Proyecto 1: Dominó

Sebastián Yamil Castellanos Gómez
Oscar Martinez Sosa
Sara Visoso Gunther

Supuestos

- Nosotros somos el jugador 1
- Queremos robar la menor cantidad de veces posible para ganar
- Queremos hacer que nuestro contrincante robe
- Queremos favorecer los escenarios en que ‘bloqueamos’ el juego a nuestro favor.

Partes del código:

- Clase ficha
- Clase dominó
- Clase nodo minimax
- Clase minimax dominó
- Prueba dominó

Clase Dominó

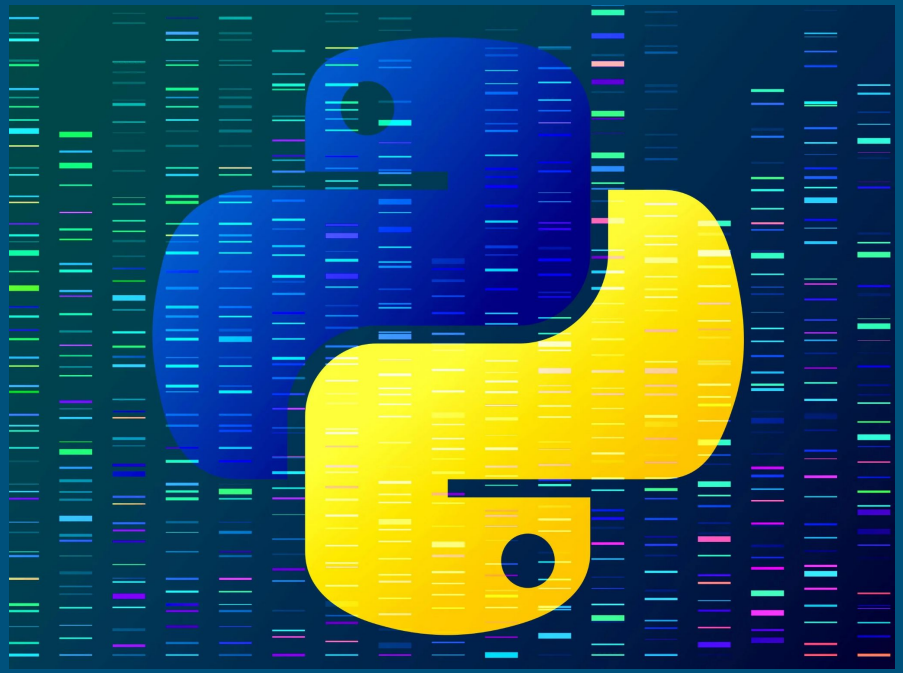
```
class Domino:
    def __init__(self):
        self.cont1 = 0
        self.cont2 = 0
        self.fichas = []
        self.fichasjugador1 = []
        self.extremos = []

        self.fichas_usadas = []
        self.jugador = 0
```

Funciones importantes de la clase:

- Inicializar
- Robar ficha (J1 y J2)
- Robar inicial (J1)
- Modificar fichas (J1)
- Usar ficha
- Ficha jugable
- Usabilidad (J1)
- Usabilidad (J2)
- Bloquea
- Número de fichas específicas (J1 y pozo)
- mulas (J1)

Librería Copy

- Permite crear copias de distintos objetos en Python.
- Nos ayuda a crear dos objetos iguales independientes del otro.
- El método copy.deepcopy() retorna una copia profunda de los objetos. Esto es útil al trabajar con objetos que contienen otros objetos, ya que crea una copia del objeto general y de todos los que contiene.



Clase nodo minimax

```python
class nodo_minimax:
    def __init__(self, estado_domino):
        self.estado_domino = estado_domino  # Instancia de la clase Domino que representa el es
        self.hijos = []  # Lista de nodos hijos
        self.valor_utilidad = 0  # El valor de utilidad asignado a este nodo
        self.bloquea = 0  # Valor entero que indica si este nodo bloquea el juego
        self.probabilidad = 1.0  # Valor de probabilidad asociado al nodo
        self.dicc_pasa = {0: False, 1: False, 2: False, 3: False, 4: False, 5: False, 6: False}
        self.jugadas_inteligentes=0 #Jugadas que determinamos son favorables estratégicamente
        self.mulas_estrategicas=0  #Valor positivo o negativo para retener una mula
```

Funciones importantes:

- Valor utilidad final
- Expandir
- Diferencia ponderada
- Bloqueo ponderado
- Probabilidad número
- Probabilidad Respuesta
- Evaluar
- Actualizar dicc pasa
- Roba todo y pasa

