Proyecto Final

Conecta 4: Agente de Toma de Decisión.

Viernes 11 de noviembre de 2022.

—

Programación para Ciencias.

—

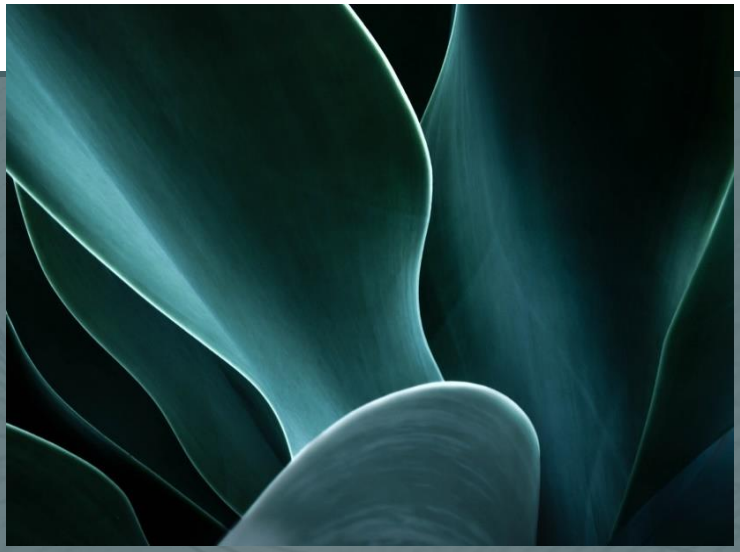
Jorge Sebastián Alejandro Chiu Molina Ramírez.

—

Matrícula: 222303457.

Introducción

El proyecto final se realizó con el propósito de desarrollar un agente de toma de decisión que fuera capaz de competir contra un usuario en el juego de Conecta 4.
Para dicho objetivo se separó el desarrollo del código en dos bloques: 1) emulación del entorno de Conecta 4 y sus reglas, y 2) creación del agente/bot.
El agente desarrollado e implementado en el proyecto funciona a partir de árboles y del algoritmo "minimax", restringiendo la profundidad del árbol y de revisión del algoritmo a 3 niveles. La razón de esto es para reducir la cantidad de instrucciones y operaciones a realizar por la computadora, así como reducir el tiempo de respuesta del agente. De igual manera, el código se desarrolló limitando su aplicación a Conecta 4; es decir, a un fichero de 6x7 donde se busca hacer cuatro en línea. De cualquier manera, tanto el agente como el entorno de juego se pueden adaptar para un distinto tamaño de fichero y cantidad de fichas a conectar en línea, aunque dicha adaptabilidad no es contemplada como característica del código de este proyecto.



Proyecto Final

Objetivos

La realización del proyecto final se llevó a cabo con el objetivo de desarrollar un agente de toma de decisión que fuera capaz de competir contra un usuario, humano o máquina, en el juego de conecta 4.
De este objetivo principal se desprendieron distintos objetivos secundarios necesarios para cumplir el primero, los cuales se enumeran y explican a continuación.

