Memoria Práctica 2

1. Documentación de minimax y la poda alfa-beta

**1.a. Detalles de implementación.**

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

Se ha realizado manualmente el algoritmo sobre el juego en árbol proporcionado en el enunciado de la práctica (simple\_game\_tree.py) y comprobado con la ejecución del script demo\_simple\_game.py, utilizando los jugadores “player1\_minimax\_alpha\_beta”, que será con quien comprobaremos el correcto funcionamiento de nuestra implementación del algoritmo minimax con poda alfa-beta, y “player2\_minimax\_alpha\_beta”, al que modificaremos el campo verbose a 0 para que no muestre información innecesaria.

Asimismo, ejecutaremos el código pero con los jugadores “player1\_minimax” y “player2\_minimax”, también modificando el campo verbose de este último a 0 por la misma razón que antes, para comprobar de otra manera más que el resultado final de nuestra implementación del algoritmo minimax con poda alfa-beta, en este caso no el proceso, es el correcto. Se puede comprobar en las capturas de pantalla adjuntas que el resultado es el esperado.

(aquí esquema del algoritmo alfa-beta manual, captura1, captura2\_1, captura2\_2)

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

Nuestra implementación del algoritmo minimax con poda alfa-beta se basa completamente en la implementación proporcionada por los profesores del algoritmo minimax.

Por tanto, esta nueva implementación se descompone en una función de inicialización \_\_init\_\_(), que devuelve el siguiente movimiento que debe realizar el jugador que use la estrategia minimax con poda alfa-beta en base a su heurística next\_move(), que delega en las dos últimas funciones: \_min\_value() y \_min\_value(). Estas funciones realizan la computación real de los valores que va tomando el algoritmo a lo largo de la búsqueda.

Las estructuras de datos son también las mismas que utilizan las funciones del algoritmo minimax ya proporcionado, a excepción de los valores que va tomando el mismo a lo largo de la búsqueda, ya que en el caso del algoritmo minimax base, éste solamente toma un único valor. Por contario, con la poda alfa-beta son necesarios dos valores intermedios (alfa y beta), que han sido considerados como una dupla - de dos valores -.

Para distintas funcionalidades necesitaremos también el uso de la librería math de python.

iii. Implementación.

Nuestra implementación es el paso a código del pseudocódigo, visto en las clases de teoría y proporcionado también en el enunciado de la práctica. Así pues, utilizaremos el término intervalo para referirnos a esa tupla de valores alfa y beta, ya que ésta debe ser entendida como el intervalo de valores que puede tomar el valor final del algoritmo.

En primer lugar, se inicializa el intervalo como (-inf, inf) y se llama a la función \_max\_value(), ya que interpretamos al primer jugador como “MAX”. Esta función consigue todos los valores beta de sus sucesores, y actualiza su propio valor alfa con el máximo de éste y los valores beta anteriores. De esta manera, se acota inferiormente el intervalo. Esos valores beta los consigue llamando a la función \_min\_value(), que consigue todos los valores alfa de sus sucesores, y actualiza su propio valor beta con el mínimo entre éste y los valores alfa anteriores. De esta manera se acota superiormente el intervalo.

Cabe destacar la naturaleza recursiva del algoritmo, ya que estos valores alfa, previamente mencionados, son conseguidos llamando a la función \_max\_value(). El caso base de la recursión es el siguiente: que haya terminado el juego ó que se haya alcanzado la profundidad de búsqueda máxima especificada en la inicialización de la estrategia.

En estos casos, las funciones \_max\_value() y \_min\_value() actualizan los valores alfa y beta respectivamente con el valor de la heurística para el estado que se está explorando.

(aquí código de MinimaxAlphaBetaStrategy)

**1.b. Eficiencia de la poda alfa-beta.**

i. Descripción completa del protocolo de evaluación.

Con los resultados del apartado 1.a.i. se puede apreciar que la poda efectivamente se está realizando. Aún así, hemos realizado otro test que esta vez demuestra la eficacia temporal del algoritmo minimax con poda alfa-beta sobre el algoritmo minimax base. Este se encuentra en el fichero adjunto tiempos.py.

Lo que este script realiza es inicializar distintos jugadores que utilizarán las estrategias minimax con poda alfa-beta y minimax base con la heurística 1 proporcionada (evaluation\_function\_dummy()) y una heurística propia usada en los torneos (evaluation\_function\_propia()). Con estos jugadores realiza partidos de reversi entre jugadores que usen la misma estrategia, de esta manera medimos el tiempo de duración del partido con la ayuda de la función time() de la librería time de python.

