Memoria P2 - Reversi

Título: Reversi

Autores: Daniel Cerrato y David Garitagoitia

Sección 1

1.1. **Documentación de minimax y poda alfa-beta**

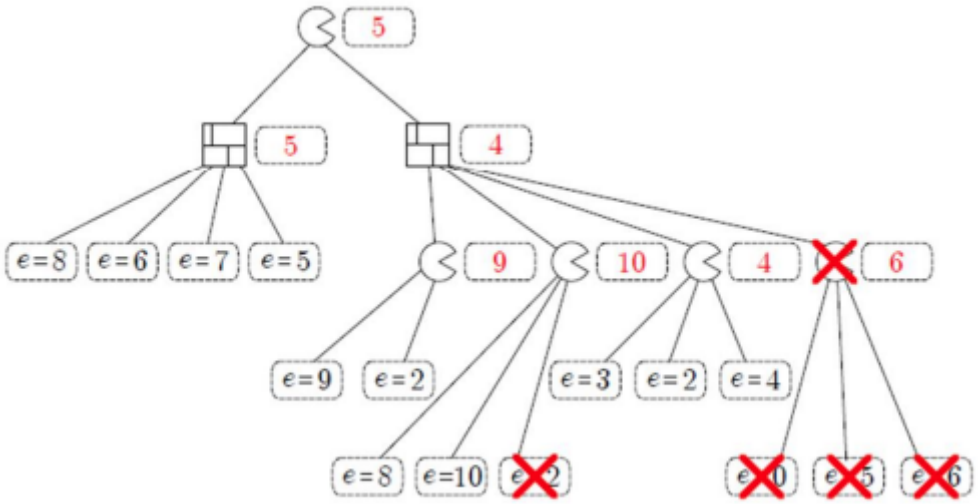
1.1.1. **Detalles de implementación**

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

Los tests que se han llevado a cabo han sido las comparaciones de las salidas por pantalla que ofrece el programa con el seguimiento a mano de las salidas esperadas, tanto para el árbol que se nos facilita con el código de la práctica, como con otro árbol cogido de los ejemplos de teoría.

1.1.1.2. **Diseño: estructuras de datos seleccionadas, descomposición funcional, etc.**

El árbol usado de la parte teórica es el siguiente:



Los números en rojo son los valores obtenidos finalmente por cada nodo. Este ejemplo se usa en los ejercicios de teoría para el algoritmo de poda alfa-beta, pero el resultado es el mismo.

1.1.1.3. **Implementación**

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

    minimax_value, minimax_successor = self._max_value(
        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))

    if minimax_successor:
        minimax_successor.minimax_value = minimax_value

    return minimax_successor
```

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

    minimax_value, minimax_successor = self.max_value(
        state,
        -np.inf,
        np.inf,
        self.max_depth_minimax,
    )

    """# Use this code snippet to trace the execution of the algorithm
    if self.verbose > 1:
        print('{}: [{:.2g}, {:.2g}]'.format(
                state.board,
                alpha,
                beta,
            )
        )"""

    if minimax_successor:
        minimax_successor.minimax_value = minimax_value

    return minimax_successor
```

Esta función es la principal del algoritmo minimax y de la poda alfa-beta. Se divide en tres partes:

1. Llamada a la función que calcula el valor del primer nodo (MAX), pasándole por argumento el estado del juego actual y una profundidad máxima a la que se quiere llegar. En el caso de la poda alfa-beta, se pasan por argumento un alfa y un beta iniciales con el valor de -∞ e ∞, respectivamente. Se recogen como retorno el valor minimax que se ha escogido en el primer nodo MAX y el estado sucesor al que se va a llegar con esa decisión.
2. Impresión por pantalla del estado antes del movimiento escogido y del valor minimax obtenido tras el algoritmo.
3. Comprobación del resultado del algoritmo: si se ha obtenido un sucesor es porque el juego aún no ha terminado, así que en este caso, se guardaría el valor minimax obtenido. Finalmente, se devuelve el estado sucesor, sea el que sea.

