Práctica 2

Búsqueda con adversarios

Junco de las Heras Valenzuela
y
Marta Vaquerizo Núñez

Índice

1. Documentación para minimax con poda alfa-beta.

a. Detalles de implementación.

i. ¿Qué pruebas se han diseñado y aplicado para determinar si la implementación es correcta?

Para comprobar que la implementación es correcta, en el fichero `demo_reversi.py`, se han creado dos jugadores `player_minimax5` y `player_minimax6` con la estrategia de minimax poda $\alpha - \beta$. Se ha comprobado que tanto con el minimax como con el minimax con poda se han tomado los mismos movimientos (puesto que tenían heurísticas deterministas deberían dar el mismo mejor movimiento). Después se ha comprobado que se ha reducido el tiempo de ejecución usando la librería `timeit` jugando una partida con dos jugadores minimax y luego jugando entre dos jugadores del tipo `player_minimax5` y `player_minimax6`. En en siguiente apartado se muestra una tabla con los tiempos.

ii. Diseño: Estructuras de datos utilizadas, descomposición funcional, etc.

En el fichero `strategy.py`, la clase `MinimaxAlphaBetaStrategy` está formada por tres funciones, a parte de la de inicialización:

- **next_move**: esta función determina el siguiente movimiento del jugador, se llama siendo un nodo MAX.
- **_min_value**: esta función calcula el β, ya que se llama siendo un nodo MIN.
- **_max_value**: esta función calcula el α, ya que se llama siendo un nodo MAX.

En la función `next_move`, se inicializan los valores de $[\alpha, \beta]$ a $[-\infty, \infty]$, y después se recorren todos los sucesores, que son nodos MIN, para obtener sus $\beta's$, y así obtener el máximo de ellas, dándonos el siguiente movimiento. Para obtener la β en cada nodo MIN, se llama a la función `min_value`, que pasa por todos sus sucesores (nodos MAX), y obtiene el mínimo de los valores minimax devueltos por la función `max_value` en cada uno de ellos, y así recursivamente hasta llegar a los nodos hoja, en los que directamente se devuelve el valor de la función utilidad como valor minimax.

El caso base de la recursión es cuando se llega a profundidad 0, esto se debe a que en cada llamada recursiva, se decrementa en uno la profundidad, por lo que el algoritmo finaliza, completando la recursión cuando llega a 0. Cada función devuelve el valor minimax, así como el α y β de ese nodo, y de esta manera su estado ancestro (padre) pueda actualizar los suyos en función de los del hijo.

III. Implementación.

```
class MinimaxAlphaBetaStrategy(Strategy):
"""Minimax alpha-beta strategy."""

def __init__(
    self,
    heuristic: Heuristic,
    max_depth_minimax: int,
    verbose: int = 0,
) -> None:
    super().__init__(verbose)
    self.heuristic = heuristic
    self.max_depth_minimax = max_depth_minimax

def next_move(
    self,
    state: TwoPlayerGameState,
    gui: bool = False,
) -> TwoPlayerGameState:
    """Compute next state in the game."""

    successors = self.generate_successors(state)
    next_state = None
    minimax_value = -np.inf
    alpha = -np.inf
    beta = np.inf

    for successor in successors:
        if self.verbose > 1:
            print('{}: {}'.format(state.board, minimax_value))

            successor_minimax_value, alpha_ret, beta_ret =
            self._min_value(
                successor,
                self.max_depth_minimax,
                alpha,
                beta
            )
            # If the actual successor gives more value, update
            next_state.
            if successor_minimax_value > minimax_value:
                minimax_value = successor_minimax_value
                alpha = minimax_value
                next_state = successor
```

```
        if self.verbose > 0:
            if self.verbose > 1:
                print('\nGame state before move:\n')
                print(state.board)
                print()
            print('Minimax value = {:.2g}'.format(minimax_value))

        # If the actual node is not final node, there will be next state.
        return next_state

    def _min_value(
        self,
        state: TwoPlayerGameState,
        depth: int,
        alpha: int,
        beta: int,
    ) -> (float, float, float):
        """Min step of the minimax-alpha-beta pruning algorithm."""
        if state.end_of_game or depth == 0:
            minimax_value = self.heuristic.evaluate(state)
            alpha = minimax_value
            beta = minimax_value
        else:
            minimax_value = beta
            successors = self.generate_successors(state)

            for successor in successors:
                if self.verbose > 1:
                    print('{}:{}'.format(state.board,minimax_value))

                successor_minimax_value, alpha_ret, beta_ret =
                self._max_value(
                    successor, depth - 1, alpha, beta
                )
                # Actualize min node.
                if alpha_ret < beta:
                    beta = alpha_ret
                    minimax_value = beta
                # Prune.
                if alpha >= beta:
                    break

        if self.verbose > 1:
            print('{}: {}'.format(state.board, minimax_value))
```

```
    return minimax_value, alpha, beta

def _max_value(
    self,
    state: TwoPlayerGameState,
    depth: int,
    alpha: int,
    beta:  int,
) -> (float, float, float):
    """Max step of the minimax-alpha-beta pruning algorithm."""

    if state.end_of_game or depth == 0:
        minimax_value = self.heuristic.evaluate(state)
        alpha = minimax_value
        beta = minimax_value
    else:
        minimax_value = alpha

        successors = self.generate_successors(state)

        for successor in successors:
            if self.verbose > 1:
                print('{}: {}'.format(state.board,minimax_value))

            successor_minimax_value, alpha_ret, beta_ret =
            self._min_value(
                successor, depth - 1, alpha, beta
            )

            # Actualize max node.
            if beta_ret > alpha:
                alpha = beta_ret
                minimax_value = alpha
            # Prune.
            if alpha >= beta:
                break

if self.verbose > 1:
    print('{}: {}'.format(state.board, minimax_value))

return minimax_value, alpha, beta
```