Estos datos de tiempos son los que usaremos en el siguiente apartado para contrastar la eficacia del algoritmo minimax con poda alfa-beta.

(aquí código de tiempos.py)

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

Cabe destacar que en vez de emplear la librería *timeit*, hemos hecho uso de la función time, ya que tiene una precisión de 10 dígitos decimales, que consideramos suficiente para nuestro estudio.

Tras ejecutar el script mencionado y descrito en el apartado anterior obtenemos los siguientes resultados:

(aquí Captura3 y tablas con esos datos)

Diagrama

Descripción generada automáticamente

iii. Medidas de mejora independientes del ordenador.

El algoritmo minimax con profundidad máxima puede entenderse como un algoritmo de búsqueda de profundidad limitada, por lo que su coste temporal es del orden de b elevado a l, donde b es el factor de ramificación del juego y l es la profundidad máxima.

Gracias a la poda alfa-beta conseguimos que este coste temporal se reduzca a algo del orden de b elevado a 3/4\*l, pudiendo en el mejor de los casos llegar hasta algo del orden de b elevado a l/2.

La independencia de los tiempos la logramos hallando el *ratio* entre los valores en segundos que nos calcula la ejecución del programa. De esta forma, conseguimos eliminar el factor particular del ordenador.

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

Como observamos en las tablas del apartado 1.b.ii., el hecho de añadir la poda alfa-beta al algoritmo minimax conlleva grandes mejoras en su coste temporal. Más específicamente, como mencionamos en el apartado anterior, pasamos de un coste temporal del orden de b elevado a l a uno del orden de b elevado a 3/4\*l.

Es fácil apreciar con estos resultados de mejora que dicha eficacia es más notable conforme aumentamos la profundidad máxima de búsqueda l. Dicho resultado lo podemos observar también en la tabla de datos anterior. Esto es debido a que, con la poda alfa-beta, somos capaces de detectar, más o menos rápidamente, ramas del árbol de juego que no tiene sentido explorar suponiendo que el rival (MIN) es eficaz, ya que obtenemos intervalos del tipo (alfa,beta) donde alfa es mayor, ó igual (en este caso habríamos encontrado un camino que lleva a un resultado igual a otro ya explorado, por lo que no interesa gastar tiempo explorándolo), que beta.

2. Documentación del diseño de la heurística

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

[1] Para el diseño de las heurísticas nos hemos basado fundamentalmente en dos fuentes. La primera de todas es una página web, que ha llevado a cabo la tarea de juntar información de diferentes fuentes, que incluye una explicación exhaustiva del juego y de distintas estrategias – movilidad, dominio de las esquinas, movilidad potencial, etc. -, tanto abstractas como concretas, e introduce las características esenciales de una función de evaluación para Reversi. No se incluye en formato APA porque el autor es anónimo y no es una publicación oficial:

<https://samsoft.org.uk/reversi/strategy.htm>

[2] La siguiente es una publicación de la Universidad de Washington, que reúne también las características que debe reunir una función de evaluación, y que serán posteriormente comentadas.

Sannidhanam V. & Annamalai, M. (-). An Analysis of Heuristics in Othello. University of Washington.

**2.b. Descripción del proceso de diseño****.**

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

El diseño de nuestras heurísticas se fundamenta en llevar a cabo evaluaciones que sigan de manera consistente una, o varias – pero no excesivas -, heurísticas de juego. De esta manera, consideramos que las heurísticas que mejor funcionan no son las que consisten en la suma de una gran cantidad de factores, ya que la tarea de integrar todos los valores numéricos derivados de las evaluaciones por separado podría dar lugar a contradicciones entre las evaluaciones, de forma que se cancelen. Este fenómeno es razonable, ya que cuantas más variables independientes se tenga, el resultado de la función resultará más impredecible, ya que la heurística, vista como función, toma valores en un dominio de mayor dimensión – y en consecuencia, con mayor número posible de valores -. Dicho esto, hemos tomado un máximo de tres estrategias, esencialmente diferentes, en cada heurística, por separado, teniendo siempre una heurística predominante. En una en particular, explicada en el apartado 2.c., aunque se tienen en cuenta distintas heurísticas, se dan en momentos de la partida diferentes.

