

Practica 2 IA-Reversi



**Javier Pérez de Lema Díez y Pablo
Sánchez**

1. Documentación de minimax y la poda alfa-beta



a. Detalles de implementación

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

Hemos utilizado los tests incluidos con la práctica de minimax y la poda alfa-beta como estrategias de decisión en los player1 y player2 en el archivo demo_simple_game_tree.py, observando que obtenemos unos resultados semejantes a los que obtenemos realizando la poda a mano.

Tras ejecutar el código python3 demo_simple_game_tree.py con dos jugadores, uno con player1 = player1_minmax y otro con player2 = player1_minmax_alpha_beta obtenemos este resultado:

Let's play SimpleGameTree!

A

It is the turn of player 'Minimax 1' [Player 1].

```
A: -inf  
B: inf  
E: 4  
B: 4  
F: 3  
B: 3  
A: 3
```

C: *inf*
G: *-inf*
K: 2
G: 2
L: 1
G: 2
C: 2
H: 5
C: 2
A: 3
D: *inf*
I: 5
D: 5
J: *-inf*
M: 4
J: 4
N: 2
J: 4
O: 6
J: 6
P: 1
J: 6
D: 5
A: 5

Game state before move:

A

Minimax value = 5

Player 'Minimax 1' [Player 1] moves AD.

D

It is the turn of player 'Minimax + alpha-beta 1' [Player 2].

D: [-*inf*,*inf*]
I: -5
D: [-5,*inf*]
J: *inf*
M: -4
J: -4
N: -2
J: -4
O: -6
J: -6

Game state before move:

D

Alpha value = -5

Player 'Minimax + alpha-beta 1' [Player 2] moves DI.

/

Game over.

Player Minimax 1 [Player 1]: 5

Player Minimax + alpha-beta 1 [Player 2]: 0

Y si cambiamos los players: con player1 = player1_minmax_alpha_beta
y otro con player2 = player1_minmax obtenemos este resultado:

Let's play SimpleGameTree!

A

It is the turn of player 'Minimax + alpha-beta 1' [Player 1].

A: [-inf,inf]

B: inf

E: 4

B: 4

F: 3

B: 3

A: [3,inf]

C: inf

G: -inf

K: 2

G: 2

L: 1

G: 2

C: 2

A: [3,inf]

D: inf

I: 5

D: 5

J: -inf

M: 4

J: 4

N: 2

J: 4

O: 6

J: 6

D: 5

Game state before move:

A

Alpha value = 5

Player 'Minimax + alpha-beta 1' [Player 1] moves AD.

D

It is the turn of player 'Minimax 1' [Player 2].

D: -inf

I: -5

D: -5

J: inf

M: -4

J: -4

N: -2

J: -4

O: -6

J: -6

P: -1

J: -6

D: -5

Game state before move:

D

Minimax value = -5

Player 'Minimax 1' [Player 2] moves DI.

I

Game over.

Player Minimax + alpha-beta 1 [Player 1]: 5

Player Minimax 1 [Player 2]: 0

ii. Diseño: estructuras de datos seleccionadas, descomposición funcional, etc.

- Nuestro código está dividido en 3 funciones:
next_move(state:twoPlayerGameState) el cual se encarga de llamar a una función de llamada a todos los estados sucesores de este estado principal. Después comienza la

llamada a la función `_minxd()` sobre cada uno de los estados sucesores.

- `_minxd(sucesor, profundidad_maxima, alfa, beta)` se encarga de hacer la evaluación min de los estados.

El algoritmo sigue el pseudocódigo proporcionado como se espera: genera sus sucesores y hace la llamada a `_maxxd()`, y al retornar los resultados alfa-beta los actualiza en el nodo padre. Hemos decidido que ambos algoritmos comience por min, pero podría haber empezado perfectamente por max de la misma forma.

Después de generar los sucesores llama a max a la espera de retorno.

- `maxxd(sucesor, profundidad_maxima, alfa, beta)` se encarga de hacer la evaluación max de los estados.

