Proyecto de Inteligencia Artificial

David Sánchez Iglesias

2023

1. Estrategia

El jugador de Hex que se propone en este proyecto usa el algoritmo Minimax para simular el juego y decidir su siguiente movimiento. Para reducir el tiempo de ejecución, se implementó la poda alfa-beta así como dos heurísticas destinadas a evaluar las diferentes posiciones durante la ejecución de Minimax.

2. Implementación

El código se encuentra en la clase *MiniMaxPlayer* del archivo *Player.py*. El método *play*, que es el encargado de devolver una jugada, es desde donde se ejecuta el algoritmo Minimax, y el método de este es *minimax*, justo debajo de *play*. Este método presenta dos optimizaciones importantes. La primera consiste en la poda alfa-beta:

```
# ...
if maximizing_player:
        max_eval = -float('inf')

        for move in valid_moves:
            row, col = move
            game_copy = board.clone()
            game_copy.place_piece(row, col, player)
            # game_copy.check_connection(player)

            evaluation, _ = self.minimax(game_copy, depth-1, alpha, beta, (3-
            if evaluation > max_eval:
                max_eval = evaluation
                best_move = move

            #Yo maximizo, mi padre minimiza
            #Si mi alpha es mayor que el beta de mi padre, dejo de buscar
            alpha = max(alpha, evaluation)
            if beta <= alpha:
                break
```

```
            return max_eval, best_move
        else:
        #... rest of the code
```

En el fragmento presentado, se muestra lo que hace el método cuando el jugador al que le toca es el que busca maximizar el valor de sus jugadas. La poda se encuentra al final del fragmento, cuando se actualiza el *alpha*. El jugador que busca maximizar intentará aumentar el alpha, y el que minimiza buscará disminuir la beta. La *evaluacion* devuelve el valor de un hijo del "nodo" en el que se encuentra el algoritmo, mientras que el *alpha* y el *beta* vienen del padre. El nodo actual actualiza su *alpha*/*beta* cada vez que se evalua un hijo, asegurando que se quede con el valor que más le conviene. La poda alfa-beta consiste en pararse en un nodo, tomar el valor del padre y compararlo con el valor que devuelven los nietos del padre. Si yo soy un nodo que maximiza, entonces mi padre minimiza. Si mi *alpha* es mayor que el *beta* de mi padre, entonces mi padre, que busca minimizar, ya encontró un valor menor. Yo busco maximizar, por lo que, si mi *alpha* vuelve a cambiar, será solo para aumentar, nunca será relevante para mi padre, pues él ya tiene un valor menor. Esta comparación se realiza en el fragmento:

```
alpha = max(alpha, evaluation)
    if beta <= alpha:
        break
```

La misma lógica se aplica para el jugador que minimiza:

```
beta = min(beta, evaluation)
    if beta <= alpha:
        break
```

La segunda optimización consiste en aplicar la heurística a las jugadas que un jugador puede realizar antes de llamar recursivamente. De este modo se intenta que los movimientos más prometedores se prueben primero, maximizando el efecto de la poda alfa-beta y evitando llamados recursivos innecesarios.

2.1. Heurística 1: Distancia de Dijkstra

Esta heurística consiste en una variación de la distancia de Dijkstra. En Hex el objetivo de cada jugador es conectar los dos bordes del tablero que le tocan, de ahí que la distancia que falta por cubrir para llegar de un lado al otro puede ser un buen indicador de qué tan cerca está el jugador de la victoria. El método propuesto inicia en una casilla ya ocupada por el jugador en cuestión, y desde ahí se expande al resto del tablero, contando, con cada casilla, la distancia recorrida. Pero, para conocer qué tan cerca está un jugador de crear un camino entre los dos bordes, se deben tener en cuenta también las fichas del propio jugador que se encuentren separadas de los bordes del tablero. Para eso, se le puso peso a las fichas: 0, para las del propio jugador; -1 para las del contrario. Mientras se

