UPC
Universidad Peruana
de Ciencias Aplicadas

FACULTAD DE INGENIERÍA

CARRERA PROFESIONAL

Asignatura:

Complejidad algorítmica

Informe de proyecto:

Quoridor

Autor(es):

Hector Egocheaga Rincón

Profesor:

Sopla Maslucan, Abraham

Lima, Noviembre del 2020

Introducción

En la década de los 80 's se dio inicio a la era de programación de videojuegos, estos juegos eran programados en lenguajes de programación de bajo nivel. A su vez, en el año 2012 periódico inglés The Guardian aseguraba que era una época de auge para los juegos de mesa, ya que aseguraban que durante esos años se presentó un crecimiento hasta del 40%. Por ello, se desarrollará un juego de mesa Quoridor, donde cada jugador maneja un peón y una serie de barreras para obstaculizar el paso del otro, este se realizará en el lenguaje de programación Python, el cual simplifica mucho la programación realizando un trabajo en pocas líneas de código, lo cual lo hace flexible sin preocuparse tanto por los detalles y limpio. Para la realización del proyecto serán necesarias herramientas como el Suite de código abierto Anaconda, el módulo del lenguaje de programación PyGame, GitHub como repositorio virtual y la metodología ágil de proyectos SCRUM.

Para el desarrollo de este proyecto , se piensa implementar técnicas de programación aprendidas en el curso de Complejidad Algorítmica utilizando el lenguaje de programación Python. El proyecto que se va a trabajar esta en base al juego de mesa: Quoridor.

Estado del arte

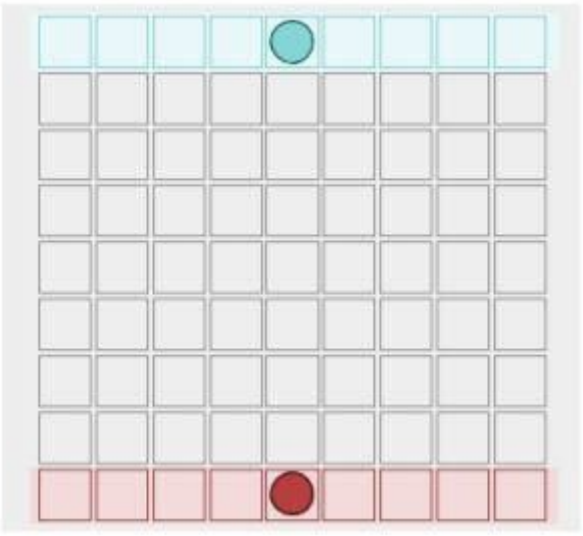
Historia de Quoridor

Quoridor es un juego de mesa de estrategia de dos a cuatro jugadores. Este fue creado en Francia por la empresa Gigamic en 1997. El juego fue inspirado por el trabajo de Mirko Marchesi que creó un juego parecido llamado Pinko Pallino. Quoridor ha recibido distintos reconocimientos como; Mensa Mind Game en 1997, Best Bet of the Toy Testing Council, 1997, Grand Prix du Jouet, 1997, entre otros.

El juego de estrategia es similar a otros tipo de juegos de mesa como Ajedrez o Damas. Quoridor es un juego relativamente reciente y no ha sido estudiado profundamente. Es decir, todavía se puede descubrir nuevas estrategias y métodos para disfrutar el juego.

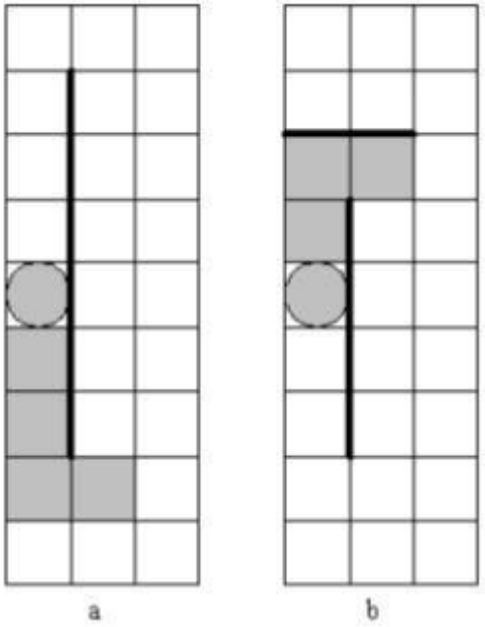
Reglas de Quoridor

Quoridor es un juego con una tabla de 9x9 casilleros. Cada jugador es representado por un peón, el cual está situado en la casilla media de los bordes y el otro jugador debe poner su pieza en frente. El objetivo es ser el primer jugador en mover tu peón al borde opuesto de donde empezó.



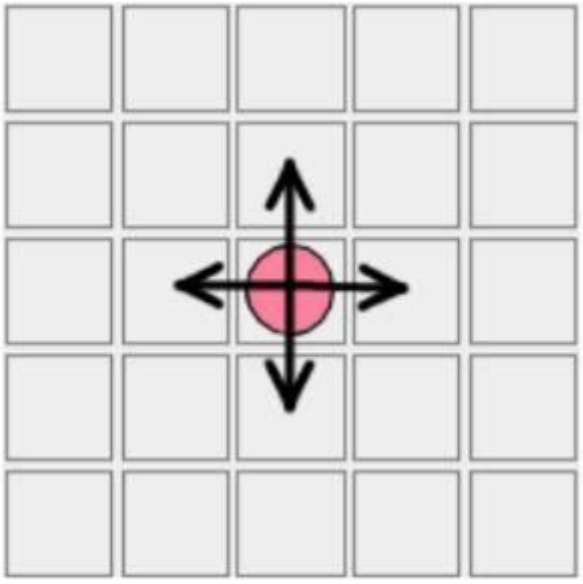
Este es una representación de la tabla con los peones de los jugadores.

