Práctica 3: Reversi

Introducción

En esta práctica desarrolaremos diferentes estrategias para la eleccion de movimientos en el juego del Reversi. Para ello implementaremos diferentes funciones de evaluación que nos indicarán que movimiento habrá que realizar en cada momento.

Desarrollo

Al comienzo de la práctica, lo primero que hicimos fue probar las funciones que nos daban: mobility, count-difference y random-strategy. Tras ver como funcionaban, intentamos hacer una versión algo mejorada, haciendo combinaciones de ellas, en primer lugar eligiendo aleatoriamente entre ellas, y después con pesos ponderados.

Esto supuso una mejora, pero no fue gran cosa, así que decidimos pensar estrategias algo más elaboradas. Para ello, también investigamos en internet distintas estrategias para el juego. La primera que vimos que podía ser interesante, era una que asignaba a cada casilla unos pesos y que viendo la explicación parecía bastante convincente e intentamos probarla. Así nació la función *weight*.

La función *weight* lo que hacía era sumar el peso de cada casilla del jugador, y le restaba los del oponente, de forma que la jugada que dirá las casillas más "privilegiadas" sería la ganadora. Esta estrategia funcionaba bastante mejor, pero sabíamos que aún podría mejorarse.

Tras algo más de investigación vimos también que mientras que tener las casillas privilegiadas en las últimas fases de la partida era lo mejor, al principio no era tan bueno y era más prioritario tener la mayor movilidad posible. Por ello, hicimos una fórmula que daba más prioridad a la estrategia movilidad al comienzo de la partida y menos yendo hacia el final, que es la siguiente:

$$\frac{status}{64} \times weight + \frac{64 - status}{64} \times mobility$$

Siendo *status* el número de fichas que hay en el tablero, lo uqe representa el estado de la partida. Esto de nuevo supuso una pequeña mejora, pero seguimos buscando.

Lo siguiente que encontramos fue que daba cierta ventaja que un jugador pusiera la última ficha, lo que llamaban paridad. Para implementar esto creamos la función *parity-check*, que lo que hace es comprobar si, si ningún jugador se salta turno, quién pondrá la última ficha. Esto lo hará devolviendo True si *player* termina o NIL si lo hace su oponente. Teniendo esta función, la añadimos a la función

haciendo que antes de calcular nada, controlara si terminaría él, y de no ser así, comprobar si el adversario no tiene movimientos (por lo que tiene que pasar). Si estas condiciones se cumplen, se elegirá ese movimiento sin importar el resultado de la fórmula anterior. Ese fue el nacimiento de nuestra primera función: *weight-mobility-parity*. Para la siguiente mejor sacamos la mayor parte de la información de un documento en concreto de la competencia. Partir de esta información decidimos aplicar las últimas dos funciones *weight-mobility-corner* y *weight-mobility-corner-pro*.

A continuación, describimos las funciones finales con mayor detalle.

Función 1: (weight-mobility-parity)

Esta función, lo primero que hacemos es realizar algunos cálculos previos. En primer lugar, obtenemos el estado de la partida a partir de la función *game-status*, que devuelve el número de fichas que hay colocadas en el tablero. Después, con la función *mobility* calculamos la movilidad del jugador, con la *weight-fn* el valor de la jugada según los pesos descritos anetriormente y con *legal-moves* pasándole el jugador oponente el número de movimientos que puede hacer.

Teniendo esto, en primer lugar se comprueba si se reúnen las condiciones anteriormente descritas sobre la paridad para realizar esa jugada o no, devolviendo como heurística 10000, que es lo mismo que decir que se haga ese movimiento ya que en ningún caso puede haber otra más alta. Si no se reúnen las condiciones, entonces se devolverá el resultado de la fórmula anterior tras sustituir por los valores previamente calculados.

Función 2: (weight-mobility-corner)

Para esta función, hemos seguido la misma estructura que en la anterior añadiendo una mejora. La mejora consiste en que antes de realizar la suma de los pesos asignados a cada casilla, comprueba el estado de las esquinas. Si están las cuatro esquinas ocupadas, le asignará unos pesos distintos que a cada casilla.

Hemos tomado esta decisión ya que al tener las esquinas ocupadas ya no tenemos que evitar las casillas contiguas a estas, es más, deberíamos darles cierta prioridad para controlar una zona si es nuestra la esquina, o evitar el control del adversario de esa zona si no es nuestra. De ahí que tengan pesos más altos que el resto de casillas cuando están ocupadas.

Función 3: (weight-mobility-corner-pro)

Para esta función, al observar cierta mejoría en la función dos, hemos optado por mejorar esta. Para ello además de tener en cuenta el caso de que las cuatro esquinas están ocupadas, hemos decidido tener en cuenta cada caso de esquina ocupada. Por lo que hemos tenido en cuenta todos los casos de una, dos, o tres esquinas ocupadas y le hemos asignado unos determinados pesos a cada caso. Solo se cambiarán los pesos de la esquina ocupada.

Conclusiones

Aunque las funciones no son perfectas, y siendo sinceros tampoco muy eficientes, estamos bastante satisfechos con los resultados que hemos obtenido. Además, creemos que esta práctica ha sido muy interesante y sobre todo útil, ya que es de lo más parecido que hemos hecho a un problema real y el modelo de práctica. El hecho de investigar por nuestra cuenta, implementar lo que nos ha parecido interesante y probar el resultado nos ha parecido mucho más didáctico y llevadero que la típica práctica que consiste en darnos unos ejercicios y una guía de cómo hacerlos y limitarnos a escribir código sobre eso directamente.