Funciones que apoyan a la función heurística

```python
def diferencia_ponderada(self,cont1, cont2):
 diferencia_fichas = cont2 - cont1

 if diferencia_fichas < -2:
    resta_ponderada = -0.5
 elif diferencia_fichas == -1:
    resta_ponderada = 0.35
 elif diferencia_fichas == -2:
    resta_ponderada = 0
 elif diferencia_fichas == 0:
    resta_ponderada = 0.5
 elif diferencia_fichas == 1:
    resta_ponderada = 0.88
 elif diferencia_fichas == 2:
    resta_ponderada = 1
 else:
    resta_ponderada = 1.1

 return resta_ponderada
```

```python
def bloqueo_ponderado(self):
    if(self.estado_domino.bloquea()):
        punt_bloqueo_j1=len(self.estado_domino.usabilidad_j1())
        punt_bloqueo_j2=len(self.estado_domino.usabilidad_j2())
        if(punt_bloqueo_j1==0):
            return -0.1
        if(punt_bloqueo_j1==1 and punt_bloqueo_j2>=3):
            return 0.3
        if(punt_bloqueo_j1==1 and punt_bloqueo_j2<3):
           return 0.65

        elif(punt_bloqueo_j1==2 and punt_bloqueo_j2<4):
            return 0.85
        elif(punt_bloqueo_j1==2 and punt_bloqueo_j2>=4):
            return 0.6
        elif(punt_bloqueo_j1>2):
            return 1.05

    else:
        return 0
```

Funciones que apoyan a la función heurística

```python
#Aquí obtendremos la probabilidad de que se juegue una ficha con un num esp
def probabilidad_numero(self,numero):
    #Numero de fichas con el num especificado
    num_fichas_disp= len(self.estado_domino.num_fichas_esp_pozo(numero))
    #Numero de fichas del mismo numero que nosotros tenemos
    num_fichas_j1=len(self.estado_domino.num_fichas_esp_j1(numero))
    #Numero de fichas que tiene el rival en total
    num_fichas_rival=self.estado_domino.cont2
    #Numero de fichas que quedan en el pozo
    num_fichas_pozo=len(self.estado_domino.fichas) - num_fichas_rival

    w= num_fichas_disp
    x= num_fichas_rival+num_fichas_pozo
    y= 7- num_fichas_j1
    z= num_fichas_rival
    proba=hypergeom.cdf(w,x,y,z)-hypergeom.pmf(0,x,y,z)
    #Restamos la función de masa de f(0) porque queremos medir
    #la probabilidad de que
    #el rival tenga una o más de las fichas con ese numero!

    return proba
```

```python
#Aquí calculamos la probabilidad de respuesta del adversario
def probabilidad_respuesta(self,ficha):
    num1=ficha.numero1
    num2=ficha.numero2
    if(len(self.estado_domino.fichas_usadas)!=0):

        num3=self.estado_domino.extremos[0]
        num4=self.estado_domino.extremos[1]
        if(num1==num3):
            return self.probabilidad_numero(num2)
        elif(num2==num3):
            return self.probabilidad_numero(num1)
        elif(num1==num4):
            return self.probabilidad_numero(num2)
        else:
            return self.probabilidad_numero(num1)
    else:
        prob=max(self.probabilidad_numero(num1),
                 self.probabilidad_numero(num2))
        return prob
```

```python
def expandir(self):
    #La función expandir nos permite hacer el arbol de busqueda
    if self.estado_domino.jugador == 1:
        usables = self.estado_domino.usabilidad_j1()

    else:
        usables = self.estado_domino.usabilidad_j2()
        self.roba_todo_y_pasa(usables)
        #Queremos favorecer escenarios clave

    # Crear nodos hijos para cada movimiento jugable y agregarlos a la lista de hijos
    for ficha in usables:


        nuevo_estado = copy.deepcopy(self.estado_domino)
        nuevo_nodo = nodo_minimax(nuevo_estado)
        #La incertidumbre sobre la siguiente jugada solo existe cuando tiramos nosotros, por eso
        #solo se modifica la probabilidad si tira el jugador2
        if(nuevo_nodo.estado_domino.jugador==1):
            nuevo_nodo.set_probabilidad(self.get_probabilidad()*nuevo_nodo.probabilidad_respuesta(ficha))

        nuevo_nodo.estado_domino.usarficha(ficha, 0)  # Simulamos el movimiento en el nuevo estado
        nuevo_nodo.bloquea+= nuevo_nodo.bloqueo_ponderado()
        #Queremos ver si el tablero se está manteniendo con las piezas que nos favorecen
        if self.dicc_pasa.get(nuevo_nodo.estado_domino.extremos[0]) or self.dicc_pasa.get(nuevo_nodo.estado_domino.extremos[1]):
            nuevo_nodo.jugadas_inteligentes = self.jugadas_inteligentes+1
        self.mulas_ponderadas()
        nuevo_nodo.estado_domino.jugador=nuevo_nodo.estado_domino.jugador%2+1
        self.hijos.append(nuevo_nodo)
```