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

La valoración de las funciones de evaluación la hemos llevado a cabo mediante torneos periódicos – del orden de dos o tres por semana - contra nuestras propias heurísticas y las básicas proporcionadas – dummy, random -. De esta forma, con un *pool* de cinco o seis heurísticas, presentábamos al torneo las tres primeras del último torneo antes de la entrega.

Asimismo, los torneos han servido para entender la eficacia de nuestras heurísticas, de forma que si quedaban en la mitad positiva del ranking, las mejorábamos. Por contrario, aquellas que quedaban por debajo no eran sujetos de mejora, sino reemplazadas por nuevas funciones de evaluación que se nos ocurrían durante el transcurso de las semanas.

En particular, tras el primer torneo, una de nuestras heurísticas presentadas logró alcanzar del orden de la décima posición. Así pues, centramos parte de los esfuerzos en mejorar esta heurística, evolucionándola a una versión más avanzada, que posteriormente sería seleccionada como una de las tres a entregar.

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”.

Una de nuestras heurísticas se basa fundamentalmente en una matriz que asigna valores a todas las posiciones del tablero 8x8, en función de lo ventajosa que resulta la colocación de una pieza en esa casilla. Dicha matriz se puede encontrar con distintos valores en múltiples páginas de internet, y que también fue enseñada en clase la segunda semana de la práctica.

Como hemos explicado en la sección 2.b.i, nuestro *modus operandi* se ha basado en la implementación de diferentes estrategias simples, que hemos realizado de forma autónoma, sin consultar material adicional. Por consecuencia, consideramos que nuestra principal fuente de información teórica ha sido la referencia [2].

No consideramos que haya tenido lugar ningún tipo de “comunicación privada”.

**2.c. Descripción de la heurística enviada finalmente.**

Heurística 1 (Solution 1) : **posicional\_dinamico\_caet**

La heurística de posicional dinámico se basa en, dada una matriz de dimensiones 8x8 variable que refleja el tablero, se suma el valor de todas las posiciones de la matriz en las que el tablero tenga una ficha del jugador. La función principal de la heurística es la siguiente:

def posicionalDinamico(juego, tablero, label):

La matriz y el acceso a sus valores se implementan en la siguiente función:

def valorPosicionDinamico8x8(clave, t) -> float:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 99 | -8 | 8 | 6 | 6 | 8 | -8 | 99 |
| -8 | -24 | -4 | -3 | -3 | -4 | -24 | -8 |
| 8 | -4 | 7 | 4 | 4 | 7 | -4 | 8 |
| 6 | -3 | 4 | 0 | 0 | 4 | -3 | 6 |
| 6 | -3 | 4 | 0 | 0 | 4 | -3 | 6 |
| 8 | -4 | 7 | 4 | 4 | 7 | -4 | 8 |
| -8 | -24 | -4 | -3 | -3 | -4 | -24 | -8 |
| 99 | -8 | 8 | 6 | 6 | 8 | -8 | 99 |

Nótese que la matriz presenta simetría en los cuatro cuadrantes que se generan desde el centro. La idea es que estos valores otorgan una medida de la utilidad, en forma de ventaja, que otorga al jugador en cuestión tener una ficha colocada en esa casilla del tablero. El principal inconveniente que presenta esta matriz es que es, en esencia, un promedio de dicha utilidad, ya que lo buena o mala que es la colocación de una ficha en una casilla es dependiente del estado del tablero en ese preciso momento. Por esta razón, hemos evolucionado la heurística de forma que los valores de la matriz sean dinámicos, o al menos dependientes del tablero. Así pues, definidas estas funciones, desarrollamos esta nueva matriz:















Todas las funciones están pensadas para que tengan los valores de interés en el intervalo [0,64], donde los valores que se encuentran en el mismo son, en realidad, la cantidad de fichas en el tablero, que es una medida de lo avanzada que esta la partida. En particular, cada una de las funciones toma un mayor valor en aquellos puntos temporales que consideramos que son más importantes para cada momento de la partida.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 99 | 0 | m(x) | l(x) | l(x) | m(x) | 0 | 99 |
| 0 | -24 | n(x) | s(x) | s(x) | n(x) | -24 | 0 |
| m(x) | n(x) | h(x) | g(x) | g(x) | h(x) | n(x) | m(x) |
| l(x) | s(x) | g(x) | f(x) | f(x) | g(x) | s(x) | l(x) |
| l(x) | s(x) | g(x) | f(x) | f(x) | g(x) | s(x) | l(x) |
| m(x) | n(x) | h(x) | g(x) | g(x) | h(x) | n(x) | m(x) |
| 0 | -24 | n(x) | s(x) | s(x) | n(x) | -24 | 0 |
| 99 | 0 | m(x) | l(x) | l(x) | m(x) | 0 | 99 |

