Pensamiento Computacional CBC

.UBA EXACTAS

Trabajo Práctico Final

2048

Nombre del grupo: Los Padres del git

Tema: 2048

Integrantes del grupo:

- Tiziano Hrabinski Ruta, DNI: 47602390, Mail: tizianohra@gmail.com
- Facundo Colombo, DNI: 44792912, Mail: facundocolombo2601@gmail.com

Curso:

- Turno: (Mañana)
- Comisión: (29001)

2do cuatrimestre 2024

ÍNDICE

0. Descripción del problema

El juego 2048 se basa en combinar fichas dentro de un tablero de 4x4 hasta que una de ellas alcance el valor de 2048. Solo se pueden combinar fichas con el mismo valor. Para combinar las fichas, se tienen que mover, lo cual se hace de una forma particular. Al mover en una dirección, arrastras todas las fichas en esa dirección. La ficha se mueve lo más posible en esa dirección hasta que ocurre una de las tres cosas:

O se combina con otra ficha del mismo valor
O choca contra el borde del tablero
O choca contra una ficha de diferente valor.

Si quedas con un tablero donde no se pueden combinar fichas, el juego termina.

1. Análisis inicial

1.1. El tablero será modelado como una matriz de n x n, las fichas serán representadas con su respectivo valor y los casilleros vacíos serán representados con un **0.**

1.2. Si la matriz que representa al tablero es de nxn, entonces se la puede acceder usando Mat[i] [j] con $\{ \forall\, n,\ i,\ j\ \in Z\ /\ 0\ \leq\ i,\ j\ <\ n \}$. Para acceder a una casilla al azar, tenes que elegir dos números al azar entre -1 y n no inclusive.

1.3. Cuando por ejemplo se recorre el tablero se quiere mover a la derecha, se recorre la matriz de manera inversa, es decir: si me quiero mover a la derecha, empiezo recorriendo la matriz desde la derecha hasta la izquierda.
Con cada iteración que hago me fijo si en la casilla (i, j) hay una ficha. En caso de que haya muevo la ficha hacia la derecha hasta:

- Chocar con el borde del tablet
- Chocar con un número

Si la ficha choca con un número X, me fijo si es de igual valor al de la ficha, en caso de serlo los sumo quedando en

- El casillero donde estaba la ficha queda en 0
- El casillero donde estaba X queda en (X + ficha)

1.4. La lógica de movimiento se repite para cada dirección, ya que siguen las mismas reglas para el comportamiento. Con esto en mente, si rotamos la matriz hasta que el movimiento que queremos hacer termine en la misma posición tal que podemos usar el movimiento que tenemos implementado, nos ahorramos el trabajo de tener que implementar cada movimiento por separado. Una vez efectuado el movimiento, rotamos la matriz para que quede con la orientación que tendría que tener si el movimiento que se hizo estuviese implementado por separado.

1.5. Si todas las casillas son diferentes a 0 y no tienen un vecino de igual valor, el tablero está atascado, y el juego se terminó.

1.6. Hay varias estrategias a seguir, ya sea desde:

- sentido horario(...->arriba -> derecha -> abajo -> izquierda->...)
- sentido anti-horario(...->arriba -> izquierda -> abajo -> derecha->...)
- Secuencial(...->arriba -> abajo -> izquierda -> derecha->...)

Por dar algunos ejemplos.

Para ver cual es la mejor estrategia, podemos hacer miles de iteraciones, comparar los datos y acercarnos a la estrategia ideal.

2. Primera parte de la resolución

Acá deberá incluir cada uno de los ejercicios que incluimos en la consigna en la 1era clase de laboratorio, explicar cómo es que realizaron la resolución, e incluir el código que usaron. En caso de que hubieran tomado alguna suposición para resolver el problema, deberán escribirlo en el ítem correspondiente.

2.1. Crear un tablero vacío

Dentro de mi función crear_tablero(n): donde n es un número natural, usamos de la librería de numpy(np) la función np.zeros para generar una matriz de n x n de 0 en tipo de dato int.