La principal característica que hace a este juego de estrategia interesante es el uso de bloques. Estos tienen un largo de dos casillas y pueden ser colocados de forma vertical o horizontal. Los bloques tienen el objetivo de facilitar el progreso del jugador o bloquear el camino del jugador contrario obligándolo a rodear las casillas y perder tiempo. Cada jugador tiene diez bloques y una vez puestos, no pueden ser movidos o eliminados.

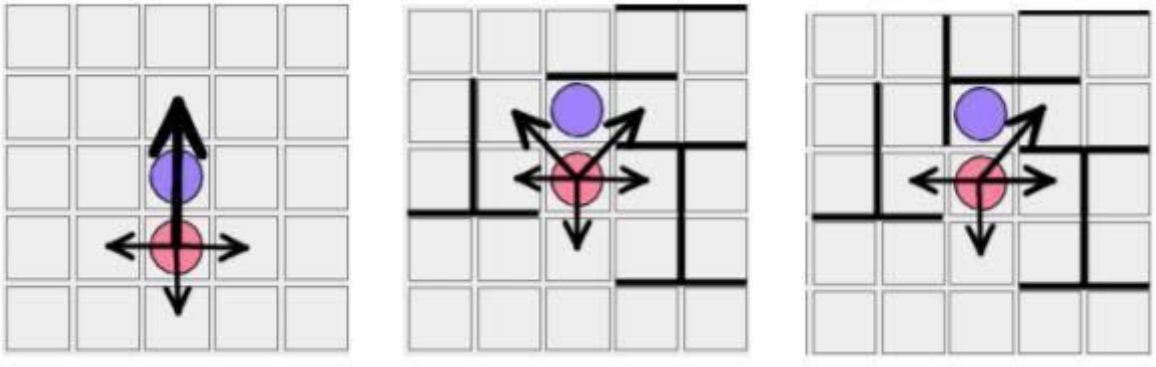


En la imagen a, se puede ver como los bloques están siendo usados en beneficio al jugador. Mientras en la imagen b, el bloque impide al jugador avanzar, lo cual obliga al jugador a dar una vuelta.

En cada turno, cada jugador puede escoger entre mover su peón o colocar un bloque. Una vez el jugador se quede sin bloques, está obligado a mover su peón. Este solo puede desplazarse por un casillero por turno en dirección vertical u horizontal o hacia adelante o para atrás.



Cuando los dos peones están en casilleros adyacentes, el jugador puede saltar sobre el peón del oponente. De esta forma, el peón avanza un casillero extra. Si hay un bloque, el jugador puede colocar su peón al lado izquierdo o derecho del oponente. Los bloques no pueden ser saltados.

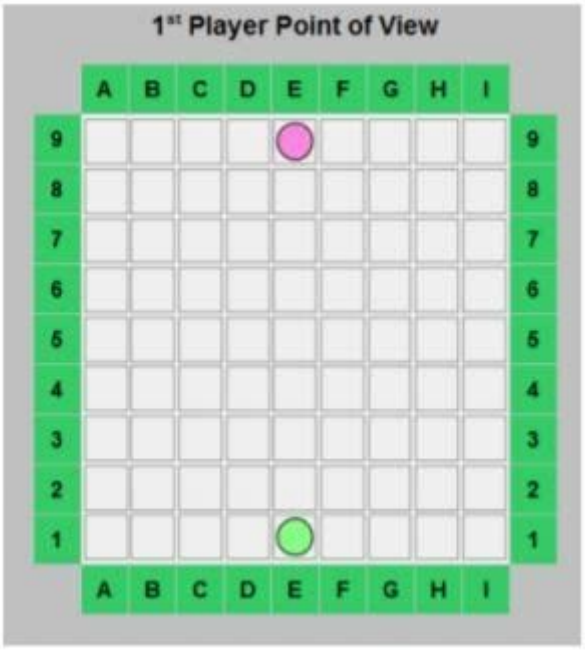


Los bloques pueden ser colocados directamente entre dos espacios que no estén ocupados por otro bloque. Sin embargo, un bloque no puede colocarse en el único camino que puede seguir un peón para llegar a su objetivo. El primer jugador que llegue a cualquiera de las nuevas casillas del lado del oponente es el ganador.