El algoritmo sigue el pseudocódigo proporcionado como se espera: genera sus sucesores y hace la llamada a `_minxd()`, y al retornar los resultados alfa-beta los actualiza en el nodo padre.

Importante:

La diferencia entre nuestro algoritmo minmax y alfa-beta esta en que en las funciones de `minxd()` y `maxxd()` de alfa-beta hemos añadido una línea que evalúa `if beta <= alfa`, en este caso hacemos un break de la función ya que esa rama del árbol nos da posibles valores que no nos interesan (no nos hace falta cambiar la expresión ya que en cada llamada a estas funciones el valor de alfa y beta se van cambiando siguiendo el algoritmo explicado en clase).

iii. Implementación.

```
class MinimaxStrategy(Strategy):
    """Minimax 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:
    """Computa el siguiente estado en el juego"""
    #EMPEZAMOS A CONTAR EL TIEMPO DE EJECUCION DEL ALGORITMO
    start = time.process_time()
    minimax_value, minimax_successor = self._maxxd(
        state,
        self.max_depth_minimax,
    )

    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))

    #GUARDAMOS LOS VALORES EN UN .TXT
    ejecucion = str(time.process_time() - start)
    with open('medidasMinMax.txt', 'a') as f:
        f.write(ejecucion)
        f.write("\n")

    return minimax_successor

def _maxxd(
    self,
    state: TwoPlayerGameState,
    depth: int,
) -> float:
    """Paso Max del algoritmo"""

    if state.end_of_game or depth == 0:
        minimax_value = self.heuristic.evaluate(state)
        minimax_successor = None
    else:

```

```

minimax_value = np.inf

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

    successor_minimax_value, _ = self._maxxd(
        successor,
        depth - 1,
    )

    if (successor_minimax_value < minimax_value):
        minimax_value = successor_minimax_value
        minimax_successor = successor

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

return minimax_value, minimax_successor

def _maxxd(
    self,
    state: TwoPlayerGameState,
    depth: int,
) -> float:
    """Paso Max del algoritmo"""

    if state.end_of_game or depth == 0:
        minimax_value = self.heuristic.evaluate(state)
        minimax_successor = None
    else:
        minimax_value = -np.inf

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

        successor_minimax_value, _ = self._minxd(
            successor,
            depth - 1,
        )
        if (successor_minimax_value > minimax_value):

```

```

        minimax_value = successor_minimax_value
        minimax_successor = successor

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

    return minimax_value, minimax_successor


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:
        """Computa el siguiente estado en el juego"""
        #EMPEZAMOS A CONTAR EL TIEMPO DE EJECUCION DEL ALGORITMO
        start = time.process_time()
        #Primero obtenemos los sucesores del estado
        succs = self.generate_successors(state)

        # Asignamos el valor mas pequeño para alpha y el maximo para beta
        alpha = -np.inf
        beta = np.inf

        # Para cada sucesor del estado raiz
        for succ in succs:
            if self.verbose > 1:
                print('{}: [{}, {}]'.format(state.board, alpha, beta))

```

```

min_value = self._minxd(succ, self.max_depth_minimax, alpha, beta)

# Obtenemos el valor maximo para beta
if (min_value > alpha):
    alpha = min_value
    minimax_successor = succ

if self.verbose > 0:
    if self.verbose > 1:
        print('\nGame state before move:\n')
        print(state.board)
        print()
    print('Alpha value = {:.2g}'.format(alpha))
#GUARDAMOS LOS VALORES EN UN .TXT
ejecucion = str(time.process_time() - start)
with open('medidasAlfaBeta.txt', 'a') as f:
    f.write(ejecucion)
    f.write("\n")
return minimax_successor

def _maxxd(
    self,
    state: TwoPlayerGameState,
    depth: int,
    alpha: int,
    beta: int,
) -> float:

    # Alcazaos el nivel mas bajo del ultimo nodod
    if state.end_of_game or depth == 0:
        maxxd = self.heuristic.evaluate(state)
    else:
        maxxd = -np.inf

    succs = self.generate_successors(state)
    for succ in succs:
        # En el caso de que el mas bajo sea mayor que el mas alto, podemos hacer "pruning"
        # de esta rama
        if beta <= alpha:
            break
        min_value = self._minxd(succ, self.max_depth_minimax, alpha, beta)

        # Obtenemos el valor maximo para beta
        if (min_value > alpha):
            alpha = min_value
            minimax_successor = succ

        if self.verbose > 0:
            if self.verbose > 1:
                print('\nGame state before move:\n')
                print(state.board)
                print()
            print('Alpha value = {:.2g}'.format(alpha))
#GUARDAMOS LOS VALORES EN UN .TXT
ejecucion = str(time.process_time() - start)
with open('medidasAlfaBeta.txt', 'a') as f:
    f.write(ejecucion)
    f.write("\n")
return minimax_successor