- Desde ahora cuando nos referimos a un tablero, nos estamos refiriendo a una matriz de n x n de int.

```
def crear_tablero(n):
  Tablero = np.zeros((n, n), int)
  return Tablero
```

2.2. Dentro de una función, llamemosla listar_pos_vacias(tablero), podemos hacer que itere sobre cada índice del tablero, y que almacene las posiciones que tienen un 0 dentro de una lista en forma de una tupla, que luego devuelve. Esto queda:

```
def listar_pos_vacías(tablero):
  CasillasVacias = []
  for i in range(tablero.shape[0]):
     for j in range(tablero.shape[1]):
        if tablero[i][j] == 0:
           CasillasVacias.append((i, j))

  return CasillasVacias
```

2.3. Colocar el valor 2 al azar en el tablero

Usando la función que implementamos previamente, llamamos a listar_pos_vacias(tablero) Para que nos de una lista de las posiciones que podemos ocupar azarosamente con un 2. Con la lista, elegimos uno de sus índices al azar, y con esta tupla ocupamos la posición (n, m) en el tablero con un 2.

```
def LlenarCasilleroVacio(Tablero):
  CasillasVacias = listar_pos_vacias(Tablero)
  Tablero[CasillasVacias[rd(0, len(CasillasVacias) - 1)]] = 2
  return Tablero
```

2.4. Programar un ”paso” de un movimiento del jugador

Para mover una ficha hacia la izquierda, usamos la función MoverHaciaIzquierda(i,j,Tablero) donde i y j son enteros y tablero es lo previamente definido.

Mientras que :

- $j \geq 0$
- $Tablero[i, j - 1] = 0$
- $j - 1 \geq 0$

Me voy desplazando hacia la izquierda y le resto 1 a j al final de cada bucle hasta que llega al final del tablero.
Si la posición X = (i, j - 1) y la posición X2 = (i, j) tienen el mismo valor en el tablero, fusiono las fichas.

Haciendo que la posición X valga la suma de los valores de posición X + X2 y que la posición X2 valga 0.

```
def MoverHaciaIzquierda(i, j, Tablero):
  while j >= 0:
    if (j - 1) == -1: return

        #fusionar
    if Tablero[(i, j - 1)] == Tablero[(i, j)]:
      Tablero[(i, j)] = 0
      Tablero[(i, j - 1)] += Tablero[(i, j - 1)]
      return
```

```
    if Tablero[(i, j - 1)] != 0 and Tablero[(i, j - 1)] != Tablero[i, j)]:  # Colisión entre
fichas de diferente valor
        return

    else:  # Desplazar hacia la izquierda
        Tablero[(i, j - 1)] = Tablero[(i, j)]
        Tablero[(i, j)] = 0

    j -= 1
```

2.5. Movimiento en todo el tablero en cualquier dirección

Sabiendo que el movimiento sigue el mismo proceso lógico sin importar la dirección (este siendo el chequeo contra vecinos o borde del tablero), como previamente dicho, si se rota la matriz que representa el tablero hasta que la arista donde se tendría que efectuar el movimiento queda donde estaba la arista hacia la izquierda (o la arista del movimiento implementado), se puede usar exactamente el mismo código, ya que termina siendo lo mismo. Una vez efectuado el movimiento, rotamos nuevamente la matriz para que todo vuelva a su lugar.

```
def RotarTableroHorizontalmenteNoventaGrados(Tablero):
  n = Tablero.shape[0]
  TableroNuevo = np.zeros((n, n), int)
  for i in range(n):
    for j in range(n):
      TableroNuevo[(j, abs(i - (n - 1)))] = Tablero[(i, j)]

  return TableroNuevo
```

La implementación de esta idea calcula la posición que debería tomar el valor en [(i, j)] al rotar la matriz y la plasma en TableroNuevo con el valor correspondiente.

2.6. Determinar si el jugador ya perdió

Para determinar si el jugador ya perdió, utilizo la función

esta_atascado(tablero), la cual contempla que el juego terminó cuando:

- Hay un 2048 en el tablero
- No hay ningún otro movimiento posible

La falta de movimiento se corrobora con la función HayMovimientoPosible(tablero), la cual verifica que todas las casillas no tengan algún tipo de movimiento posible con sus casillas vecinas. Es decir:

- La casilla vecina no tiene el mismo valor que la casilla que se está verificando
- La casilla vecino es diferente de 0

A su vez, la función HayMovimientoPosible(tablero) usa la función ObtenerVecinos(tablero,posicion), la cual requiere por parámetro un tablero y una tupla con las coordenadas de la posición de la que se desea obtener vecinos.