1. Emulación del entorno de Conecta 4 y sus reglas.

Para poder implementar el agente deseado era fundamental pensar en la forma en que se emularían las reglas que rigen el juego de Conecta 4, así como el entorno que permitiría darle seguimiento al desarrollo de la partida y la interacción del usuario, si es que este es humano, con el fichero.
El entorno del juego se implementó como una matriz 6x7 usando numpy, la cual es impresa cada que el estado de la partida cambia. Además, a lo largo del código siempre se trata al agente como el booleano *True*, y al usuario (retador) como el booleano *False*.
Entre las reglas por aplicar, una fundamental era la continuidad del juego, la cual se encuentra tal cual como una función del código. Entiéndase por continuidad del juego el criterio que permite determinar si se ha llegado a un empate (cuando ningún jugador ha ganado y el fichero se encuentra lleno), si existe un ganador o si se puede seguir jugando (no existe ganador y el fichero no está lleno). Dicho criterio se encuentra tal cual como una función dentro del código. Esta función recibe el fichero después del último movimiento realizado y el booleano relacionado al jugador que realizó dicho movimiento (*turno*), devolviendo un entero (pudiendo ser éste 0 para empate, 1 para victoria del agente, -1 para victoria del usuario o 2 para continuidad de la partida) el cual se termina guardando en una variable *status* que conserva el estado de continuidad del juego.
Otra regla importante y necesaria para asegurar el correcto funcionamiento del juego era la determinación de las posibles columnas en las cuales se pueden introducir fichas. Para dicho criterio se implementó una función *movval* que recibe la matriz del fichero de la partida en su último estado, y devuelve una tupla con la lista de las columnas disponibles para introducir una ficha y la cantidad de columnas disponibles. Este criterio es importante en la validación de los movimientos que introduzca el usuario, si es que es humano, y en la creación del árbol del agente.
Por último, una característica simple, pero importante, en el desarrollo del entorno es la sucesión continua e intercalada de introducción de fichas por parte de los jugadores. Dentro del código se puede identificar dicho proceso dentro de un ciclo *while,* el cual no se interrumpe mientras la variable *status* sea igual a 2. Para resolver el aspecto del intercalado de turnos, después de la introducción de la ficha por parte

del último jugador en turno y de la verificación de continuidad de la partida, se realiza el cambio del valor del booleano *turno*.

2. Determinación de criterio de evaluación de estado de partida.

Un aspecto indispensable en el desarrollo del agente de toma de decisión de este proyecto, así como de cualquier agente que haga uso del algoritmo *minimax*, es el criterio de evaluación de un estado determinado de la partida.
Dicho criterio permite reducir o interpretar el estado de la partida a un número real que indique qué jugador se encuentra en ventaja bajo las condiciones actuales del estado de partida. Este proceso de valoración de la partida implica un análisis del fichero bajo cierto conjunto de principios y reglas que permitan asignar un valor numérico a cada condición o característica presente en el estado de la partida, para finalmente obtener un único valor global que represente al estado, al cual se hará referencia como *puntaje del estado* o, simplemente, *puntaje*.
Cabe destacar que el criterio de evaluación, los principios y reglas que lo conforman, así como su eficacia, precisión en la valoración y utilidad, dependen y recaen en el proceso de análisis y apreciación de la partida que sea utilizado como base. En otras palabras, el criterio de evaluación está sujeto a la subjetividad del desarrollador.
A continuación, se detallan algunos conceptos, principios y reglas que son la base para el criterio de evaluación creado e implementado en el proyecto.
Los conceptos declarados se presentan para facilitar y agilizar la explicación de los principios y reglas utilizados.
Conceptos:

- Jugador en turno: Jugador que tiene concedido el derecho de realizar un movimiento.
- Jugador opuesto: Jugador que no tiene concedido el derecho de realizar un movimiento.
- 4-tupla: Usaremos este concepto para hacer referencia a cualquier conjunto de 4 fichas del mismo jugador en turno inmediatamente contiguas en línea (horizontal, vertical, o diagonal).
- 3-tupla favorable: Cualquier conjunto de 3 fichas del mismo jugador en turno y una casilla vacía inmediatamente contiguas en línea (horizontal, vertical, o diagonal), en cualquier orden; es decir, la casilla vacía se puede encontrar intercalada en cualquier posición de dicha línea, y las fichas situadas en las demás posiciones.
- 3-tupla adversa: Cualquier conjunto de 3 fichas del mismo jugador opuesto y una casilla vacía inmediatamente contiguas en línea (horizontal, vertical, o diagonal), en cualquier orden.
- Columna disponible (Cd): Columna que tiene por lo menos una casilla disponible para insertar una ficha.
- Casilla inmediata (Ci): Casilla donde se posicionaría una ficha si se insertara en determinada columna disponible (Cd). En otras palabras, casilla vacía más baja/profunda perteneciente a determinada columna disponible (Cd).

- Casilla Ancla (Ca): Casilla que se toma como referencia para la búsqueda por direcciones, y que puede tomar cualquier k-posición ($1 \leq k \leq 4$) dentro de las 4 posiciones que conforman a toda n-tupla.
- Casilla Usada (Cu): Casilla donde se insertó la ficha para originar determinado estado alternativo.
- Estado Originador: Estado de partida que se usa como base para originar un estado alternativo.
- Estado Alternativo: Estado de partida originado en caso de insertar una ficha del jugador en turno en alguna de las columnas disponibles.