```

```

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

    minxd = self._minxd(
        succ, depth - 1, alpha, beta,
    )

    # Obtenemos el maximo valor de todos los descendientes
    if minxd > maxxd:
        maxxd = minxd

    # Queremos que alfa tenga el valor mayor de todos
    if maxxd > alpha:
        alpha = maxxd


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

return maxxd

def _minxd(
    self,
    state: TwoPlayerGameState,
    depth: int,
    alpha: int,
    beta: int,
) -> float:

    # Alcazaos el nivel mas bajo del ultimo nodod
    if state.end_of_game or depth == 0:
        minxd = self.heuristic.evaluate(state)
    else:
        minxd = np.inf

    succs = self.generate_successors(state)
    for succ in succs:
        # En el caso de que el mas bajo sea mayor que el mas alto, podemos hacer "pruning"
        # de esta rama
        if (beta <= alpha):

```

```

break

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

maxxd = self._maxxd(
    succ, depth - 1, alpha, beta,
)

# Obtenemos el menor valor de todos los descendientes
if (maxxd < minxd):
    minxd = maxxd

# Queremos que beta tenga el valor mas pequeño
if (minxd < beta):
    beta = minxd

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

return minxd

```

iv. Otra información relevante.

Hemos implementado líneas de código que almacenan en un fichero el tiempo de ejecución de ambos algoritmos para el posterior análisis usando la librería “time”.

También se ha implementado un pequeño programa en python donde se calcula la media de tiempo de ejecución y luego hace un cociente entre los valores.

b. Eficiencia de la poda alfa-beta.

i. Descripción completa del protocolo de evaluación.

Para evaluar los tiempos de ejecución hemos implementado líneas de código que almacenan en un fichero el tiempo de ejecución de ambos algoritmos para el posterior análisis.

Hemos valorado dos casos: el primero en el que minmax es el algoritmo para todos los jugadores, y luego en otra ejecución igual pero con poda alfa-beta.

Los ficheros son medidasMinMax.txt y medidasAlfaBeta.txt
Las dos heurísticas que hemos comparado son la dummy que viene implementada en la documentación de la práctica y la otra es nuestra mejor heurística (según el ranking del torneo de heurísticas), ambas han jugado dos partidas entre ellas dándonos un total de 115 valores de tiempo de ejecución para cada algoritmo (almacenadas en medidasMinMax.txt y medidasAlfaBeta.txt)

La salida por consola es la siguiente:

Results for tournament where each game is repeated 2=1x2 times, alternating colors for each player

	total:	opt1_dummy	opt2_catxo
opt1_dummy	1:	---	1
opt2_catxo	1:	1	---

ii. Tablas donde se incluyan tiempos con y sin poda.

Se ha implementado un pequeño programa en python llamado “getTimeImprovement.py” donde se calcula la media de tiempo de ejecución y luego hace un cociente entre los valores.

Con la salida de este programa podemos realizar la tabla de tiempos:

	MinMax	PodaAlfaBeta
Tiempo promedio (s)	0.1900	0.2870

$$\text{Cociente averageMinMax/averageAlfaBeta} = 0.66201$$

iii. Medidas de mejora independientes del ordenador.

No hay medidas de mejora independientes al ordenador

iv. Análisis correcto, completo y claro de los resultados.