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

    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._min_value(
                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
```

Esta función contiene el algoritmo de decisión de los nodos MAX. El algoritmo se basa en el que se ha visto en teoría:

Inicialmente, se comprueba si el estado que se está comprobando es el final o si se ha llegado a la profundidad máxima permitida. En caso afirmativo, simplemente se sube en el árbol, retornando el valor del estado recogido en la heurística propuesta y nada como sucesor, pues el sucesor no es importante en los nodos hoja del árbol, ni tiene sentido devolver un sucesor de un estado final.

En caso de no estar en ninguna de esas dos situaciones, el algoritmo se lleva a cabo. Se generan y recorren los sucesores del estado recibido. Primero, se baja en el árbol pasando al algoritmo de los nodos MIN, pasando por argumento lo mismo que se recibió. De esta llamada a función se recogen el valor minimax calculado en el nodo inferior y el estado sucesor, pero este último no es importante, así que lo tomamos como variable anónima ('_').

```python
def max_value(self, state: TwoPlayerGame, alpha, beta, depth) -> float:

    max_successor = None

    if state.end_of_game or depth == 0:
        return self.heuristic.evaluate(state), max_successor

    for successor in self.generate_successors(state):
        if self.verbose > 1:
            print('{}: [{:.2g}, {:.2g}]'.format(
                    state.board,
                    alpha,
                    beta,
                )
            )
        aux_alpha, _ = self.min_value(successor, alpha, beta, depth-1)
        if aux_alpha >= beta:
            return aux_alpha, successor
        if alpha < aux_alpha:
            alpha = aux_alpha
            max_successor = successor