Una vez definidos los conceptos anteriores, procedemos a indicar las puntuaciones determinadas para cada tipo de tupla (n-tupla) definida.
Cabe destacar, y para evitar confusiones más adelante, que durante el proceso de creación del árbol del agente (proceso que se detallará más adelante) se realiza reiteradamente este proceso de evaluación para cada uno de los nodos del árbol, donde los papeles de *jugador en turno* y *jugador opuesto* se intercambian cada nivel, recordando que cada uno de los jugadores se representan con un booleano.
Puntuación:

- 4-tupla: +4000 si el *jugador en turno* es el agente (*True*), -4000 si el *jugador en turno* es el usuario (*False*).
- 3-tupla favorable: +1 si *jugador en turno* es *True*, -1 si *jugador en turno* es *False*.
- 3-tupla adversa: -100 si *jugador en turno* es *True*, +100 si *jugador en turno* es *False*.

Por último, presentamos los principios de orden y eliminación de redundancia utilizados.
Principios de Orden
Entiéndase por principio de orden toda regla que determina la secuencia de evaluación respecto a alguna condición específica.
En este caso, el único principio de orden utilizado es el *principio de victoria*. Esto hace referencia a que el orden en que se evalúa el estado de la partida es el siguiente:

a) Búsqueda de 4-tupla.
b) Búsqueda de 3-tuplas favorables.
c) Búsqueda de 3-tuplas adversas.

El orden de los dos últimos puntos puede ser intercambiado sin mayor relevancia. La razón de priorizar la búsqueda de una 4-tupla está relacionado con el hecho de que el jugador en turno siempre buscará ganar la partida en el próximo movimiento a realizar (es decir, durante su actual turno), con lo cual cualquier análisis del estado de la partida posterior a encontrar una 4-tupla se vuelve redundante e inútil (esto está vinculado con un principio de eliminación de redundancia que será abordado más adelante).
Además, el *principio de victoria* está directamente relacionado con la disminución de operaciones a realizar por la computadora y el tiempo de respuesta del agente,

sirviendo como predecesor y dando pie al *principio de gane* para descartar todo análisis posterior si se llegase a encontrar una 4-tupla.

Principios de Eliminación de Redundancia

Entiéndase por principio de eliminación de redundancia toda regla que permita evitar y descartar parte del proceso de análisis que no aporte ninguna ventaja, o incluso sea adverso, al criterio de evaluación del estado de la partida.

El primer principio de eliminación de redundancia es el *principio de gane*, el cual descarta todo análisis posterior si se llegase a encontrar una 4-tupla.

Otro principio de eliminación de redundancia es el principio de dirección de búsqueda, el cual se realiza en base a dos condiciones: posición en matriz e imposibilidad hacia arriba.

El primero de ellos hace referencia a la posibilidad de buscar una n-tupla con base en la fila y columna de la entrada de la matriz de la *casilla ancla*. Es decir, si se tomara como casilla ancla una casilla de la última columna de la matriz, en el sentido horizontal se podría buscar una n-tupla hacia la izquierda, pero no hacia la derecha, pues no habría casillas disponibles para checar hacia la derecha de la casilla ancla.

Tomando esto en cuenta, y sabiendo que la casilla ancla puede tomar cualquier k-posición de la n-tupla, este principio aplica para cuando la casilla ancla se toma en cualquier k-posición, siendo necesario asegurar que existan casillas suficientes en ambas direcciones del sentido determinado. Un ejemplo de esto es el siguiente:

Sea diagonal derecha hacia abajo el sentido determinado de entre los cuatro posibles (horizontal/derecha, vertical/abajo, diagonal derecha hacia abajo y diagonal izquierda hacia abajo), si la casilla ancla es la 2-posición de la n-tupla, será necesario asegurar que exista una casilla en la dirección negativa (izquierda hacia arriba) y dos en la dirección positiva (derecha hacia abajo).

La segunda condición (imposibilidad hacia arriba) hace referencia a la imposibilidad de encontrar cualquier tipo de n-tupla en dicha dirección si la casilla ancla es cualquier casilla inmediata.

