficha que ve y llegando rapidamente a la meta.
    - Suele tender emboscadas en la cual un color bloquea con barrera y otro come fichas.
    - En determinadas ocasiones las fichas se quedan en las casas de los colores del oponente, impidiendole sacar fichas.
    - También tiene en cuenta los dados del oponente, colocandose en "puntos ciegos" donde el oponente no dispone del número necesario para comer. Destaca el 6, ya que si todas tus fichas están fuera te mueves 7 casillas por lo que toda ficha a distancia 6 necesita de un 5 y un 1.
    - Es común ver como se come fichas propias con tal de comerse la de otro jugador y llegar antes a la meta. También se come sus propias fichas para entrar directo a la meta o a la cola de meta.
- En cuanto al tipo de heuristicas: tanto ganaJ1 como ganaJ2 son heurísticas claramente ofensivas ya que aunque hagan barreras de vez en cuando, intentan moverse rápidamente por el tablero. Sin embargo la heurística definitiva es más equilibrada: tiende a quedarse en zonas seguras e ir avanzando progresivamente pero no desperdicia ninguna oportunidad de comer si le conviene.
- Respecto al resto de compañeros, aprovechando la posibilidad de competir entre nosotros he podido comparar mis heurísticas con el resto. En general, se ha optado por heurísticas muy defensivas usando anywall e isWall. Al parecer, este tipo de heurísticas son más efectivas que las ofensivas pues generalmente vencen mi agente (reñidas, eso sí).
- El minimax también es capaz de ganar a los 3 agentes como j1 y j2 con la heuristica definitiva aunque con mayor dificultad que el alfa-beta.