    return alpha, max_successor
```

En ambos algoritmos, se comprueba si el valor obtenido del nodo inferior es mayor que el valor minimax actual, en caso afirmativo, se guarda este valor como el actual de minimax y se guarda el estado sucesor del que se ha obtenido.

Sin embargo, en el algoritmo de poda, es justo antes de esto que se comprueba el valor de beta. Como estamos en un nodo MAX, si el valor obtenido del nodo inferior es mayor o igual que el valor de beta del nodo actual, entonces se produce la poda. En el código, la poda se realiza a través de un "return" directo, devolviendo el valor del nodo inferior sin compararlo con el alpha actual y el último sucesor recogido. Esto es así porque da igual enviar el valor alfa obtenido en último sucesor, como es un valor mayor que el beta, el nodo MIN que pueda haber por encima, preferirá su beta que el alfa que se le acaba de enviar.

El algoritmo de los nodos MIN es exactamente igual. Las únicas modificaciones que se hacen son la llamada a la función de max_value para seguir bajando en el árbol y cambiar alfa por beta, el nombre de aux_alpha por aux_beta y los símbolos de mayor y menor en las comprobaciones finales del bucle. Estos cambios son necesarios porque los nodos MIN tratan de reducir el beta al mínimo posible, de manera que el beta que nos llegue del nodo MAX inferior puede llegar a ser menor que el que ya se tiene o que este sea menor o igual que el alfa que ya se tiene, lo que produciría una poda.

1.1.2. **Eficiencia de la poda alfa-beta**

1.1.2.1. **Descripción completa del protocolo de evaluación**

Se van a realizar varias pruebas de tiempos y cantidad de nodos explorados para comprobar la eficiencia del algoritmo minimax sin poda y con poda. Para ello se realizarán pruebas con las demo de torneo y de árbol de juego simple.

Empezaremos las pruebas con el árbol de juego simple, donde se utilizarán los dos árboles comentados en el apartado 1.1.1.2: el dado junto a la práctica y el recogido de la parte teórica.

Posteriormente, se llevarán a cabo las pruebas en el torneo, usando las heurísticas proporcionadas al inicio de la práctica para ambos algoritmos minimax, de esta forma nos aseguramos de que no hay irregularidades por esa parte.

El orden de las pruebas será el siguiente:

Primero, se realizarán las pruebas temporales de los dos algoritmos con los dos árboles y, después, con el torneo.

Luego, se llevarán a cabo las pruebas ajenas al ordenador con el mismo orden que las pruebas temporales.

Con los árboles se realizarán 20 ejecuciones del código, ya que es un proceso corto, recibiendo así medidas de sobra para estar más seguros de recoger mejores medidas y tener más posibilidades de recoger medidas más bajas. Mientras tanto, para los torneos, se usarán las dos primeras heurísticas proporcionadas en el código que se nos entregó al inicio de la práctica y se correrán 5 torneos, recogiendo los tiempos por separado de cada partida de cada torneo, de esta manera, tendremos más o menos los mismos tiempos.

Con los tiempos obtenidos, se calculará la media y este valor será la referencia para comparar los algoritmos. Además, guardaremos el menor tiempo para tener una mejor apreciación de lo sucedido. Evidentemente, el valor máximo no nos servirá de nada, pues lo más probable es que se haya visto afectado en mayor medida por el ordenador.

Hay que tener en cuenta que en las medidas temporales, el ordenador puede afectar a los tiempos de procesado, así que el menor tiempo que salga, no tiene porque ser el mínimo absoluto.

En la parte de medidas independientes del ordenador, no existirá este problema, así que las medidas deberían tener el mismo valor para el mismo algoritmo en la misma prueba, salvo para los torneos, que variarán gracias a que las heurísticas no siempre devuelven los mismos valores para cada estado y que los jugadores se intercambian el turno inicial en cada torneo.

1.1.2.2. **Tablas donde se incluyan tiempos con y sin poda**

Tiempos

Simple Game Tree - Árbol Práctica		
Ronda	Tiempo sin poda	Tiempo con poda
1	0,00319	0,00208
2	0,00258	0,00197
3	0,00237	0,00191
4	0,00259	0,00198
5	0,00260	0,00188
6	0,00250	0,00191
7	0,00216	0,00191
8	0,00235	0,00189
9	0,00264	0,00180
10	0,00235	0,00182
11	0,00230	0,00184
12	0,00215	0,00222
13	0,00229	0,00180
14	0,00227	0,00190
15	0,00226	0,00183
16	0,00225	0,00180
17	0,00225	0,00181
18	0,00235	0,00185
19	0,00221	0,00185
20	0,00222	0,00201
Media	0,00239	0,00190
Menor	0,00215	0,00180

Simple Game Tree - Árbol Teoría		
Ronda	Tiempo sin poda	Tiempo con poda
1	0,00326	0,00269
2	0,00329	0,00260
3	0,00322	0,00267
4	0,00368	0,00259
5	0,00364	0,00287
6	0,00324	0,00277
7	0,00365	0,00246
8	0,00387	0,00246
9	0,00321	0,00259