El último principio de eliminación de redundancia es el *principio de multiplicidad*. Hace referencia a que para cualquier casilla ancla, si se encuentra algún tipo de n-tupla en cualquier sentido, no es necesario seguir buscando más n-tuplas del mismo tipo en ningún otro sentido en la misma casilla ancla. La razón de la existencia de este principio recae en distintos puntos:

- Para una 4-tupla, como se había mencionado con anterioridad, una vez encontrada una tupla de este tipo, no hay utilidad en seguir buscando más tuplas de este tipo en cualquier otro sentido o cualquier otro tipo de tupla.
- Para 3-tuplas favorables o adversas, el motivo de este principio surge de la consideración del papel que juega el jugador opuesto o en turno, respectivamente, en la eliminación de esa n-tupla en su siguiente turno respectivo.

3. Desarrollo del método de generación del árbol para toma de decisión del agente.

El proceso de generación del árbol implementa distintos aspectos importantes del proyecto. Sabiendo que el árbol se genera a partir de los posibles *estados alternativos* después de cada uno de los turnos de los jugadores y con el objetivo de aplicarle el algoritmo *minimax*, entonces se considera a este como un árbol de nodos, donde cada uno de ellos es un *estado alternativo* representado por una tupla de datos: el movimiento creador y el puntaje. Hablaremos de movimiento creador a la columna disponible en la cual se debe insertar la ficha del jugador en turno para generar dicho estado alternativo.

Como se explicó en un inicio, para que el tiempo de respuesta del agente fuera tolerable, se determinó que la profundidad máxima del árbol fuera de 3 niveles, sin tomar en cuenta al nivel cero (la raíz del árbol). En estricto sentido, el árbol es de profundidad 4, pero como a la raíz no se le considera un nodo sujeto a análisis por parte del algoritmo *minimax*, tampoco se le considera como nivel sujeto a análisis al ser el único nodo de su nivel. De este modo, el árbol siempre tendrá máximo 3 niveles además del nivel cero (raíz), si es que dicha cantidad de niveles se llegara a alcanzar al momento de su generación. Un detalle importante es la cantidad máxima de hijos que un nodo puede tener. En este caso, como la cantidad máxima de movimientos válidos (columnas disponibles) es 7, éste es la cantidad máxima de hijos que un nodo puede tener.

Procedemos a explicar el método de generación del árbol:

El árbol se genera a partir de la raíz, la cual al no estar sujeta a análisis y para efectos prácticos del código se le asigna un movimiento no válido (-1) y un puntaje de 0 dentro de su tupla.

Una vez creada la raíz, se procede a generar el resto del árbol. Para ello se implementó la función *generarArbol*, la cual se encarga de insertar los nodos del árbol de manera sistemática, y que permitiera facilitar la ideación del funcionamiento del algoritmo *minimax* adaptado a dicho tipo de árbol.

En este caso, uno de los criterios de introducción de los nodos es el orden por nivel. Para esto, siempre se introducen los nodos de izquierda a derecha, dejando todos aquellos espacios para hijos inexistentes a la derecha.

Como el algoritmo *minimax* se basa en tomar de manera intercalada el valor máximo/mínimo de un nivel, y el valor mínimo/máximo del siguiente, de igual manera se generan los nodos de nuestro árbol.

El primer nivel no raíz del árbol representa a los *estados alternativos* posibles durante el turno del agente (*True*) originados a partir del estado actual de la partida como *estado originador*, con lo cual se evalúan y puntúan a dichos estados con el agente (*True*) como *jugador en turno* y al usuario (*False*) como *jugador opuesto*.

El segundo nivel no raíz del árbol representa a los *estados alternativos* posibles durante el turno del usuario (*False*) originados a partir de alguno de los *estados alternativos* del primer nivel no raíz como *estado originador*, con lo cual se evalúan y puntúan a dichos estados con el usuario (*False*) como *jugador en turno* y al agente (*True*) como *jugador opuesto*.