iv. Otra información relevante.

Al tener que realizar la implementación del algoritmo minimax poda $\alpha - \beta$, hemos podido entender mejor el funcionamiento de las llamadas recursivas del algoritmo, visto en clase de forma teórica.

b. Eficiencia de la poda alfa-beta. Descripción completa del protocolo de evaluación.

i. Tablas con información sobre los tiempos empleados con y sin poda.

A continuación, se muestra la tabla de tiempos de 1 ejecución:

Heurística \ Versión	**Sin poda**	**Con poda**
Proporcionada	1519,234	122,369
Nuestra	2818,446	164,016

Cuadro 1: **Tiempos de ejecución (s)**.

Cabe destacar que en la partida con el minimax y nuestra heurística, se quedó en la mitad porque uno de los jugadores se pasó de tiempo. Así que, si la mitad de la partida tardó 2818,446 segundos, aproximadamente toda la partida hubiese durado unos 4818,446 segundos, como unos 2000 segundos más, y no el doble, ya que cuando quedan pocas posiciones por abarcar del tablero, se acelera la partida.

ii. Medidas de mejora independientes del ordenador.

Para tomar medidas independientes del ordenador, se han tomado 2 medidas con el ordenador de un miembro de la pareja, y otras 2 medidas usando el ordenador del otro miembro, y luego se ha hecho la media aritmética. A continuación se muestra la tabla con los nuevos tiempos:

Heurística \ Versión	**Sin poda**	**Con poda**
Proporcionada	1413,832	108,556
Nuestra	2820,446	152,160

Cuadro 2: **Tiempos de ejecución (s)**.

Cabe destacar que nuestra heurística sin poda sigue dando timeout, de ahí que tenga un tiempo parecido a la tabla de una sola ejecución, mientras que los valores de tiempo de las demás casillas decrementan. Esto se debe a que el ordenador, que no ha tomado los tiempos del apartado anterior, ejecutaba, en media, las heurísticas en menos tiempo que el otro ordenador, y eso ha producido que los tiempos generales hayan decrementado.

III. Un análisis correcto, claro y completo de los resultados.

Primero se va a analizar los tiempos de la heurística proporcionada. Se puede observar que usando la estrategia minimax con poda $\alpha - \beta$ se obtiene un tiempo bastante menor que cuando se usa la de minimax, con una diferencia de 1250 segundo aproximadamente. Esto mismo ocurre si comparamos los tiempos de nuestra heurística, solo que la diferencia de tiempos es incluso mayor.

También se puede observar que los tiempos de nuestra heurística son mayores que los de la heurística proporcionada. Esto se debe a que la heurística proporcionada es más simple que la nuestra. Es importante destacar que en el caso de sin poda, la diferencia de tiempos es muchísimo mayor en comparación con la diferencia de tiempos en el caso de con poda. La razón de este suceso es que en el caso de sin poda se pasa por todos los nodos, y si se realiza la estrategia con una heurística más compleja, que tarda más, el tiempo aumenta bastante.

IV. Otra información relevante.

Hemos podido comprobar que el algoritmo minimax con poda alfa-beta acelera considerablemente el tiempo de ejecución de las heurísticas, aunque la peor complejidad sea la misma del algoritmo sin poda, la complejidad media es mejor, comprobado empíricamente.

2. Documentación del proceso de diseño de la heurística.

a. Descripción de trabajo anterior sobre estrategias para el juego Reversi, incluyendo estrategias en formato APA.