10	0,00308	0,00305
11	0,00310	0,00260
12	0,00302	0,00242
13	0,00329	0,00247
14	0,00299	0,00246
15	0,00310	0,00248
16	0,00300	0,00257
17	0,00309	0,00241
18	0,00308	0,00251
19	0,00302	0,00251
20	0,00306	0,00252
Media	0,00324	0,00259
Menor	0,00299	0,00241

Tournament		
Ronda	Tiempo sin poda	Tiempo con poda
1	18,975	5,453
2	16,105	5,795
3	15,430	5,677
4	10,959	5,746
5	18,719	6,982
6	15,620	6,913
7	11,660	6,617
8	12,035	4,757
9	18,596	6,901
10	12,167	6,832
Media	16,038	5,931
Menor	10,959	4,757

Cantidad de nodos explorados

Simple Game Tree - Árbol Práctica		
Ronda	Nodos sin poda	Nodos con poda
1	23	20
2	23	20
3	23	20
4	23	20
5	23	20
6	23	20
7	23	20
8	23	20
9	23	20
10	23	20
11	23	20
12	23	20
13	23	20
14	23	20
15	23	20
16	23	20
17	23	20
18	23	20
19	23	20
20	23	20
Media	23	20
Menor	23	20

Simple Game Tree - Árbol Teoría		
Ronda	Nodos sin poda	Nodos con poda
1	27	22
2	27	22
3	27	22
4	27	22
5	27	22
6	27	22
7	27	22
8	27	22
9	27	22
10	27	22
11	27	22
12	27	22
13	27	22
14	27	22
15	27	22
16	27	22
17	27	22
18	27	22
19	27	22
20	27	22
Media	27	22
Menor	27	22

Tournament		
Ronda	Nodos sin poda	Nodos con poda
1	27581	7195
2	21117	5848
3	30234	6979
4	23452	5702
5	29286	7093
6	20275	6448
7	23886	6089
8	22290	6409
9	30692	5796
10	19578	7041
Media	26334	6563
Menor	19578	5702

1.1.2.3. **Medidas de mejora independientes del ordenador**

Como medida para conocer la mejora del algoritmo minimax al incorporar la poda, hemos usado el módulo “timeit” que se indica en la práctica, pero es verdad que medir los tiempos de un algoritmo a través del tiempo de procesado pueden verse afectados por las especificaciones del ordenador en el que se estén llevando a cabo.

Para evitar esto, se ha pensado que una buena medida sería calcular el número de nodos visitados sumando la cantidad de nodos de ambos jugadores durante una partida. Esta medida es ajena al ordenador de uso ya que mide la cantidad de nodos, lo que depende del propio algoritmo y no del tiempo de procesado de cada nodo. Medir la cantidad de nodos es lo importante, ya que la diferencia entre el algoritmo con poda y sin ella es la cantidad de nodos que se exploran y la “toma de decisiones” de MIN Y MAX.

1.1.2.4. **Análisis correcto, completo y claro de los resultados**

Analizando los tiempos y los nodos explorados en general, se observa claramente que la poda afecta positivamente en el rendimiento de las pruebas, como era de esperar.

Pero ahora analicemos más detenidamente los casos:

En tiempos, no hay mucho que comentar, el ordenador no ha trastocado prácticamente las pruebas y los resultados son concluyentes: la poda evita la sobrecarga de cálculo y termina siempre antes dando las misma decisiones.

Los torneos son más de lo mismo, solo que como en los torneos se hacen varias llamadas a la función next_move, se hace más evidente la diferencia de tiempos, llegando a estar el algoritmo con poda 10 segundos por debajo de media y 5 segundos por debajo en la medida más baja. Esta diferencia se genera a partir de un estado de juego avanzado, lo que indica que ya no hay tantos nodos que explorar y, por tanto, en general, se tardará relativamente “poco” en calcular el siguiente movimiento. Si esto lo llevamos a toda una partida, desde el inicio, los tiempos se dispararían y la diferencia se haría aún más notable.

En la parte de nodos explorados, para los árboles y como era de esperar, los nodos explorados son siempre los mismos para cada algoritmo. Esto sucede porque los árboles son los mismos y los valores de los nodos hoja no cambian, por lo que en todos los nodos se toman las mismas decisiones. Aquí también se hace notar el uso de poda, evitando tener que llegar a nodos innecesarios o irrelevantes para la decisión final.

En el torneo sí que varía la cantidad de nodos explorados en un mismo algoritmo. Esto sucede porque el valor de los nodos hoja sí que varía al usar heurísticas que generan valores aleatorios en cada llamada, de manera que las decisiones en los nodos intermedios cambian, y unas veces la poda se realizará antes y otras después.

De cualquier forma, es evidente que la poda también afecta positivamente en este caso. Es más, como se observa en los números, la diferencia de nodos explorados es 4 veces menor tanto de media como en los mejores casos. Repito, esto se ha medido en un estado de juego avanzado, desde el inicio, estos datos se dispararían y la diferencia se haría notar mucho más.

Como conclusión, simplemente decir que, como se ha hecho evidente con estas pruebas básicas, el uso de la poda en el algoritmo minimax hace que la toma de decisiones sea mucho más rápida y eficiente, tanto temporalmente como en cantidad de cálculo y memoria.

1.1.2.5. **Otra información relevante**