Como se puede observar, en el apartado ii), la poda AlfaBeta es significativamente más lenta que MinMax.

Esto no debería de ser así, ya que ambos algoritmos están implementados de forma muy parecida, pero no hemos conseguido averiguar cual es la causa de que estas mediciones pongan de manifiesto lo contrario que teóricamente debería ocurrir:

La poda alfa beta al encontrar un nodo el cual cumple que $\text{alfa} \leq \text{beta}$, ese nodo y todos sus hermanos no se tienen en cuenta a la hora de actualizar alfa, y se pasa a evaluar el siguiente nodo.

La única razón por la que la poda AlfaBeta pueda estar obteniendo valores de tiempo peores es por la demora en la comparación anterior ($\text{¿alfa} \leq \text{beta?}$), pero desde el punto de vista computacional, el coste de este es ridículo en comparación a las ventajas que tiene no visitar más nodos para ser valorados.

El código sigue el algoritmo a implementar, y no hemos sido capaces de encontrar la razón de la demora al ejecutar

v. Otra información relevante.

Ninguna

2. Documentación del diseño de la heurística.



- a. Revisión de trabajos previos sobre estrategias de Reversi strategies, incluyendo referencias en el formato APA.

<https://stackoverflow.com/questions/13314288/need-heuristic-function-for-reversi-thello-ideas>

http://home.datacomm.ch/t_wolf/tw/misc/reversi/html/index.html

- b. Descripción del proceso de diseño:
 - i. ¿Cómo se planeó y ejecutó el proceso de diseño?

Tras comprobar las fuentes, descubrimos una heurística sencilla la cual tiene en cuenta la puntuación. Posteriormente descubrimos la importancia de obtener esquinas por lo que estas eran valoradas positivamente, pero las adyacentes a las esquinas eran valoradas negativamente, al hacer posible que el oponente adquiera esquinas. Algo similar ocurre también con los bordes.

- ii. ¿Seguiste algún procedimiento sistemático para evaluar las heurísticas diseñadas?

Hemos comprobado las heurísticas usando los tests (automáticos y manuales) incluidos con la práctica para ver un primer resultado del

rendimiento. A medida que teníamos más heurísticas también podíamos compararlas entre ellas para ver qué estrategia es mejor o peor. Algo similar ocurre con los torneos donde podíamos comprobar el rendimiento frente al de las diseñadas por otros compañeros.

iii. ¿Utilizaste ideas desarrolladas por otros para mejorar las estrategias diseñadas? Si están disponibles públicamente, incluye referencias en formato APA; en otro caso, incluye el nombre de la persona que te dio la información y dale el crédito oportuno como “comunicación privada”

Únicamente hemos utilizado las fuentes mencionadas arriba

.c. Descripción de la heurística enviada finalmente.



Nuestra heurística tiene en cuenta primeramente las puntuación que se va a obtener al realizar cierto movimiento, para obtener un valor inicial.

Dicho valor no es muy relevante por lo que posteriormente lo ajustamos en función del tipo de casilla que tengamos.

- Esquinas (muy favorable)
- Borde (favorable)
- Adyacente a borde (penalización leve)
- Adyacente a esquina (desfavorable)
- Diagonal a esquinas (muy desfavorable)