explora el tablero, si aparece una ficha propia del jugador, el contador no modifica el valor que se lleva hasta el momento, sino que acepta esas casillas como parte de la frontera expandida para que, en la siguiente iteración, se puedan expandir las casillas adyacentes a estas. Como resultado, se obtiene la cantidad de casillas que faltan por cubrir para formar el camino más corto posible de un lado al otro, en dependencia del jugador. Un dato importante es que este algoritmo **siempre** empezará a buscar desde una casilla del borde (columna o fila 0) que tenga una ficha del propio jugador. En *minimax* se calcula esta distancia para toda casilla en la que puede jugar el jugador, antes de llamar recursivamente, lo que significa que primero se juega en una copia del tablero y luego se calcula la heurística. Por esto, en un inicio, las casillas de los bordes tendrán mayor valor que las centrales, propiciando que casi siempre el jugador ponga una ficha en estas casillas al hacer su primera jugada. Esto se dejó así a propósito para facilitar el cálculo de la distancia de Dijkstra y darlo en un menor tiempo posible.

2.2. Heurística 2: Cantidad de puentes

En hex, cuando dos fichas del mismo color se encuentran a una casilla de distancia, pero existen dos caminos entre ellas, con esa misma distancia, sin fichas enemigas de pormedio, se dice que hay un puente. En otras palabras, cuando hay dos casillas vacías del tablero, adyacentes ambas a dos casillas del mismo color, o cuando hay dos caminos de distancia 1 entre dos fichas del mismo color.

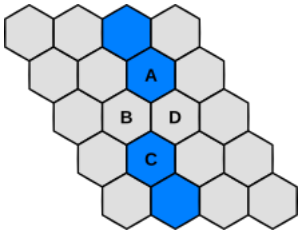

Figura 1: Ejemplo de puente

En la figura, se dice que entre las fichas A y C, hay un puente, pues las casillas B y D estan vacías y son adyacentes a ambas: A y C. El método que se encarga de calcular la cantidad de puentes, *find_bridges*, consiste en recorrer el tablero, tomar todas las combinaciones, sin repetición, de casillas ocupadas por fichas del jugador; verificar que la diferencia entre sus componentes sea menor o igual a 2 para constatar la cercanía entre ambas; extraer todas las casillas adyacentes a cada una y comprobar si la intersección de ambos conjuntos tiene tamaño 2. En caso afirmativo, se aumenta un contador.

2.3. Manejo del tiempo

Para asegurar que este programa no se demore demasiado calculando las jugadas, se usaron dos variables que ponen límites al método *minimax*, pues es este el método que realiza la simulación del juego.

- *depth*: Es la profundidad máxima que se va a explorar el algoritmo. Se pasa como parámetro al método *play*, y limita la profundidad hasta la cual se va a explorar.
- *TOP_MOVES*: Esta variable se inicializa en el constructor de la clase *MiniMaxPlayer*, y se usa para limitar la cantidad de posibles jugadas que, para cada jugador, *minimax* explora. Ya que las jugadas se ordenan antes de ser exploradas por el valor que devuelven las heurísticas, el programa asume que las *TOP_MOVES* primeras después de ordenar, son las de mayor importancia, las más prometedoras, y desestima el resto, ahorrando muchos llamados recursivos dentro de *minimax*.

Ambas variables traen valores por defecto probados con éxito en tableros de 11x11 e inferiores. En tableros con tamaños superiores el tiempo de ejecución puede ser superior a los 10 segundos en algunas posiciones. Se recomienda mantener *TOP_MOVES* con un valor entre 7 y 10, o hasta 15 si se no se requiere de un tiempo límite muy corto.
Por su parte, *depth* se recomienda mantenerla entre 3 y 5. En el método *heuristic* es donde se hace uso de ambas heurísticas y se establece un valor definitivo para cada jugada. Los valores se ajustan para dar más "importancia" a una heurística o a otra. Dichos valores fueron ajustados manualmente en función del desempeño del programa.