Para obtenerlos, verifica si en las 4 direcciones posibles (izquierda, arriba, derecha, abajo) tiene un vecino (ya que la ficha podría estar en una esquina o borde), y si las direcciones partiendo desde la posición donde estamos parados fueron corroboradas correctamente, son añadidas a una lista de “vecinos”.

```
def esta_atascado(tablero):
  atascado = 2048 in tablero
  atascado = atascado or not HayMovimientoPosible(tablero)
  return atascado

def HayMovimientoPosible(matriz):
  if 0 in matriz: return True

  movimientoposible = False
  i = 0
  # si soy un 0 no hace falta seguir
  while not movimientoposible and i < matriz.shape[0]:
    j = 0
    while j < matriz.shape[1] and not movimientoposible:
      vecinos = ObtenerVecinos(matriz, (i, j))
      for vecino in vecinos:
        if matriz[vecino[0]][vecino[1]] == matriz[(i, j)] or matriz[vecino[0]][vecino[1]] == 0:
          movimientoposible = True
      j += 1
    i += 1
```

```
  # if not movimientoposible: print("no hay movimiento posible en", matriz)
  return movimientoposible


def ObtenerVecinos(matriz, posición):
  direcciones = [(0, 1), (0, -1), (1, 0), (-1, 0)]
  posicionesVecinas = []
  for dir in direcciones:
    NuevaPosicion = np.array(dir) + np.array(posición)
    if EnTablero(matriz, NuevaPosicion):
      posicionesVecinas.append(tuple(NuevaPosicion))

  return posicionesVecinas


def EnTablero(matriz, Posicion):
  for n in range(2):
    if matriz.shape[n] <= Posicion[n] or Posicion[n] < 0:
      return False

  return True
```

2.7. Partida completa con estrategia al azar

Para probar posibles estrategias usamos la funcion EjecutarEstrategia(ObtenerMovimiento) la cual recibe por parámetro una funcion (la cual recibe por parámetro un tablero y devuelve un string con el comando a usar en esa iteración del juego) y devuelve un data frame para analizar.

```
def EjecutarEstrategia(obtenerMovimiento):
  df = pd.DataFrame()

  tablero = GameLogic.crear_tablero(4)
  tablero = GameLogic.llenar_pos_vacias(tablero, 2)

  i = 0
  finJuego = False
  while not finJuego and not GameLogic.esta_atascado(tablero) and i < 1000:
# es un watch dog
    comandoActual = obtenerMovimiento(cp.deepcopy(tablero))
    if not GameLogic.mover(tablero, comandoActual):
      finJuego = True
    i += 1
    tablero = GameLogic.llenar_pos_vacias(tablero, 1)

  df["Cantidad de turnos"] = [i]
  df["SumatoriaTotal"] = [sum(tablero)]
  df["NumeroMasAlto"] = [np.max(tablero)]
  print(tablero)
  return df
```

Para probar partidas al azar específicamente, le pasamos a EjecutarEstrategia(ObtenerMovimiento) la función EjectutarMovimiento Aleatorio(Tablero) la cual recibe un tablero.
La funcion tiene una lista con los posibles comandos, estos son “barajados” y puestos a prueba cada vez que es llamada.

3. Otras preguntas que nos interesa resolver

Acá deberá incluir las preguntas adicionales que les planteamos en la 2da clase de laboratorio. Como en este caso no les damos instrucciones precisas de qué deben resolver, deberán incluir más detalles de cómo y por qué decidieron resolver cada ítem de determinada manera. Se deberán incluir todos los gráficos y explicaciones que crean necesarias para responder las consignas planteadas.

3.1.