```
import timeit

rounds = 20
while rounds > 0:
    initial_time = timeit.default_timer()

    scores = match.play_match()

    final_time = timeit.default_timer()

    print(final_time - initial_time)

    rounds -= 1
```

```
#player1, player2 = player1_minimax, player2_minimax
player1, player2 = player1_minimax_alpha_beta, player2_minimax_alpha_beta
```

Para conseguir los tiempos de los árboles, simplemente se ha generado el código de la primera imagen y se ha ido intercambiando la línea comentada de la segunda, además de intercambiar el comentario de los códigos para generar los árboles como se puede ver abajo.

```
"""self._successor_lists = {
    'A': ['B', 'C', 'D'],
    'B': ['E', 'F'],
    'C': ['G', 'H'],
    'D': ['I', 'J'],
    'G': ['K', 'L'],
    'J': ['M','N','O','P']
}
self._terminal_state_scores = {
    'E': [4, 0],
    'F': [3, 0],
    'H': [5, 0],
    'I': [5, 0],
    'K': [2, 0],
    'L': [1, 0],
    'M': [4, 0],
    'N': [2, 0],
    'O': [6, 0],
    'P': [1, 0],
}"""
```

```
self._successor_lists = {
    'A': ['B', 'C'],
    'B': ['D', 'E', 'F', 'G'],
    'C': ['H', 'I', 'J', 'K'],
    'H': ['L', 'M'],
    'I': ['N', 'O', 'P'],
    'J': ['Q', 'R', 'S'],
    'K': ['T', 'U', 'V'],
}
self._terminal_state_scores = {
    'D': [8, 0],
    'E': [6, 0],
    'F': [7, 0],
    'G': [5, 0],
    'L': [9, 0],
    'M': [2, 0],
    'N': [8, 0],
    'O': [10, 0],
    'P': [2, 0],
    'Q': [3, 0],
    'R': [2, 0],
    'S': [4, 0],
    'T': [0, 0],
    'U': [5, 0],
    'V': [6, 0],
}
```

Para los torneos, se ha generado el código de las primeras imágenes que vienen a continuación y, para comparar minimax con poda y sin poda, simplemente se ha cambiado el nombre de la clase a la que se hace la llamada, como se puede ver en la última imagen.

```
rounds = 5
while rounds > 0:
    scores, totals, names = tour.run(
        student_strategies=strats,
        increasing_depth=False,
        n_pairs=n,
        allow_selfmatch=False,
    )

    rounds -= 1
```

```
import timeit

initial_time = timeit.default_timer()

self.__single_run(player1_first, pl1, name1, pl2, name2, scores, totals)

final_time = timeit.default_timer()

print(final_time - initial_time)
```

```
pl1 = Player(
    name=name1,
    strategy=MinimaxStrategy(  #MinimaxAlphaBetaStrategy(
        heuristic=Heuristic(name=sh1.get_name(), evaluation_function=sh1.evaluation_function),
        max_depth_minimax=depth,
        verbose=0,
    ),
)
pl2 = Player(
    name=name2,
    strategy=MinimaxStrategy(  #MinimaxAlphaBetaStrategy(
        heuristic=Heuristic(name=sh2.get_name(), evaluation_function=sh2.evaluation_function),
        max_depth_minimax=depth,
        verbose=0,
    ),
)
```

Para la parte de la comparación con datos ajenos al ordenador, se seguirán intercambiando los comentarios necesarios para cambiar de árbol y de estrategia, pero se ha generado código nuevo para calcular el número de nodos explorados.

Para el cálculo de nodos basta con generar una variable que guarde el número de nodos explorados de cada clase minimax inicializado a 0, hacer que cada llamada a las funciones de max_value y min_value aumenten este contador y, en el momento de impresión de los datos, se impriman estos valores y se restauren a 0. Esto último se puede realizar con funciones getter y setter simples.

Sección 2

2.1. Documentación del diseño de la heurística

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

Se revisaron múltiples estudios del juego además de manuales y guías de estrategia entre las que destacamos:

Kukreja, K. (2013, March 30). Heuristic/Evaluation Function for Reversi/Othello. Kartikkukreja. https://kartikkukreja.wordpress.com/2013/03/30/heuristic-function-for-reversiothello/

Reversi Strategy. (n.d.). Https://Documentation.Help/Reversi-Rules/Strategy.Htm

Cherry, K. A. (May 2011). An intelligent Othello player combining machine learning and game specific heuristics [Louisiana State University]. https://digitalcommons.lsu.edu/cgi/viewcontent.cgi?article=1766&context=gradschool_theses

Lu, K. (2014) The Game Theory of Reversi

Sannidhanam, V. (2015) An Analysis of Heuristics in Othello [University of Washington].
https://courses.cs.washington.edu/courses/cse573/04au/Project/mini1/RUSSIA/Final_Paper.pdf

2.1.2. Descripción del proceso de diseño

2.1.2.1. ¿Cómo se planeó y ejecutó el proceso de diseño?

El primer paso fue jugar varias partidas del juego online hasta asentar una base intuitiva de las jugadas que podían ser más prometedoras y manejarnos con las bases del juego. Tras realizar varias partidas, llegamos a la conclusión de que las esquinas eran muy importantes, puesto que estas no pueden ser volteadas, lo que supone puntos asegurados.

En la línea de esto, también gozarán de gran importancia las casillas que faciliten la captura de las esquinas y de menor valor aquellos que comprometan las esquinas, por lo que será preferible evitar las casillas contiguas a las esquinas, mientras que las que están a dos casillas de distancia de las esquinas serán ventajosas al facilitar la captura de las esquinas.
Con ello procedimos a crear las primeras heurísticas, la primera de ellas simplemente retornaba la diferencia de fichas de nuestro jugador y el rival, el objetivo de esta heurística tan básica será el de servir como primer punto de referencia para el resto de heurísticas.