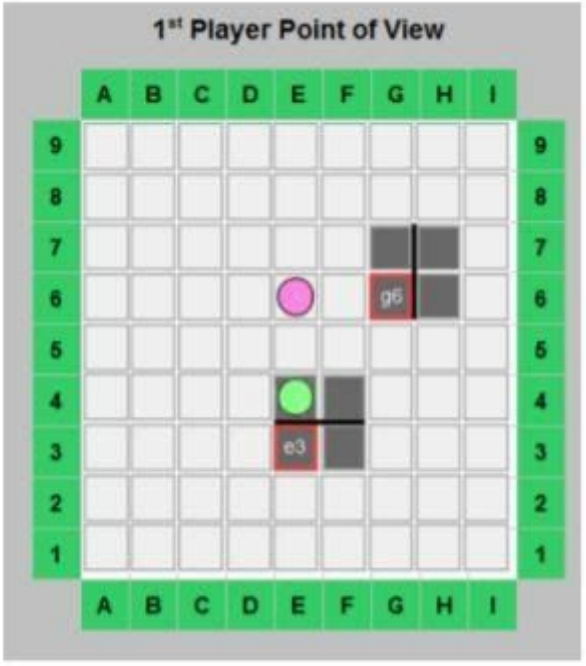
Notación

En busca de encontrar una forma de examinar los movimientos de los jugadores se ha decidido establecer una notación propuesta por la comunidad del juego, ya que no existe una oficial.

La notación propuesta es similar a la de un tablero de ajedrez. Cada casillero tiene un número o letra único. Cada columna es nombrada con una letra a partir de la A hasta la I y a cada fila se le asigna un número del 1 al 9. Cada movimiento es registrado primero con la columna seguido de la fila. El primer jugador siempre empieza en E1 y el jugador oponente, en E9.



Cada bloque está registrado por la casilla izquierda más cercana al lado inferior de la tabla y se debe especificar si está en forma horizontal o vertical. El bloque toca cuatro casilleros.

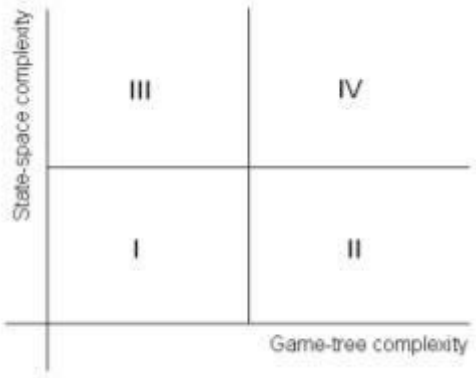


Para distinguir entre el movimiento de un peón o bloque podemos observar en la tabla que los peones están registrados por la notación de una casilla. Mientras que los bloques, terminan indicando su dirección; horizontal o vertical.

	Player 1	Player 2
1	E2	E8
2	E3	E7
3	E4	E6
4	E3h	G6v

Complejidad del juego

Antes de empezar a analizar la complejidad del juego, se debe definir dos términos: State-Space y Game-tree. El State-space o espacio de búsqueda se refiere al número de diferentes posibles posiciones que pueden ocurrir en un juego. Mientras, el game-tree se refiere al número total de jugadores que pueden suceder. El juego se puede colocar en una de cuatro categorías.



La primera categoría se refieren a juegos como Tic-Tac-Toe. Estos juegos tienen un pequeño Game-Tree y State-Space pequeño. La segunda categoría son juegos con mayor Game-tree pero pequeño State-Space. Algoritmos de Fuerza Bruta son usados comúnmente para este tipo de juegos. Caso contrario son los juegos de la tercera categoría, donde el State-Space es muy grande para utilizar métodos de Fuerza Bruta pero el Game-Tree es suficientemente pequeño para utilizar un Árbol binario de búsqueda. En la cuarta categoría se encuentran juegos como Ajedrez. Estos tienen un gran State-Space y Game-Tree y son mucho más complejos para superar.

El State-Space es un número de posibles posiciones en el juego. En Quoridor es el número de caminos para colocar el peón multiplicado por el número de maneras de colocar bloques. Sin

embargo, desde que hay un número de jugadas no permitidas, lo vuelve difícil de calcular. Existen 81 casilleros para colocar un peón, lo cual quedaría 80 casilleros libres para colocar el otro peón. Entonces, el total de número de posiciones S con dos peones:

$$S_p = 81 * 80 = 6480$$

Para calcular el número total de situaciones obtenidas por bloques, el número de maneras para colocar un bloque tiene que ser conocido. Ya que cada bloque tiene una longitud de dos casilleros, existen 8 maneras para colocar un bloque en una fila. Dado que hay 8 filas, entonces existen 64 posiciones para poner un bloque de forma horizontal en la tabla. Ya que existen la misma cantidad de columnas como de filas, un bloque puede ser puesto en 128 formas diferentes. Pero un bloque ocupa cuatro posiciones. Entonces el total de posibles posiciones de bloques puede ser estimado de la siguiente manera.

$$S_f = \sum_{i=0}^{20} \prod_{j=0}^{i} (128 - 4i) = 6.1582\ 10^{38}$$

Para conseguir un estimado del tamaño del State-Space, el número tiene que ser multiplicado con el número de posiciones del peón. Entonces el total de Space-State está dado de la siguiente forma.

$$S = S_p * S_f = 6480 * 6.1582\ 10^{38} = 3.9905\ 10^{42}$$

El Game-Tree está estimado por la elevación del promedio del Branching factor por el promedio del número de Ply. El Branching factor es el número de nodos que tiene un Game-Tree. El Ply es el movimiento que realiza un jugador, entonces el total del número de Ply es el resumen del número de jugadas hechas por los jugadores. Se estima que el promedio de Branching factor es 60.4 y el promedio de jugadas por juego es 91.1. Ahora el Game-Tree puede ser estimado con la siguiente ecuación.

$$G = 60.4^{91.1} = 1.7884\ 10^{162}$$

El State-Space y Game-Tree de Quoridor puede ser comparado con la complejidad de otros juegos de mesa reconocidos. De la tabla mostrada podemos concluir que Quoridor tiene similar State-Space del Ajedrez y mayor Game-Tree. Entonces, Quoridor pertenece a la cuarta categoría según su dificultad.