Nótese que la matriz sigue siendo simétrica y que las esquinas mantienen aún cuatro valores estáticos. Esto se debe a que, además, hemos implementado una función a parte para la evaluación de la situación de los bordes, y a la que se le hace una llamada en *posicionalDinamico*(). La cabecera de la función es la siguiente:

def esquinaExpandida(tablero, label) -> int:

La lógica de esta función es que, evalúa las casillas adyacentes a las esquinas en el caso de que algún jugador tenga una esquina capturada. En específico, itera sobre todas las esquinas y tiene en cuenta una L desde la esquina, siguiendo esta configuración:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

Lo importante de este caso es que, si el jugador en cuestión tiene capturadas todas las casillas negras, entonces la casilla roja, que resulta ser, generalmente, una mala posición para poner una ficha, se vuelve de gran valor, ya que resulta ser estable, tomando el papel de la esquina adyacente. En ese caso, sumaríamos 123 al valor previamente calculado mediante la suma de los valores del tablero en la matriz de posicional dinámico. En la llamada a esta función por *posicionalDinamico()*, se agrega un peso de forma sinusoidal de amplitud 2, que da más importancia a la estabilidad de esta esquina expandida hacia la mitad avanzada de la partida.

Heurística 2 (Solution 2) : **movilidad\_y\_esquinas\_expandidas\_caet**

La siguiente heurística es la síntesis de tres estrategias esencialmente distintas.

i) Movilidad

Esta estrategia es simple, y consiste en priorizar jugadas que aumenten la cantidad de movimientos futuros. Se implementa en la siguiente función:

def mobility(tablero, label, juego) -> int:

Cabe destacar que, para equilibrar la magnitud de los valores devueltos, esta función se multiplica por 7, de forma que se equipare al valor del resto de funciones.

ii) Valor esquinas

Esta estrategia es, en esencia, una simplificación de la heurística posicional anteriormente descrita. Suma 99 por cada esquina capturada; resta -24 por las adyacentes diagonalmente a la esquina; resta -8 por las inmediatamente adyacentes a las esquinas. La matriz de valores sería la siguiente:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 99 | -8 |  |  |  |  | -8 | 99 |
| -8 | -24 |  |  |  |  | -24 | -8 |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
| -8 | -24 |  |  |  |  | -24 | -8 |
| 99 | -8 |  |  |  |  | -8 | 99 |

Esta función fue creada para poder aportar a la heurística una noción de la importancia de capturar las esquinas. La cabecera de la función es la siguiente:

def valorEsquinas(board, label):

iii) *EdgeFunction*, función de bordes

La lucha por los bordes comienza a ser importante a partir de cierto momento de la partida, ya que llevar a cabo un buen desarrollo en esta zona del tablero puede ser sinónimo de más oportunidades en el futuro. Por consecuente, esta función tiene como finalidad evaluar la situación de los bordes para un tablero dado.

La lógica es que, en el caso de que un jugador tenga una esquina, las casillas adyacentes tienen un valor positivo o negativo – en función de si la esquina es tuya o del enemigo -. De esta forma, primeramente, se construyen listas con las celdas a estudiar.

A partir de aquí, se distinguen dos bucles. El primero tiene como finalidad sumar puntos para cada casilla adyacente a una esquina que sea tuya. La suma se efectúa solo para aquellas casillas que se puedan conectar con la esquina en cuestión a través de casillas exclusivamente de tu color. De nuevo, esta idea orbita alrededor del concepto de estabilidad de fichas, ya que esas fichas no podrán ser capturadas por el enemigo. Un esquema del funcionamiento es el siguiente:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 8 |  |  |  |  |  |  |  |
| 10 |  |  |  |  |  |  |  |
| 12 |  |  |  |  |  |  |  |
| 14 |  |  |  |  |  |  |  |
| 16 |  |  |  |  |  |  |  |
| 18 |  |  |  |  |  |  |  |
| 20 |  |  |  |  |  |  |  |
|  | 20 | 18 | 16 | 14 | 12 | 10 | 8 |

En esta configuración, la esquina es tuya, y sumaría la puntuación indicada en cada casilla por cada ficha que conecte con la esquina a través de fichas únicamente de tu color, y únicamente a través de caminos que recorran el borde.

iv) Esquina expandida

Se trata de la función ya introducida en la primera heurística, que fue reutilizada para esta. En esencia, es muy parecida a la del valor de las esquinas. Esta es la razón por la que, aunque se hagan cuatro llamadas a funciones, solo se distingan tres estrategias diferentes.