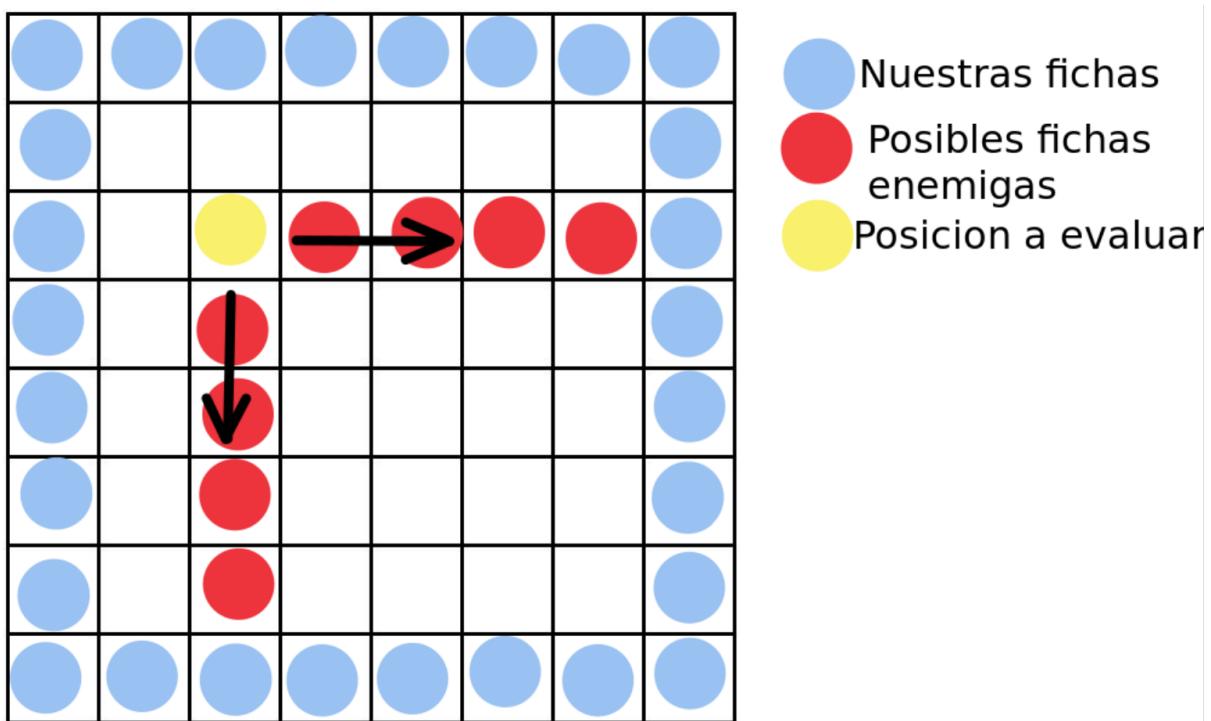
	A	B	C	D	E	F	G	H
1	Blue	Orange	Cyan	Cyan	Cyan	Orange	Blue	
2	Orange	Red	Yellow	Yellow	Yellow	Red	Orange	
3	Cyan	Yellow	Green	Green	Green	Yellow	Cyan	
4	Yellow	Yellow	Green	Green	Green	Yellow	Cyan	
5	Cyan	Yellow	Green	Green	Green	Yellow	Cyan	
6	Yellow	Yellow	Green	Green	Green	Yellow	Cyan	
7	Orange	Red	Yellow	Yellow	Yellow	Red	Orange	
8	Blue	Orange	Cyan	Cyan	Cyan	Orange	Blue	

Las posiciones penalizadas, ayudarían al enemigo a obtener una posición favorable.

Las casillas a la larga favorecen al jugador haciendo que sea más probable que en futuras jugadas las fichas sean comidas por nosotros minimizando el riesgo de que nuestras fichas sean comidas.

En la última entrega de heurísticas implementamos un sistema de detección de fichas enemigas en posiciones comprometidas, llamada “aVerEsaTransa()” en la Solution3, la cual miraba si en las casillas que se encontraban directamente adyacentes en posición opuesta a la esquina más próxima (respecto a la coordenada de la casilla que estábamos evaluando) estaban ocupadas por el jugador contrario.

De ser así era más probable que eventualmente esas casillas fueran comidas, ya que nuestra heurística premiaba de forma generosa hacerse con los lados y esquinas del tablero, haciendo teóricamente muy efectivo, pero computacionalmente más lento.



Para nuestro infortunio, esta mejora no trajo ninguna ventaja sino que quedó por detrás de la Solution 1, nuestra solución inicial y ganadora de todas.

Solution2 fue simplemente un reajuste de ponderaciones de casillas en el tablero con la esperanza de obtener resultados significativamente mejores, no fue el caso.

d. Otra información relevante.