Game	log(state-space)	log(game-tree)
Tic-tac-toe	3	5
Nine Men's Morris	10	50
Awari/Oware	16	32
Pentominoes	12	18
Connect Four	14	21
Checkers	33	50
Lines of Action	24	56
Othello	28	58
Backgammon	20	144
Quoridor	42	162
Chess	46	123
Xiangqi	52	150
Arimaa	42	190
Shogi	71	226
Connect6	172	140
Go	172	360

Metodología del documento

Lenguaje

- **Python:** Python es un lenguaje de programación Open Source. Este está optimizado de tal manera que garantiza calidad, productividad, portabilidad e integración. Es usado por un gran número de desarrolladores en todo el mundo en distintas áreas: Internet scripting, programación de sistemas, interfaces de usuario, entre otros.

Librerías y entorno de desarrollo:

Se utilizará la siguiente bibliotecas para la creación y experimentación del proyecto:

- Numpy: esta librería proporciona una estructura de datos de matriz que tiene algunos beneficios sobre las listas regulares de Python. Algunos de estos beneficios son: ser más compacto, acceder más rápido a leer y escribir artículos, ser más conveniente y más eficiente.

  NumPy es un paquete de Python que significa "Numerical Python", es la librería principal para la informática científica, proporciona potentes estructuras de datos, implementando matrices y matrices multidimensionales. Estas estructuras de datos garantizan cálculos eficientes con matrices.

- Seaborn: es una librería para Python que permite generar fácilmente elegantes gráficos. Seaborn está basada en matplotlib y proporciona una interfaz de alto nivel que es realmente sencilla de aprender. Dada su gran popularidad se encuentra instalada por defecto en la distribución Anaconda.

  La representación de datos es una tarea clave del análisis de datos. La utilización de una gráfica adecuada puede hacer que los resultados y conclusiones se comuniquen de una forma adecuada o no. Conocer y manejar diferentes herramientas es clave para poder seleccionar la gráfica adecuada en cada ocasión. En esta entrada se va a repasar básicamente las funciones que ofrece la librería Seaborn.

- Matplotlib: es una biblioteca para la generación de gráficos a partir de datos contenidos en listas o arrays en el lenguaje de programación Python y su extensión

matemática NumPy. Proporciona una API, pylab, diseñada para recordar a la de MATLAB.

- Time: librería time será utilizada para extraer el tiempo que el algoritmo demora en resolver sus retos.
- Random: librería que genera valores aleatorios

Validaciones:

- ❏ El juego tiene como mínimo 2 jugadores y como máximo 4.
- ❏ Si el usuario realiza un movimiento de bloque, este ocupará dos espacios de casillas en el tablero, teniendo en cuenta de que solo puede colocar un bloque por turno.
- ❏ Se tendrá que hacer un click sobre la ficha del jugador para validar su movimiento.
- ❏ El jugador solo puede mover un espacio de forma vertical u horizontal, teniendo en cuenta que solo puede realizar un movimiento por turno.
- ❏ Los jugadores no pueden limitar a totalidad su paso al otro extremo.
- ❏ El jugador puede colocar una barrera (cada barrera bloquea dos cuadros y debe colocarse de manera que lo haga perfectamente para evitar confusiones)
- ❏ En caso que un jugador quede al frente de otro, este en su turno puede escoger saltar por encima del primero, en caso no pueda realizar ese movimiento, podría moverse en diagonal.
- ❏ El juego termina cuando un jugador llega al otro extremo del lado del tablero del cual inició, para así obtener la victoria.

Declaración de algoritmos

- **Backtracking:** Es una estrategia para encontrar soluciones, este se destaca porque no realiza una búsqueda extensa sino porque toma parte de ella para encontrarla. Este se usará para detectar los movimientos que realiza cada jugador en el tablero.

```
def buscarEspacioBloque(posicionMouse, muro):
  x,y=posicionMouse()
if muro.getEstadoColocado()==False:
 if(muro.getPosX()< x < muro.getPosX()+muro.getAncho() and muro.getPosY() < y < muro.getPosY()+muro.getAlto()):
pygame.draw.rect(pantalla, Colores.azul, [muro.getPosX(), muro.getPosY(), muro.getAncho(), muro.getAlto()], 0)
else:
buscarEspacioBloque(posicionMouse, muro)
```

- **<u>BFS:</u>**Es un algoritmo de búsqueda no informada utilizado para recorrer o buscar elementos en un grafo . Será utilizado para recorrer el tablero , explorando todas los posibles movimientos que se pueda recorrer que estén disponibles para llegar a su punto final.

```
def bfs(G, s):
    n = len(G)
    path = [None]*n
    visited = [False]*n
    queue = [s]
    visited[s] = True
    while len(queue) > 0:
        u = queue[0]
        queue = queue[1:]
        #print(G[u])
        for v in G[u]:
            if not visited[v]:
                queue.append(v)
                path[v] = u
                visited[v] = True
    return path
```

- **Fuerza Bruta**: Es una técnica que consiste en enumerar sistemáticamente todos los posibles candidatos para la solución de un problema con el fin de verificar si cada opción cumple con la solución de un problema. Esta se considera sencilla de implementar y encuentra una solución. Sin embargo, su coste de ejecución es proporcional al número de soluciones posibles.