Heurística 3 (Solution 3) : **mega\_mix\_caet**

En el caso de la tercera heurística, hemos diseñado una heurística que tenga en cuenta hasta cinco estrategias esencialmente diferentes – mediante seis funciones -, pero no de forma simultánea. Las nuevas estrategias diseñadas e implementadas para esta función son:

i) Potencial movilidad

Es una nueva interpretación de la movilidad, aunque esta vez con vista más al futuro. Casillas vacías alrededor de las fichas del enemigo se puede traducir en unos pocos turnos en oportunidades propias de captura de fichas enemigas. De esta forma, se itera sobre cada ficha del enemigo y se suma una unidad de valor - 1 - por cada casilla no ocupada adyacente - horizontal, vertical y diagonalmente -.

La estrategia es implementa en la siguiente función:

def potentialMobility(juego, tablero, yo) -> int:

ii) Máximas fichas

La siguiente estrategia mantiene el argumento que cuantas más fichas propias haya en el tablero, mejor. La cabecera de la función es la siguiente:

def maxFichas(tablero, label):

La función devuelve el número de fichas propias en el tablero.

iii) Movilidad anillo

La finalidad de esta estrategia es valorar un tablero en las siguientes posiciones:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

El grueso de la función es la definición de las listas que almacenan las posiciones en negro, y en rojo. La lógica de la estrategia es que, el anillo negro es de vital importancia para la lucha por los bordes, dad su adyacencia. En este caso, la función sencillamente itera sobre cada bloque horizontal o vertical de casillas negras y resta la multiplicación de las fichas tuyas que se encuentran en el anillo negro por la cantidad de fichas suyas en el anillo rojo. Esta afirmación se fundamenta en que, cuanta más fichas tenga el enemigo en las casillas, rojas, más vulnerables son las fichas propias en el anillo negro.

La función que implementa la estrategia es la siguiente:

def movilidadAnillo(tablero, label) -> int:

Además de las estrategias exclusivamente diseñadas para esta estrategia, se han reutilizado otras explicadas en otras secciones – otras heurísticas – de la sección 2.c. No obstante, no todas las heurísticas se evalúan en todas las etapas de la partida. Así pues, hemos distinguido cuatro fases distintas, a las que aplicamos las estrategias que consideramos más convenientes:

El parámetro agrupado en los intervalos es la cantidad de fichas, propias y enemigas, que se encuentran en el tablero. Es una forma de medir el avance de la partida.

|  |  |  |
| --- | --- | --- |
| FASE | INTERVALO | ESTRATEGIAS |
| 1 | [0,18] | Posicional Dinámico, Movilidad Anillo |
| 2 | [19,33] | Función de bordes, Movilidad, Potencial movilidad. |
| 3 | [34,50] | Posicional Dinámico, Movilidad |
| 4 | [51,64] | Movilidad, Máximo número de fichas |

Algunos valores de estas funciones se han ponderado para equiparar las magnitudes de los valores devueltos por las funciones.

Las decisiones de las estrategias empleadas siguen la lógica explicada a continuación.

* En la primera fase, es importante evitar colocar fichas erróneamente en el anillo definido en *movilidadAnillo()*. A su vez, la estrategia de posicional dinámico ofrece un buen comienzo de partida.
* En la segunda fase, la movilidad juega un papel importante, ya que priorizar tener un mayor número de posibilidades será recompensado en un futuro. Asimismo, da lugar el comienzo de la lucha por los bordes, ya que experimentalmente hemos determinado que las piezas comienzan a llegar a los extremos del tablero.
* En la tercera fase, posicional dinámico arroja buenos resultados, y se aporta también importancia a la movilidad, aunque no a la potencial, ya que huecos en fichas enemigas en este estado de la partida no es siempre generalmente positivo.
* En la última fase, se debe priorizar el número máximo de fichas. Priorizar la movilidad es, en esencia, priorizar el número máximo de fichas también. Se debe a que, si hay más movimientos disponibles, es más probable que haya un escenario con una cantidad considerada de fichas.