Aclaración: todas las funciones a continuación son pasadas como parámetro de EjecutarEstrategia(ObtenerMovimiento) y estrategia 1 y 2 devuelven la función ProbarComando(tablero, comando) la cual ejecuta los movimientos:

```
def ProbarComando(tablero, comando):
  MovimientoPosible = False
  comandoActual = ""
  while len(comando) > 0 and not MovimientoPosible:
     comandoActual = comando.pop(0)
     MovimientoPosible = Logic2048.mover(tablero, comandoActual)

  return comandoActual
```

- Estrategia 1: Arriba -> Derecha -> izquierda -> abajo

  Esta estrategia intenta siempre llevar los valores hacia arriba a la derecha siguiendo la secuencia de arriba:

```
def ObtenerMovimientoEsquinaDerecha(tablero):
  ComandosASeguir = ["arriba", "derecha", "izquierda", "abajo"]
  return ProbarComando(tablero, ComandosASeguir)
```

- Estrategia 2: Izquierda -> abajo-> arriba-> derecha

  Opuesta a la 1er estrategia, esta intenta llevar los valores hacia la esquina inferior izquierda

```
def ObtenerMovimientoEsquinaIzquierda(tablero):
  ComandosASeguir = ["izquierda", "abajo", "arriba", "derecha"]
  return ProbarComando(tablero,ComandosASeguir)
```

Estrategia 3: Montecarlo

Esta estrategia nace de un intento de replicar GOAP(Goal Oriented Action Programing) para el juego 2048.
GOAP es un tipo de IA que se usa en videojuegos para hacer planes y ejecutar acciones para llegar a un estado.

Informando más del tema, descubrí que lo que quería hacer en realidad eran bifurcaciones de posibles caminos en el juego en vez de llegar a un estado.
Para esto use el método **Montecarlo**, un algoritmo que te permite ver el mejor posible camino en base a simulaciones de qué pasaría si hago x movimiento y viendo si me deja un mejor o peor tablero.
A partir de esas simulaciones puedo definir el mejor movimiento a realizar.
Mi replicación de este algoritmo está compuesto por estas funciones:

- ObtenerTablerosPosibles(Tablero) :
  Esta funcion pide por parametro un tablero y devuelve un diccionario de {string : tablero}.
  Este método simula lo que pasaría con el tablero si se intenta mover en x dirección. Si el tablero se puede mover en esa dirección entonces se añade al diccionario con su respectivo movimiento, caso contrario no se añade.

```
def ObtenerTablerosPosibles(tablero):
  tableros = {}
  for movimiento in ["izquierda", "derecha", "arriba", "abajo"]:
     tableroSimulado = cp.deepcopy(tablero)
     if Logic2048.mover(tableroSimulado, movimiento) and
HayMovimientoPosible(tableroSimulado):
        tableros[movimiento] = tableroSimulado

  return tableros
```

- AgarrarRandom(lista,n):
  - Esta funcion pide por parámetro una lista y un número n.
  - Devuelve una lista
  El método agarra n elementos random de una lista y la devuelve

```
#https://www.geeksforgeeks.org/python-random-sample-function/
def AgarrarRandom(lista, n):
  if n > len(lista):
     n = len(lista)
  return random.sample(lista, n)
```

- TendraMovimiento(tablero)
  - Recibe un tablero
  - Devuelve un bool

El método corrobora si el tablero tendrá un movimiento

```
def TendraMovimiento(tablero):
  tablerosCopia = ObtenerTablerosPosibles(tablero)
  MovimientoPosible = len(tablerosCopia) > 0
  return MovimientoPosible
```

- CantidadMovimientosFuturos(tablero,n,ramificado)
    - Recibe un tablero, un int n para agarrar posiciones vacías y un int para el “ramificado” de la funcion
    - Devuelve un int con la cantidad de tableros exitosos