  El siguiente código es un ejemplo de Fuerza Bruta que se utilizará en el proyecto.

```
def crearTabla(dimensiones,casilla):
   for i in range(0,dimensiones[0],casilla.getAncho()):
       for j in range(0,dimensiones[1],casilla.getAlto()):
            pygame.draw.rect(pantalla, Colores.negro, [i, j, 80, 10, 0)
          pygame.draw.rect(pantalla, Colores.negro, [i, j, 80, 10, 0)
```

Eficiencia de los algoritmos a usar

A continuación se mostrarán el nombre los algoritmos junto con sus respectiva eficiencia algorítmica:

- Backtracking : T(n)=1+T(n)+O(n)
- Quick : T(n)=1+2T(n)+O(n)
- BFS : T(n)=1+T(n)+O(n)
- Fuerza Bruta : T(n)=1+T(n)+O(n)

Desarrollo

El proyecto se ha decido realizar de manera que los jugadores tengan decisiones aleatorias. Esto quiere decir que cada vez que se compile el código, se tendrá un resultado diferente.

Bibliotecas:

Las bibliotecas utilizadas se mostrarán de las siguiente forma:

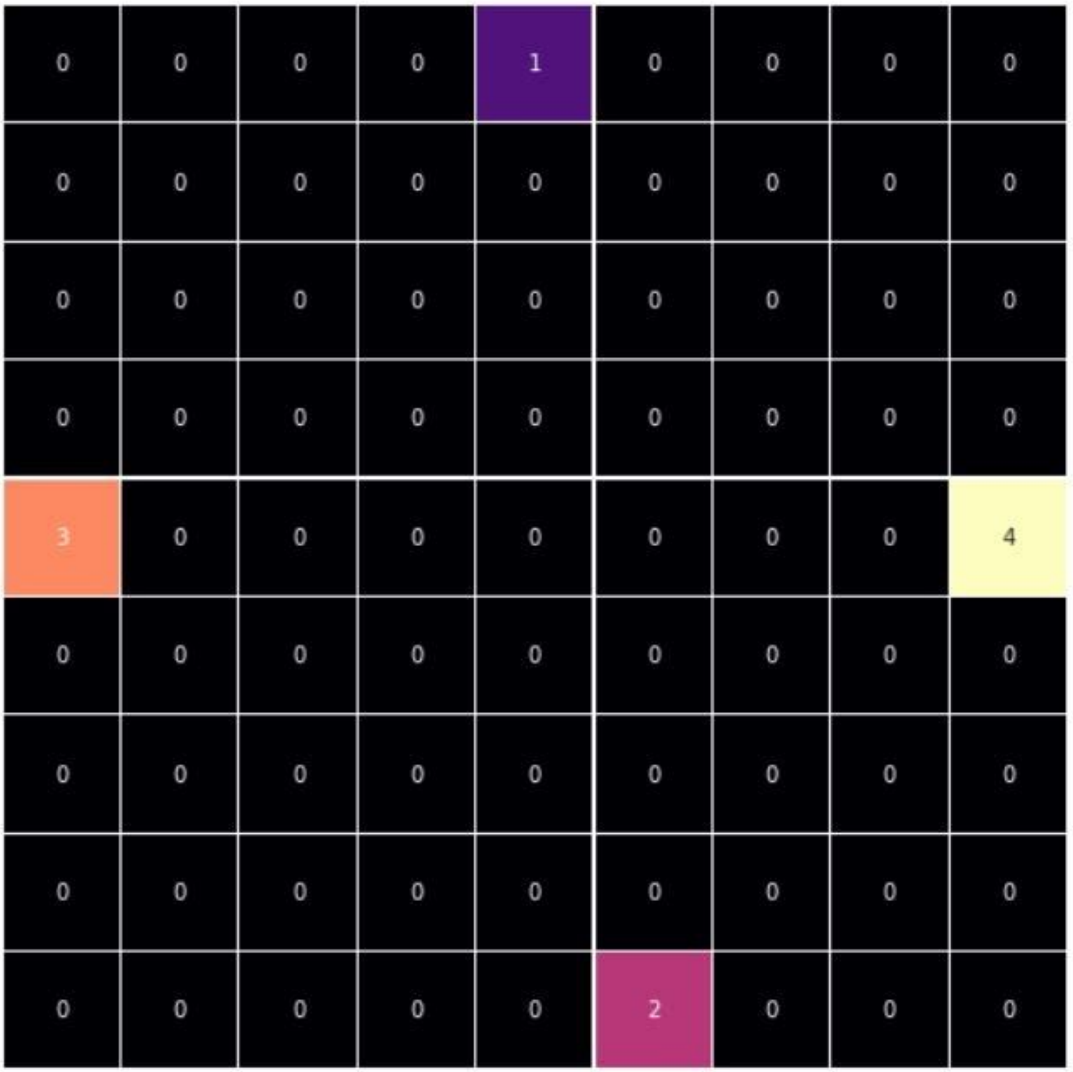
```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import random
import time
from collections import deque
```