En general, sea cual sea la profundidad del árbol, los niveles no raíz impares corresponden a *estados alternativos* donde el *jugador en turno* es el agente (*True*),

y los niveles no raíz pares a *estados alternativos* donde el *jugador en turno* es el usuario (*False*).
De esta manera, el árbol generado contiene todos los *estados alternativos* posibles hasta 3 turnos adelante.
Parte de la generación del árbol es la evaluación de cada *estado alternativo*, proceso que se explica a continuación y el cual consiste en determinar el *puntaje* a partir del criterio de evaluación antes abordado.
Cabe resaltar que en este proceso se toman en cuenta los conceptos definidos, las puntuaciones y los principios de orden y de eliminación de redundancia establecidos en el criterio de evaluación. Es importante hacer notar la diferencia, pues el criterio de evaluación consiste en el conjunto de reglas y principios que permiten evaluar un *estado*, mientras que el proceso de evaluación consiste en la aplicación de dicho criterio para la obtención del *puntaje del estado*.

4. Proceso de evaluación de *estado*

Este proceso es parte del proceso de generación del árbol; es decir, es un proceso anidado dentro de otro proceso, el cual es necesario para la realización de aquel que lo contiene. Éste se encuentra en el código como la función *evaluar*, y es llamada dentro de la función *generarArbol*.
La función *evaluar* se aplica a *estados alternativos*, es decir, no se aplica al *estado de partida actual* ni a ningún *estado de partida anterior*. En otras palabras, se aplica exclusivamente a *estados de partida* que podrían originarse si se realizara determinado movimiento. Esto debido a que se desea saber que tan favorable o adverso sería determinado *estado alternativo*, y permitir al agente tomar una decisión con base en ello.
La función recibe la matriz de un *estado alternativo*, junto con el booleano del *jugador en turno*, y la fila y columna de la *Casilla Usada* por el *jugador en turno*. Por su parte, la función devuelve el puntaje del *estado alternativo*.
El *puntaje* base del *estado* (antes de la evaluación) se inicializa como cero.
En primer lugar, el *principio de victoria* es aplicado, siendo la búsqueda de una 4-tupla lo primero en realizarse por parte de la función. La búsqueda de la 4-tupla se realiza únicamente tomando como *Casilla Ancla* a la *Casilla Usada*. La razón de esto es debido a que si se generó el *estado alternativo* que se está evaluando es porque el estado de continuidad del juego del *estado originador* era distinto de 1 y -1, lo cual implica que antes de generarse el *estado alternativo* no existía ninguna 4-tupla en el *estado originador*. Además, el *principio de dirección de búsqueda* y el *principio de* multiplicidad son aplicados.
Una vez que se terminó de realizar la búsqueda de la 4-tupla, si dicha n-tupla se encontró, rigiéndose por el *principio de gane* la función devuelve la puntuación de la 4-tupla como *puntaje* del *estado alternativo* (4000 si *jugador en turno* es *True*, -4000 en caso contrario). Si no se encontró una 4-tupla, se procede a buscar 3-tuplas favorables y adversas.
Para ambas búsquedas (de 3-tupla favorable y adversa) se toma como Casilla Ancla a cada una de las Casillas Inmediatas en el *estado alternativo* que se está evaluando.

De igual manera que en el caso de la 4-tupla, el *principio de dirección de búsqueda* y el *principio de multiplicidad* son aplicados para cada una de las Casillas Inmediatas que se toman como Casillas Anclas en ambas búsquedas.
Finalmente, por cada *3-tupla favorable* encontrada en el *estado alternativo* se suma la puntuación correspondiente al tipo de tupla y al booleano del *jugador en turno*. De manera similar, por cada *3-tupla adversa* encontrada en el *estado alternativo* se suma la puntuación correspondiente al tipo de tupla y al booleano del *jugador en turno*.
Una vez realizado esto, se devuelve la suma obtenida como *puntaje* del *estado alternativo*.

5. Aplicación del algoritmo *minimax*.

Por último, solo se menciona que el algoritmo *minimax* adaptado a este tipo de árbol es utilizado por el agente para obtener el mejor movimiento posible y realizar la toma de decisión.