```
 if state.is_player_max(state.player1): #si somos el jugador 1
     return 64 - state.scores[0] 8*8=64 es el max
   else:
     return 64 - state.scores[1] #si somos el jugador 2

dif = state.scores[0]-state.score[1]
if state.is_player_max(state.player1):
     return dif
else:
     return -dif
```

Una vez hecho esto, tras informarnos para proceder con una siguiente heurística vimos como varias referencias coincidían con la importancia de buscar estados donde se incremente el número de posibles movimientos[1], así como reducir las opciones del rival, es por ello que la siguiente heurística que decidimos probar únicamente tenía en cuenta este factor.

```
mov=len(state.game._get_valid_moves(state.board,state.player1.label)) -
len(state.game._get_valid_moves(state.board,state.player2.label))
   if state.is_player_max(state.player1): #si somos el jugador 1
     return mov
   return -mov
```

[1] Kukreja, K. (2013, March 30). Heuristic/Evaluation Function for Reversi/Othello. Kartikkukreja.
https://kartikkukreja.wordpress.com/2013/03/30/heuristic-function-for-reversiothello/

Al poseer en este punto dos heurísticas se procedió a comprobar si esta última lograba sacar ventaja a la primera, a lo cual obtuvimos un resultado favorecedor.
Tras ello unimos ambas heurísticas para tener en cuenta tanto movilidad como el número de fichas y procedimos a probar la siguiente heurística.
En esta nueva heurística mantendremos los avances anteriores y añadiremos un nuevo factor que como ya vimos en nuestras primeras pruebas, es de gran importancia, las esquinas, para este simplemente le añadiremos a la ecuación la diferencia entre nuestras esquinas y las del rival y volveremos a revisar si mejora la heurística anterior, tras tener una respuesta positiva en el torneo continuamos con la evolución de la heurística.

```
mov1 = len(state.game._get_valid_moves(state.board,state.player1.label))
mov2 = len(state.game._get_valid_moves(state.board,state.player2.label))
p1 = state.scores[0]
p2 = state.scores[1]