Tablero:

Primero se crea la clase tablero. En el cual, recibe el tamaño del tablero y se coloca el valor de “0” a cada cuadro. Después, se define las posiciones de los jugadores, los cuales tendrán un valor distinto a 0. Siendo el primer jugador el valor “1”; el segundo jugador, “2”; el tercer jugador, “3” y el cuarto jugador, “4”. Una vez se haya asignado los valores, se crea una función que permite mostrar el tablero, donde se diferencia cada valor con un color. Por último, se crean funciones que retornan la matriz del tablero y posiciones de los jugadores.

```
class Tablero:

  def __init__(self, n : int):
    self.tamanio = n
    self.matriz = np.zeros(shape=(self.tamanio,self.tamanio)).astype(int)
    self.pos_jugador = (-1,-1)
    self.pos_jugador_segundo = (-1,-1)
    self.pos_jugador_tercero = (-1,-1)
    self.pos_jugador_cuarto = (-1,-1)

  def Generar_Tablero(self):

    pos_x = self.tamanio // 2
    pos_y = 0

    pos_x_p2 = (self.tamanio // 2) + 1
    pos_y_p2 = self.tamanio - 1

    pos_x_p3 = 0
    pos_y_p3 = self.tamanio // 2

    pos_x_p4 = self.tamanio - 1
    pos_y_p4 = self.tamanio // 2

    self.matriz[pos_y,pos_x] = 1
    self.matriz[pos_y_p2,pos_x_p2] = 2
    self.matriz[pos_y_p3,pos_x_p3] = 3
    self.matriz[pos_y_p4,pos_x_p4] = 4
```

```
  self.pos_jugador = (pos_y,pos_x)
  self.pos_jugador_segundo = (pos_y_p2,pos_x_p2)
  self.pos_jugador_tercero = (pos_y_p3,pos_x_p3)
  self.pos_jugador_cuarto = (pos_y_p4,pos_x_p4)

  return self.matriz

def Dibujar_Tablero(self):
  plt.figure(figsize=(np.size(self.matriz,0),np.size(self.matriz,1)))
  sns.heatmap(self.matriz, linewidths=.1, linecolor='white',annot=False, cmap='magma', yticklabels=False,xticklabels=False, cbar=False, square=True);
  sns.heatmap(self.matriz, linewidths=.1, linecolor='white',annot=True, cmap='magma', yticklabels=False,xticklabels=False, cbar=False, square=True,
              mask=np.array(self.matriz)<0);
  plt.show()

def get_pos(self):
  return self.pos_jugador

def get_pos_segundo(self):
  return self.pos_jugador_segundo

def get_pos_tercero(self):
  return self.pos_jugador_tercero

def get_pos_cuarto(self):
  return self.pos_jugador_cuarto

def get_tamanio(self):
  return self.tamanio

def get_matriz(self):
  return self.matriz
```



Grafo:

Se crea la clase “Grado”, la cual recibe al tablero. Luego, se crean dos listas que tienen los valores de los movimientos de los jugadores.

```
self.Movimientos_Fila = [1,-1,0,0,1,1,-1,-1]
self.Movimientos_Columna = [0,0,1,-1,-1,1,-1,1]
```

Se crea una lista de los movimientos que puede realizar el jugador.

```
self.Movimientos = list("DURLHPJK")
```

D = Abajo

U = Arriba
L = Izquierda
R = Derecha
J = Diagonal izquierda hacia arriba
K = Diagonal derecha hacia arriba
H = Diagonal izquierda hacia abajo
P = Diagonal derecha hacia abajo

Se crean matrices que serán visitadas para determinar si un jugador estuvo en esa posición o no.

```
self.visitados = [[-1 for i in range(self.mesa.get_tamanio())] for j in range(self.mesa.get_tamanio())]
self.visitados_p2 = [[-1 for i in range(self.mesa.get_tamanio())] for j in range(self.mesa.get_tamanio())]
self.visitados_p3 = [[-1 for i in range(self.mesa.get_tamanio())] for j in range(self.mesa.get_tamanio())]
self.visitados_p4 = [[-1 for i in range(self.mesa.get_tamanio())] for j in range(self.mesa.get_tamanio())]
```

Se define los valores que tienen las ubicaciones iniciales de cada jugador y el lugar al cual tiene que llegar.

```
pos = self.mesa.get_tamanio() // 2
pos_p2 = self.mesa.get_tamanio() // 2
pos_p3 = self.mesa.get_tamanio() // 2
pos_p4 = self.mesa.get_tamanio() // 2
```

Se crean funciones que tienen como finalidad asegurar que el jugador se mueva dentro del tablero y que si hay un obstáculo u otro jugador, no permita que avance y busque otra ruta. Esto quiere decir que cada jugador tendrá esta función.

```
def validacion(self,fila,columna):
  tamanio = self.mesa.get_tamanio()
  return (fila >= 0) and (fila < tamanio) and (columna >= 0) and (columna < tamanio) \
  and (self.matriz[fila][columna] != 1)  and (self.matriz[fila][columna] != 2) \
  and (self.matriz[fila][columna] != 3) and (self.matriz[fila][columna] != 4) \
  and (self.matriz[fila][columna] != 5) and (self.visitados[fila][columna] == -1) \
  and (self.visitados_p2[fila][columna] == -1)
```