Conclusiones

Al plantear el proyecto y su objetivo no se tenían contemplados algunos aspectos y debilidades en el funcionamiento del agente. Un ejemplo de esto es la tendencia a la derecha que tiene el agente en la selección de la columna al momento de realizar su movimiento, en caso de que tenga dos posibles movimientos con el mismo puntaje máximo. Se implementó un par de líneas de código con el objetivo de hacer aleatoria la selección entre dos movimientos con el mismo puntaje máximo. Sin embargo, se encontró que este proceso también resultaba en tendencia a la derecha con los siguientes porcentajes para cada una de las columnas (siendo este un caso especial en el que todas las columnas tienen el mismo puntaje, y por lo tanto, teniendo todas el puntaje máximo, pero que se puede generalizar para cualquier otro caso): 7ma columna (50% de las veces), 6ta columna (25% de las veces), 5ta columna (12.5%), 4ta (6.25%), 3ra (3.125%), 2da (1.5625%), 1ra (1.5625%).

Otro aspecto que no se contempló en el funcionamiento del agente, pero que no termina por afectar su competitividad, es el hecho de que cuando el agente cae en Zugzwang (aunque en este caso, no empeora su situación, si no que simplemente cualquier movimiento que realice no evita que pierda en el siguiente turno) ni siquiera realiza el mínimo esfuerzo por eliminar al menos una de 3-tuplas adversas presentes en el estado de la partida.

Entre los aspectos que se tenían contemplados, pero que costó definir su manera de implementar fue la optimización del código para reducir la cantidad de operaciones a realizar por la máquina y disminuir el tiempo de respuesta del agente. Los principios de orden y de eliminación de redundancia fueron parte de la solución a dicho problema, pues permiten descartar una gran parte del proceso de evaluación y generación del árbol que termina siendo inútil, e incluso absurdo, en muchos de los casos.

Finalmente, se realizaron adaptaciones al código, pero que no se tomaron en cuenta como parte del proyecto, para analizar el funcionamiento, tiempo de respuesta y porcentaje de victoria del agente cuando la profundidad de generación del árbol y de análisis del algoritmo *minimax* se aumentaba a 5 o 7 niveles. Como era de esperar, el tiempo de respuesta se veía afectado, aumentando conforme aumentaba la cantidad de niveles. Sin embargo, al enfrentar agentes con diferente nivel de profundidad de análisis, aquel con mayor nivel de profundidad presentaba ventaja en la probabilidad de victoria.