El método trata de predecir la cantidad de movimientos posibles que puede llegar a tener el tablero simulando que pasaría si relleno x posición con un 0 y me intento mover en las 4 direcciones con el método ObtenerTablerosPosibles.
La cantidad de tableros posibles que sale de esto es sumado a una variable local llamada “éxitos”, luego de esto se vuelve a hacer el método de manera recursiva para sumarse a éxitos haciendo una simulación del tablero simulado, esto se hace “ramificado - 1” veces.
Si ramificado es igual a 0, entonces la función devuelve 0, haciendo que la función “principal” reciba la cantidad de éxitos totales.

```
def CantidadMovimientosFuturos(tablero, n, ramificado):
  if ramificado <= 0: return 0
  exitos = 0

  posicionesVacias = AgarrarRandom(Logic2048.listar_pos_vacias(tablero), n)
  if len(posicionesVacias) <= 0: return exitos

  for i in range(len(posicionesVacias) - 1):
     tableroCopia = cp.deepcopy(tablero)
     tableroCopia[posicionesVacias[i]] = 2
     posibles = len(ObtenerTablerosPosibles(tableroCopia))
     if posibles > 0:
        exitos += posibles / 4
        exitos += CantidadMovimientosFuturos(tableroCopia, n, ramificado - 1)

  return exitos
```

Para normalizar “exitos” lo divido por n * ramificación.

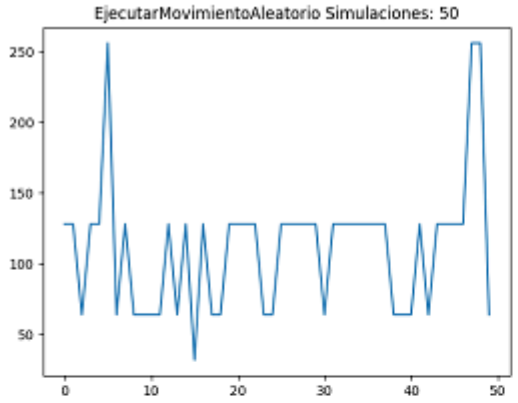
- MejorTablero(tableros,dimTablero)
- recibe una lista de tableros y la dimension de los tableros

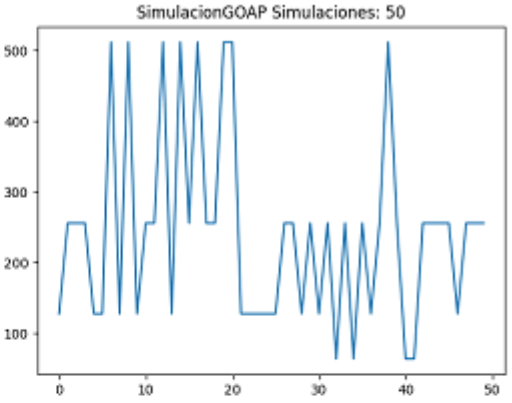
- Devuelve un tablero
  La función se decide cuál de los 4 tableros es el mejor en base a
  - el promedio de futuros posibles (Para normalizar “exitos” lo divido por n * ramificación).
  - Cual es el número más alto de ese tablero dividido 2048(para normalizar)

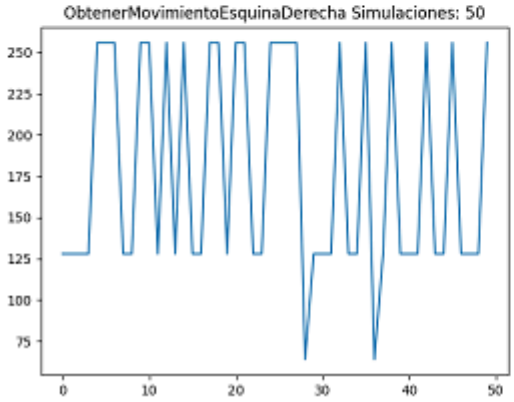
```
def MejorTablero(tableros, dimTablero):
  n = 3
  ramificado = 1
  mejorTableroActualmente = (-1, np.zeros((dimTablero, dimTablero), dtype=int))
  for tablero in tableros:
    if TendraMovimiento(tablero):
      tablerosPosibles = CantidadMovimientosFuturos(cp.deepcopy(tablero), n, ramificado) / n * ramificado
      tablerosPosibles += np.max(tablero) / 2048
      if tablerosPosibles > mejorTableroActualmente[0]:
        mejorTableroActualmente = (tablerosPosibles, tablero)
  return mejorTableroActualmente[1]
```