Funciones que apoyan a la función heurística

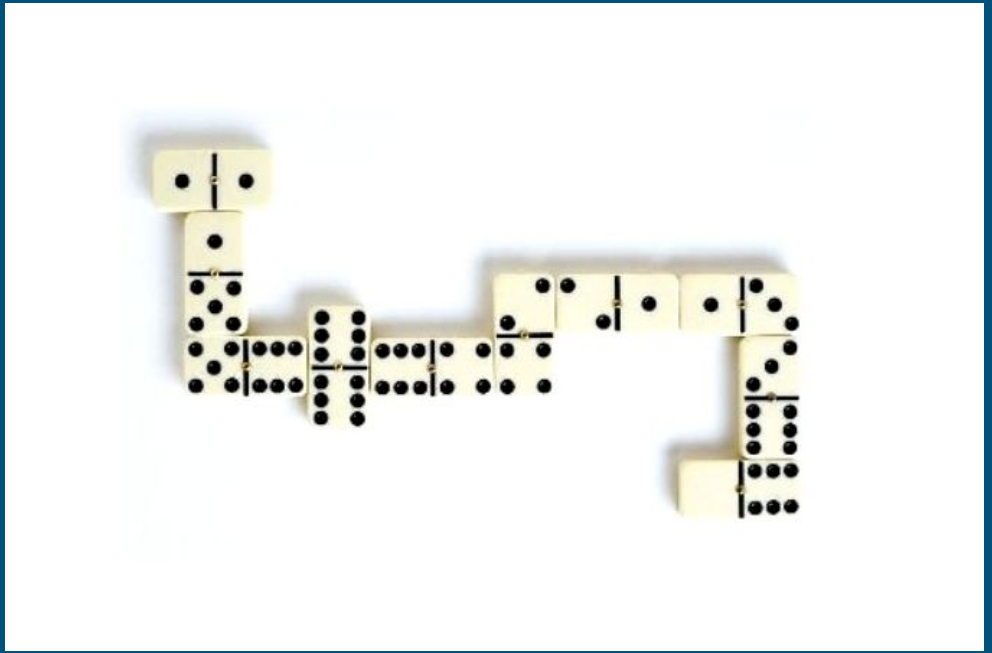
```python
def roba_todo_y_pasa(self,usabilidad):
    if(len(usabilidad)==0):
    #Esta función tiene como único propósito modelar el escenario en que el rival de alguna forma
    #no tuvo fichas usables porque todas las tenemos nosotros así que deberá robar todas las fichas del pozo
        self.jugadas_inteligentes=100
        self.estado_domino.cont2=len(self.estado_domino.fichas)
        #Queremos favorecer MUCHO estos escenarios, por lo que le damos un gran peso.
    elif(len(usabilidad)==1):
        self.jugadas_inteligentes=5
```

```python
  #Actualiza el diccionario
 def actualizar_dicc_pasa(self, num1, num2):
     if self.estado_domino.jugador == 2:
         # Reiniciar todos los valores a False
         self.dicc_pasa = {0: False, 1: False, 2: False, 3: False, 4: False, 5: False, 6: False}

         # Actualizar los valores correspondientes a los números recibidos
         self.dicc_pasa[num1] = True
         self.dicc_pasa[num2] = True
```

Función Heurística

Motivación:

- Siempre tener menos fichas que el contrincante
- Bloquear el juego a nuestro favor
- Tomar en cuenta la probabilidad para tomar decisiones informadas



Función Heurística

```
def evaluar(self):
    cont_jugador1 = self.estado_domino.cont1
    cont_jugador2 = self.estado_domino.cont2

    resta_ponderada = self.diferencia_ponderada(cont_jugador1, cont_jugador2)
    bloqueo_ponderado = self.bloqueo_ponderado()
    proba=self.probabilidad


    eval = proba*(resta_ponderada * 0.4 +bloqueo_ponderado*0.5)+self.bloquea*.5 + self.jugadas_inteligentes*.3 + self.mulas_estrategicas/2



    return eval
```

Clase minimax dominó

```python
class minimax_domino:

    def __init__(self, estado_inicial):
        self.raiz = nodo_minimax(estado_inicial)
        self.profundidad_max=7 #Esto dependerá de
        #la capacidad de cada ordenador!
```

```python
def encontrar_mejor_jugada(self, profundidad):
    #Primero expandimos el nodo para tener de dónde hacer nuestro
    self.raiz.expandir()
    mejor_jugada = None

        #Para que busque en los hijos, primero expandimos el nodo
    mejor_eval=float('-inf')
    for hijo in self.raiz.hijos:

            eval = self.minimax(hijo, profundidad+1, True)
            if eval > mejor_eval:
                mejor_eval = eval
                mejor_jugada = hijo.estado_domino

    return mejor_jugada
```

```python
def minimax(self, nodo, profundidad, es_maximizante):

    nodo.expandir()
    if profundidad == self.profundidad_max or len(nodo.hijos) == 0:

        #Medimos la probabilidad de respuesta, queremos beneficiar a los escenarios donde es baja la
        #probabilidad de respuesta, por eso la modificamos.
        proba=nodo.get_probabilidad()
        nodo.set_probabilidad(1-proba*.5)
        nodo.valor_utilidad_final()
        return nodo.valor_utilidad

    if es_maximizante:
        #Aquí empezamos a maximizar
        max_eval = float("-inf")
        for hijo in nodo.hijos:
            eval = self.minimax(hijo, profundidad + 1, False)
            max_eval = max(max_eval, eval)
        return max_eval
    else:
        #Aquí minimizamos
        min_eval = float("inf")
        for hijo in nodo.hijos:
            eval = self.minimax(hijo, profundidad + 1, True)
            min_eval = min(min_eval, eval)
        return min_eval
```