Apéndice con el Código Implementado

```
def right (i,j,l,mat,find):
    v=True
    c=1
    while v and c<l:
        if mat[i][j+c]!=find[c]:
            v=False
        c+=1
    return v

def down (i,j,l,mat,find):
    v=True
    c=1
    while v and c<l:
        if mat[i+c][j]!=find[c]:
            v=False
        c+=1
    return v

def downright (i,j,l,mat,find):
    v=True
    c=1
    while v and c<l:
        if mat[i+c][j+c]!=find[c]:
            v=False
        c+=1
    return v

def downleft (i,j,l,mat,find):
    v=True
    c=1
    while v and c<l:
        if mat[i+c][j-c]!=find[c]:
            v=False
        c+=1
    return v

def gane (n,m,mat,find,l):
    for i in range(n):
        for j in range(m):
            if mat[i][j]==find[0]:
                if j<=m-l:
                    d=right(i,j,l,mat,find)
```

```
                if d:
                    return True
            if i<=n-l:
                d=down(i,j,l,mat,find)
                if d:
                    return True
            if i<=n-l and j<=m-l:
                d=downright(i,j,l,mat,find)
                if d:
                    return True
            if i<=n-l and j>=l-1:
                d=downleft(i,j,l,mat,find)
                if d:
                    return True
    return False

def abajo(tab,i,j):
    lista=[]
    for n in range(4):
        lista.append(tab[i+n][j])
    return lista

def derecha(tab,i,j):
    lista=[]
    for n in range(4):
        lista.append(tab[i][j+n])
    return lista

def abajoDer(tab,i,j):
    lista=[]
    for n in range(4):
        lista.append(tab[i+n][j+n])
    return lista

def abajoIzq(tab,i,j):
    lista=[]
    for n in range(4):
        lista.append(tab[i+n][j-n])
    return lista

class raiz:

    def __init__(self,m=-1,p=0):
        self.p=p
```

```
        self.m=m
        self.ha=None
        self.hb=None
        self.hc=None
        self.hd=None
        self.he=None
        self.hf=None
        self.hg=None

    def hijos(self):
        if self.hg is not None:
            return 7
        if self.hf is not None:
            return 6
        if self.he is not None:
            return 5
        if self.hd is not None:
            return 4
        if self.hc is not None:
            return 3
        if self.hb is not None:
            return 2
        if self.ha is not None:
            return 1
        return 0

    def minimax(self,tipo,prof,ctrl=False):
        if self.hijos()==0 or prof==0:
            return self.m,self.p
        else:
            mov=None
            k=-10000 if tipo else 10000
            opc=[]
            for i in range(self.hijos()):
                if i==0:
                    opc.append(self.ha.minimax(not tipo, prof-1))
                elif i==1:
                    opc.append(self.hb.minimax(not tipo, prof-1))
                elif i==2:
                    opc.append(self.hc.minimax(not tipo, prof-1))
                elif i==3:
                    opc.append(self.hd.minimax(not tipo, prof-1))
                elif i==4:
                    opc.append(self.he.minimax(not tipo, prof-1))
```

```
            elif i==5:
                opc.append(self.hf.minimax(not tipo, prof-1))
            elif i==6:
                opc.append(self.hg.minimax(not tipo, prof-1))
        if tipo:
            for i in range(self.hijos()):
                v,w=opc[i]
                if w==4000 and ctrl:
                    if i==0:
                        if self.ha.hijos()==0:
                            return v,w
                    elif i==1:
                        if self.hb.hijos()==0:
                            return v,w
                    elif i==2:
                        if self.hc.hijos()==0:
                            return v,w
                    elif i==3:
                        if self.hd.hijos()==0:
                            return v,w
                    elif i==4:
                        if self.he.hijos()==0:
                            return v,w
                    elif i==5:
                        if self.hf.hijos()==0:
                            return v,w
                    elif i==6:
                        if self.hg.hijos()==0:
                            return v,w
                if (w>k):
                    mov=v
                    k=w
                elif (w==k):
                    import random
                    mov,k=random.choice([(mov,k),(v,w)])

        else:
            for i in range(self.hijos()):
                v,w=opc[i]
                if (w<k):
                    mov=v
                    k=w
                elif (w==k):
                    import random
```

```
                    mov,k=random.choice([(mov,k),(v,w)])
            if self.m==-1:
                return mov,k
            else:
                return self.m,k


def movval(tab):
    lista=[]
    c=0
    for i in range (7):
        if tab[0][i]==0:
            lista.append(i)
            c+=1
    return lista,c

def validar(lista,u):
    return False if u in lista else True

def insertar(fich,col,player,v=False):
    for i in range (5,-1,-1):