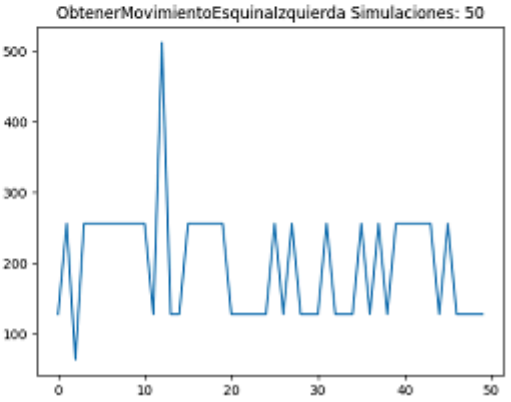
- SimulacionGOAP(tablero)
- Recibe un tablero
- Devuelve un string (comando)
  Usa la funcion tableros posibles para conseguir todos los tableros “futuros” de ese tablero,la cual devuelve un diccionario de {string : tablero}.
  Si consigue tableros posibles, analiza cuál es el mejor en base a la funcion mejor tablero

SIMULACION 4X4

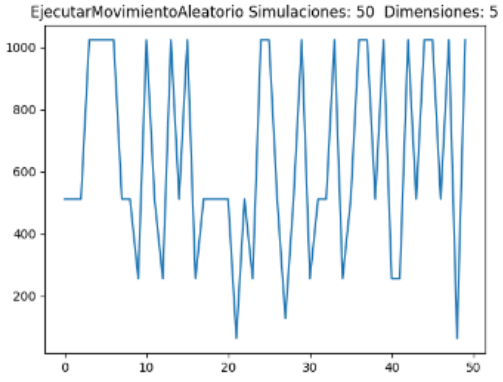


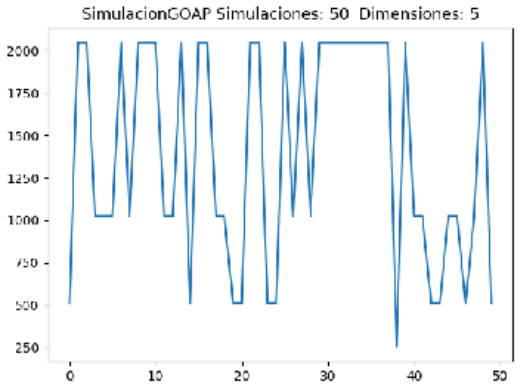


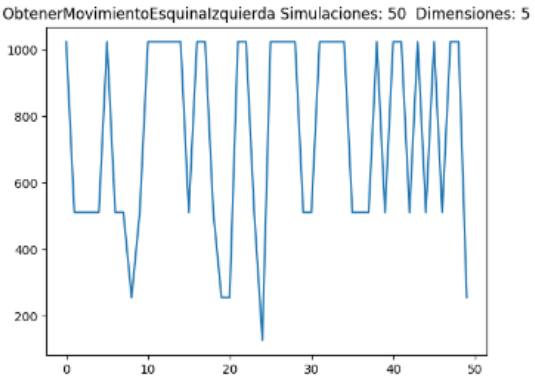


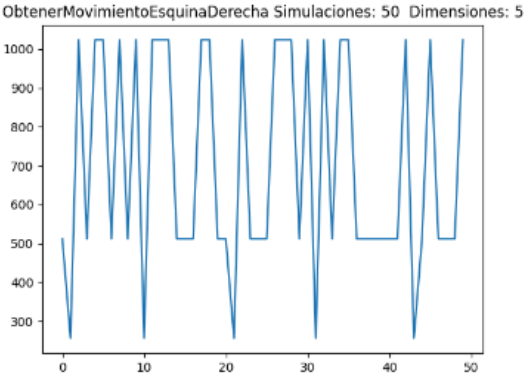


SIMULACION 5X5

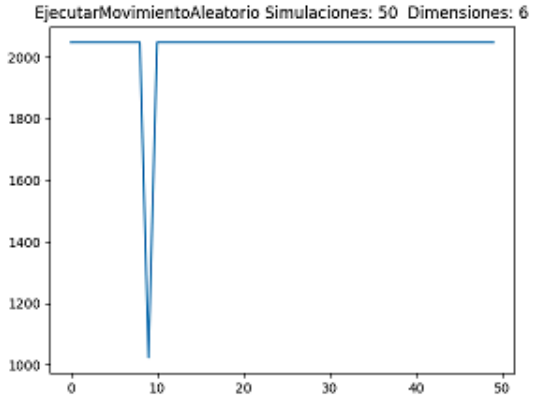


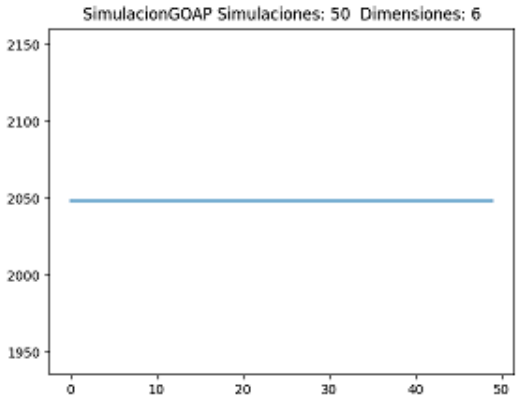


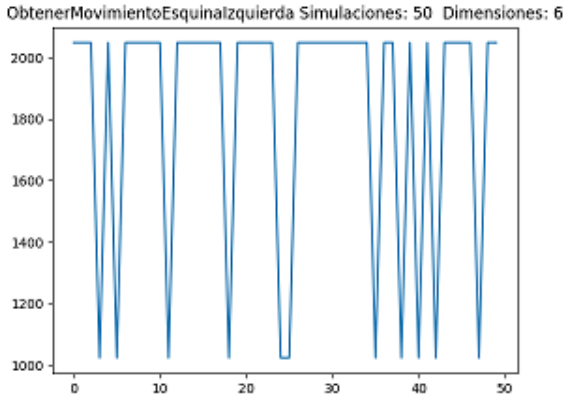


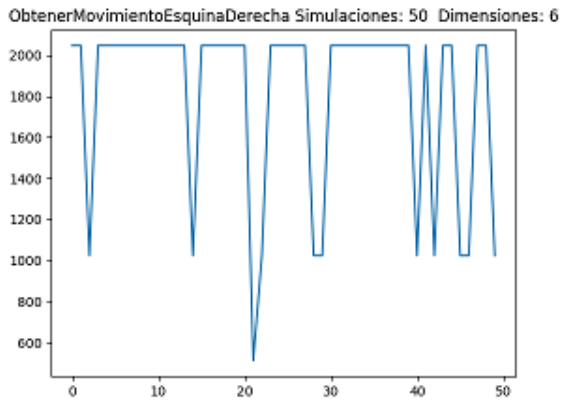


SIMULACION 6X6









4. Conclusiones del trabajo

Acá deberán poner sus conclusiones sobre el trabajo con la resolución del problema:

- *Qué dificultades encontraron y cómo las solucionaron.*
- *Evaluar críticamente el resultado del trabajo comparado con lo que ustedes habían pensado en la primera clase, donde se hizo el análisis del problema (sin usar computadoras).*
- *Indicar algunas cosas que quedaría para trabajar a futuro a partir de los que hicieron como parte del trabajo, ya que sea una variante de lo hecho, o porque implicaría incluir más funcionalidades o extensiones sobre lo pedido.*

Con ya los algoritmos planteados en la clase teórica, hacer el trabajo fue una cuestión de desglosar lo hablado en funciones pequeñas.

Lo que más dio dificultad fue el hacer la IA con el método Montecarlo ya que requirió informarnos de temas que son más avanzados de lo que vimos en la cursada.

A futuro tenemos que volver a ver la IA para mejorarla para hacer que sea más rápida y eficiente(ya que si estuviera más trabajada y pulida debería poder siempre ganar en 2048). Algo a destacar que me sorprendió es ver que con un tablero de 6X6 usar la estrategia de movimientos aleatorios es casi igual de eficiente que “predecir el futuro” con el método Montecarlo.

Tiziano Hrabinski Ruta
Facundo Colombo