for i in esqi:
    for j in esqj:
       pos = state.board.get((i+1, j+1), '_',)
       if(pos == state.player1.label): #si en esa esquina esta el jugador 1
         c1 = c1+1 #sumamos 1 a las esquinas de jugador 1
       elif (pos == state.player2.label): #si en esa esquina esta el jugador 2
         c2 = c2+1 #sumamos 1 a las esquinas de jugador 2
result=0
  if(c1+c2!=0):
    result=result + (70/100) * ((c1-c2)/(c1+c2)*100)
  if(mov1+mov2!=0):
    result=result + (15/100) *((mov1-mov2)/(mov1+mov2)*100)
  if(p1+p2!=0):
    result+=result + (15/100) *((p1-p2)/(p1+p2)*100)
```

Analizando y estudiando diferentes análisis del juego[2] llegamos a la idea de crear un mapa de influencia, dotando de valor a las distintas casillas del tablero.

[2] Cherry, K. A. (May 2011). An intelligent Othello player combining machine learning and game specific heuristics [Louisiana State University].
https://digitalcommons.lsu.edu/cgi/viewcontent.cgi?article=1766&context=gradschool_theses

5.2 Static Weights Heuristic Function

4	-3	2	2	2	2	-3	4
-3	-4	-1	-1	-1	-1	-4	-3
2	-1	1	0	0	1	-1	2
2	-1	0	1	1	0	-1	2
2	-1	0	1	1	0	-1	2
2	-1	1	0	0	1	-1	2
-3	-4	-1	-1	-1	-1	-4	-3
4	-3	2	2	2	2	-3	4

Figure 3: Shows the static weights assigned to each individual position in the board

De esta forma dotaremos de mayor valor las posiciones de las esquinas y de menor valor las posiciones adyacentes a éstas, pues como ya vimos en el primer análisis del juego, no son ventajosas al ofrecer al rival la posibilidad de capturar una esquina, también las posiciones del centro tendrán menor valor (a diferencia de lo que vemos en el análisis referenciado anteriormente en el que las fichas del medio tienen valor 1, probando decidimos dotarlas de valores negativos), ya que ofrecen la posibilidad de perder muchas fichas en pocos movimientos también ajustamos el resto de zonas mediante prueba y error y otros tableros de influencia encontrados por internet, había ocasiones en las que ganaba uno y ocasiones en las que ganaba otro así que tenemos distintos tableros

Vimos otros factores importantes como la estabilidad o las fronteras estudiando algunos programas.
Con todo ello pusimos en práctica la primera gran heurística, ajustamos un poco los pesos de la matriz de influencia y fuimos recogiendo aspectos en distintos torneos hasta llegar al final.

Por último simplificamos el código par hacerlo más eficiente y probamos alguna heurística más, incluyendo datos como la frontera

```
else: #posición vacía (comprobar discos frontera) arriba, abajo, d a der, d a izq, der, izq, d ab der, d ab izq
        for aux in range(8): #comprobamos las 8 esquinas
            pos = state.board.get((i+1 + diri[aux], j+1 + dirj[aux]), '_',) #vemos que estaba adyacente a ese espacio vacio
            if(pos == state.player1.label):
              fm1+=1 #si es de jugador 1, el jugador 1 tiene una ficha frontera más
            elif (pos == state.player2.label):
              fm2+=1 #si es de jugador 2, el jugador 2 tiene una ficha frontera más
        continue
```

O las casillas x[3], estuvimos un tiempo ajustando pesos, pero al no notar mejoría, ya que a veces perdía y otras ganaba sin una clara tendencia decidimos tomarla como definitiva.

[3] Reversi Strategy. (n.d.). Https://Documentation.Help/Reversi-Rules/Strategy.Htm

2.1.2.2. **¿Seguiste algún procedimiento sistemático para evaluar las heurísticas diseñadas?**

Como ya hemos hablado en el apartado anterior, cada vez que se realizaban cambios significativos en las heurísticas se hacían comprobaciones mediante torneos para ver si había una mejora significativa, también en las primeras heurísticas se jugaba manualmente algunas rondas para ver rápidamente algunos aspectos en los que fallaba, además los torneos semanales sirvieron como guía para el desarrollo general.

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

Como ya hemos apuntado en el primer punto, varias de las ideas fueron recogidas de distintos sitios de internet, a los mencionados anteriormente también añadir las siguientes referencias:

Lu, K. (2014) The Game Theory of Reversi

Sannidhanam, V. (2015) An Analysis of Heuristics in Othello [University of Washington]. https://courses.cs.washington.edu/courses/cse573/04au/Project/mini1/RUSSIA/Final_Paper.pdf

2.1.3. Descripción de la heurística enviada finalmente

El proceso de la heurística será, por tanto, recorreremos todas las posiciones comprobando que ficha se encuentra en esa posición para sumar los puntos asociados a esa posición al jugador correspondiente; en caso de que esa una esquina la medimos con un contador a parte y en caso de estar vacía comprobamos las fronteras, a estos datos le añadimos la movilidad y la coin parity y finalmente estableciendo unos pesos a cada valor calculamos el valor final de la heurística