        if fich[i][col]==0:
            if player:
                fich[i][col]=2
                return fich if v else (fich,i)
            else:
                fich[i][col]=1
                return fich if v else (fich,i)

def continuidad(tab,player):
    if player:
        if gane(6,7,tab,[2,2,2,2],4):
            return 1
        else:
            for i in range (7):
                if tab[0][i]==0:
                    return 2
            return 0
    else:
        if gane(6,7,tab,[1,1,1,1],4):
            return -1
        else:
            for i in range (7):
                if tab[0][i]==0:
```

```
                return 2
        return 0

def mostrarTab(tab):
    print('\n\n')
    for i in range(6):
        for j in range(7):
            print(tab[i][j],end='\t')
        print('\n')

def contadorV(lista,player):
    n=2 if player else 1
    for i in range(4):
        if lista[i]!=n:
            return False
    return True

def contador3T(lista,player):
    y,v=0,0
    n=2 if player else 1
    for i in range(4):
        if lista[i]==n:
            y+=1
        elif lista[i]==0:
            v+=1
    return True if (y==3 and v==1) else False

def conecta4(tab,mv,fil,player):
    if fil<=2:
        sec=[2,2,2,2] if player else [1,1,1,1]
        if down(fil,mv,4,tab,sec):
            return True
    for d in range(4):
        if mv>=0+d and mv<=3+d:
            if contadorV(derecha(tab,fil,mv-d),player):
                return True
        if (fil>=0+d and fil<=2+d) and (mv>=0+d and mv<=3+d):
            if contadorV(abajoDer(tab,fil-d,mv-d),player):
                return True
        if (fil>=0+d and fil<=2+d) and (mv>=3-d and mv<=6-d):
            if contadorV(abajoIzq(tab,fil-d,mv+d),player):
                return True
    return False
```

```
def conecta3T(tab,i,j,player):
    if i<=2:
        if contador3T(abajo(tab,i,j),player):
            return True
    for d in range(4):
        if j>=0+d and j<=3+d:
            if contador3T(derecha(tab,i,j-d),player):
                return True
        if (i>=0+d and i<=2+d) and (j>=0+d and j<=3+d):
            if contador3T(abajoDer(tab,i-d,j-d),player):
                return True
        if (i>=0+d and i<=2+d) and (j>=3-d and j<=6-d):
            if contador3T(abajoIzq(tab,i-d,j+d),player):
                return True
    return False

def filCor(tab,listC,c):
    listF=[]
    for i in range(c):
        for j in range(5,-1,-1):
            if tab[j][listC[i]]==0:
                listF.append(j)
                break
    return listF

def borrar(copy,i,j):
    copy[i][j]=0
    return copy

def evaluar(tab,mv,fil,player):
    if conecta4(tab,mv,fil,player):
        return 4000 if player else -4000
    listC,c=movval(tab)
    listF=filCor(tab,listC,c)
    p=0
    for n in range(c):
        i,j=listF[n],listC[n]
        if conecta3T(tab,i,j,player):
            p+=1 if player else -1
        if conecta3T(tab,i,j,not player):
            p-=100 if player else -100
    return p

def generarArbol(tab,r,player,n):
```

```
    tope=4000 if player else -4000
    listm,c=movval(tab)
    for i in range(c):
        x=listm[i]
        rep,fil=insertar(tab,x,player)
        ptj=evaluar(rep,x,fil,player)
        if i==0:
            r.ha=raiz(x,ptj)
            if ptj!=tope and n>1:
                generarArbol(rep,r.ha,not player,n-1)
        elif i==1:
            r.hb=raiz(x,ptj)
            if ptj!=tope and n>1:
                generarArbol(rep,r.hb,not player,n-1)
        elif i==2:
            r.hc=raiz(x,ptj)
            if ptj!=tope and n>1:
                generarArbol(rep,r.hc,not player,n-1)
        elif i==3:
            r.hd=raiz(x,ptj)
            if ptj!=tope and n>1:
                generarArbol(rep,r.hd,not player,n-1)
        elif i==4:
            r.he=raiz(x,ptj)
            if ptj!=tope and n>1:
                generarArbol(rep,r.he,not player,n-1)
        elif i==5:
            r.hf=raiz(x,ptj)
            if ptj!=tope and n>1:
                generarArbol(rep,r.hf,not player,n-1)
        elif i==6:
            r.hg=raiz(x,ptj)
            if ptj!=tope and n>1:
                generarArbol(rep,r.hg,not player,n-1)
        borrar(rep,fil,x)

def centinela(tab):
    r=raiz()
    generarArbol(tab,r,True,3)
    m,p=r.minimax(True,3,True)
    return m

import numpy as np
tab=np.reshape([i*0 for i in range (42)], (6,7))
```

```
status=2
turno=False
mostrarTab(tab)
while status==2:
    listam,c=movval(tab)
    if turno:
        b=centinela(tab)
        tab=insertar(tab,b,turno,v=True)
    else:
        u=int(input('Ingrese su movimiento:'))-1
        while validar(listam,u):
            u=int(input("Movimiento no válido. Por favor, vuelva a ingresar su
movimiento:"))-1
        tab=insertar(tab,u,turno,v=True)
    status=continuidad(tab,turno)
    if turno:
        turno=False
    else:
        turno=True
    mostrarTab(tab)
if status==1:
    print("Lo sentimos. La máquina te ha derrotado.")
elif status==-1:
    print("¡Felicidades, has ganado!")
elif status==0:
    print('La partida ha terminado en tablas. Bien jugado, y suerte para la
próxima.')
```