Primero buscamos información sobre cómo jugar al Reversi en pdfs como el siguiente [1]. Y luego, en una aplicación para móviles, jugamos una serie de partidas hasta que fuimos entendiendo y aprendiendo una serie de estrategias, las cuales implementaríamos luego, así como que probaríamos su eficacia en los torneos anticipados.

[1] Engel, K. T. Learning a Reversi Board Evaluator with Minimax.

b. Descripción del proceso de diseño:

i. ¿Cómo se planificó y realizó el proceso de diseño?

Tras investigar las estrategias para el juego, decidimos crear funciones que calculasen un valor a partir de una estrategia, por ejemplo, crear un tablero con pesos en cada posición, que favorece atacar las esquinas, o tener en cuenta la movilidad del adversario, es decir, cuantos movimientos puede realizar. Las heurísticas se calculan con distintos pesos de estas funciones.

Se planificó el diseño de un modo incremental, escribíamos código para nuestras primeras heurísticas, las poníamos a prueba en los torneos anticipados y veíamos cómo se desempeñaban contra los otros adversarios. También competíamos manualmente contra nuestras heurísticas, viendo posibles fallos como que regalase una esquina o una X-square (Casilla en diagonal de la esquina).

ii. ¿Se utilizó un procedimiento sistemático para la evaluación de las heurísticas diseñadas?

Para evaluar las heurísticas, las poníamos a competir entre ellas y veíamos las jugadas que realizaban, si alguna de estas jugadas parecía sospechosa o débil, se cambiaba lo que hacía que eso ocurriese. Otro procedimiento era jugar uno mismo contra la máquina, y observar de nuevo las jugadas de la máquina. A parte de estos prodecimientos que se hacían antes de la entrega de los torneos, una vez que salía el *ranking*, veíamos la posición en la que nuestra heurística había quedado, y en función de la misma, mejorabamos las heurísticas y repetíamos todo el procedimiento anterior.

iii. ¿Utilizaste en el diseño estrategias diseñadas por otros? Si estas están disponibles de manera pública, proporciona referencias en formato APA; en caso contrario, indica el nombre de la persona que proporcionó la información y reconoce dicha contribución como "comunicación privada".

La idea de poner pesos a las casillas del tablero la hemos tomado del pdf [2]. Otro paper que también nos ha ayudado a entender mejor el juego del Reversi, así como sus bases teóricas (con la idea de mejorar nuestro juego para hacer una mejor heurística) es el siguiente [3].

[2]Sannidhanam, V., & Annamalai, M. (2015). An Analysis of Heuristics in Othello.
[3]Lu, K. (2014). The Game Theory of Reversi.

c. Descripción de la heurística final entregada.

Las heurísticas y todo el código relacionado con ellas está en el fichero:

`2301_p2_09_de_las_Heras_Vaquerizo.py`.

Las tres heurísticas que hemos entregado son muy parecidas, solo cambian los pesos que les hemos asignado a unas variables, que dependen de funciones que atacan distintas estrategias, de tal forma que cada una busque un objetivo distinto.

Lo primero que hace la heurística es comprobar si el estado es un estado terminal o no, si lo es, devuelve el valor real de la puntuación, es decir, el número de fichas que tienen tu color menos el número de fichas que tienen el color del rival. Luego se inicializan unas variables, una como un tablero en forma de array, y otra un tablero en forma de diccionario.

Ahora, se obtiene el factor de movilidad, es decir, el número de movimientos válidos que tiene actualmente. Cuantos más movimientos válidos tenga mejor, porque hay más probabilidad de elegir un movimiento que sea muy bueno y saque mucha ventaja posicional. A este valor se le restan los movimientos válidos del oponente, ya que cuantos más movimientos tenga el oponente, menos nos beneficia.

Seguidamente, itera sobre todas las casillas del tablero y le suma el valor asignado en la posiciones en las que se encuentra el jugador, y se le resta el valor asignado en las posiciones en las que se encuentra el opononente.

En ret (de retorno), se guarda una media ponderada del factor valores del tablero y de la movilidad con distintos pesos dependiendo de la heurística. Finalmente, si el jugador juega como un jugador MAX devuelve esa estimación, si juega como MIN devuelve la estimación negada (Porque asumimos que juega MAX, entonces la estrategia MIN quiere hacer la jugada opuesta a MAX).

d. Otra información relevante.

Gracias a los múltiples torneos anticipados, hemos podido comprobar cuán buenas eran nuestras heurísticas con respecto a las de otras parejas, viendo qué tipo de cosas funcionaban y cuales no. Por ejemplo, poner algo de indeterminismo (con un random()) no nos dió buenos resultados, así que en la entrega final hemos suprimido esas llamadas.