Una vez ya se hayan definido las funciones de validaciones, se crea la función que trabaja las paredes. Estas se ubicaran de manera aleatoria en cada fila y bloquearán los caminos. Después se procederá a implementar la función BFS. En la cual, se llamará a la función de las paredes y colocará el valor de “-2” en las partes visitadas.

```
self.visitados[self.inicio[0]][self.inicio[1]] = -2
self.visitados_p2[self.inicio_p2[0]][self.inicio_p2[1]] = -2
self.visitados_p3[self.inicio_p3[0]][self.inicio_p3[1]] = -2
self.visitados_p4[self.inicio_p4[0]][self.inicio_p4[1]] = -2
```

Se crea la función que reconstruye el camino recorrido por el jugador. Esta se encarga de retroceder desde el punto final hasta el punto inicial. Al final, retorna la cantidad de movimientos y cuáles fueron los movimientos realizados.

```
def Reconstruir_Camino(self):
  fila,columna = self.destino_01
  while self.visitados[fila][columna] >= 0:
    i = self.visitados[fila][columna]
    self.mov_p1.appendleft(self.Movimientos[i])
    fila -= self.Movimientos_Fila[i]
    columna -= self.Movimientos_Columna[i]
  self.camino_p1 = len(self.mov_p1)
```

Se crea una función que recibe el parámetro del jugador que ganó. El motivo es porque si se muestran todos los caminos realizados por los jugadores, no se podría distinguir quién ganó y no se podría visualizar de forma legible en el tablero.

```
def Mostrar_Camino(self, ganador):
  if ganador == 1:
    fila,columna = self.destino_01
    mov = deque()
    while self.visitados[fila][columna] >= 0:
      i = self.visitados[fila][columna]
      mov.appendleft(self.Movimientos[i])
      self.matriz[fila][columna] = 1
      fila -= self.Movimientos_Fila[i]
      columna -= self.Movimientos_Columna[i]
```

Se empieza a realizar el BFS. Primero se revisa si la cola está vacía, luego se crea un “For” para que recorra los movimientos y visitamos los vecinos, llamamos a la función de validación para que compruebe que el vecino está dentro de los parámetros establecidos y si cumple, se introduce a la cola.

```
while len(cola):
  fila, columna = cola.popleft()
  for i in range(8):
    nx = fila + self.Movimientos_Fila[i]
    ny = columna + self.Movimientos_Columna[i]
    if self.validacion(nx,ny):
      self.visitados[nx][ny] = i
      cola.append((nx,ny))
```

Se crea la función “Ganador”. La cual reconstruye el camino que realizó el jugador. Cuando se llamen a los jugadores se sabrá la cantidad de movimientos que realizó cada uno y se almacenará en una lista.

```
ganador = [self.camino_p1, self.camino_p2, self.camino_p3, self.camino_p4]
minimo = min(ganador)
```

Una vez se establezca el camino mínimo del jugador ganador, se mostrará el número de movimientos que se llevó a cabo y se visualizará la ruta recorrida.

```
if minimo == self.camino_p1:
  print("\n\n----------EL GANADOR ES EL PRIMER JUGADOR--------------------")
  print("La cantidad de movimientos realizados es: ", self.camino_p1)
  self.Mostrar_Camino(1)

elif minimo == self.camino_p2:
  print("\n\n----------EL GANADOR ES EL SEGUNDO JUGADOR--------------------")
  print("La cantidad de movimientos realizados es: ", self.camino_p2)
  self.Mostrar_Camino(2)

elif minimo == self.camino_p3:
  print("\n\n----------EL GANADOR ES EL TERCER JUGADOR--------------------")
  print("La cantidad de movimientos realizados es: ", self.camino_p3)
  self.Mostrar_Camino(3)

elif minimo == self.camino_p4:
  print("\n\n----------EL GANADOR ES EL CUARTO JUGADOR--------------------")
  print("La cantidad de movimientos realizados es: ", self.camino_p4)
  self.Mostrar_Camino(4)
```

Tiempo:

Se crea la función “Tiempo” el cual nos permitirá visualizar el tiempo ejecutado del BFS y a la vez, cuando demoro un jugador en acabar el juego.

```
def Tiempo(main,tamanio):
  inicio = time.time()
  main(tamanio)
  final = time.time()
  return (final - inicio) * 1000
```

Main:

En esta función recibe el parámetro del tamaño de la tabla, el cual se le asigna a la clase tablero. También se mostrarán las posiciones iniciales de los jugadores para conocer el punto de inicio y también se visualiza la matriz inicial. Luego se llama a la clase Grafo, en donde recibirá la clase tablero, después se llama a la función BFS. Por último se llama la función Ganador. También se mostrará los movimientos realizados por cada jugador y solo se mostrará el recorrido del jugador que ganó.

```
def main(tamanio):
  mesa = Tablero(tamanio)
  mesa.Generar_Tablero()
  mesa.Dibujar_Tablero()

  bfs = Grafo(mesa)
  print("\n\n------------------DATOS GENERALES--------------------------")
  print("La posicion del primer jugador es => ", mesa.get_pos())
  print("La posicion del segundo jugador es => ", mesa.get_pos_segundo())
  print("La posicion del tercero jugador es => ", mesa.get_pos_tercero())
  print("La posicion del cuarto jugador es => ", mesa.get_pos_cuarto())
  print("El tamanio del tablero es => ", mesa.get_tamanio())
  print("El destino del primer jugador es => ", bfs.get_destino())
  print("El destino del segundo jugador es => ", bfs.get_destino_p2())
  print("El destino del tercero jugador es => ", bfs.get_destino_p3())
  print("El destino del cuarto jugador es => ", bfs.get_destino_p4())

  bfs.BFS()
  bfs.Ganador()

  print("\n\n\n------------------------------CAMINO RECORRIDO-------------------------")
  mesa.Dibujar_Tablero()
```

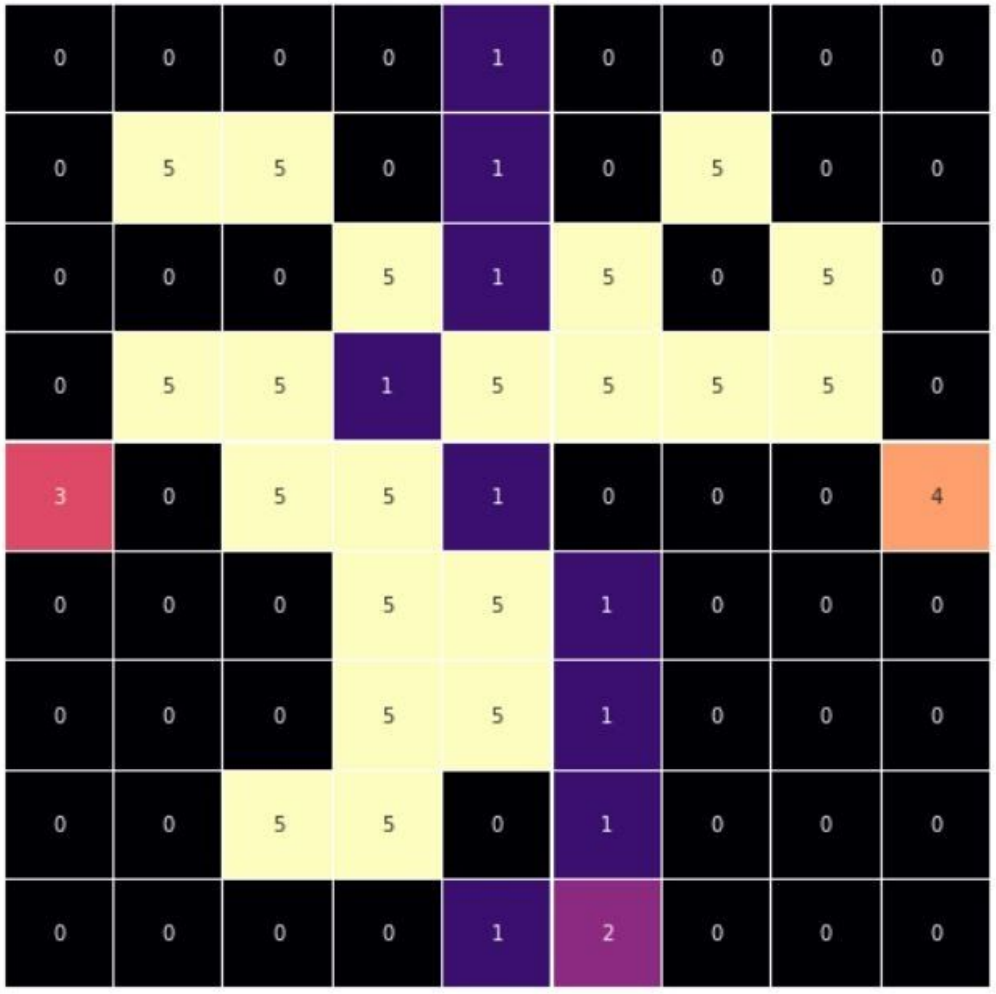
```
print("\n\nEl tiempo es => ", Tiempo(main,9), "segundos")
```

Resultados:

Resultado #1

```
------------------DATOS GENERALES----------------------------
La posicion del primer jugador es =>  (0, 4)
La posicion del segundo jugador es =>  (8, 5)
La posicion del tercero jugador es =>  (4, 0)
La posicion del cuarto jugador es =>  (4, 8)
El tamanio del tablero es =>  9
El destino del primer jugador es =>  (8, 4)
El destino del segundo jugador es =>  (0, 6)
El destino del tercero jugador es =>  (6, 8)
El destino del cuarto jugador es =>  (5, 0)


----------EL GANADOR ES EL PRIMER JUGADOR--------------------
La cantidad de movimientos realizados es:  8


----------------------MOVIMIENTOS REALIZADOS----------------------------
Movimientos realizados por el primer jugador =>  DDHPPDDH
Movimientos realizados por el segundo jugador =>  UUUJJKKK
Movimientos realizados por el tercer jugador =>  UKRPPRRPP
Movimientos realizados por el cuarto jugador =>  LHHHHLJUJ
```

```
------------------------------CAMINO RECORRIDO-----------------------------
```

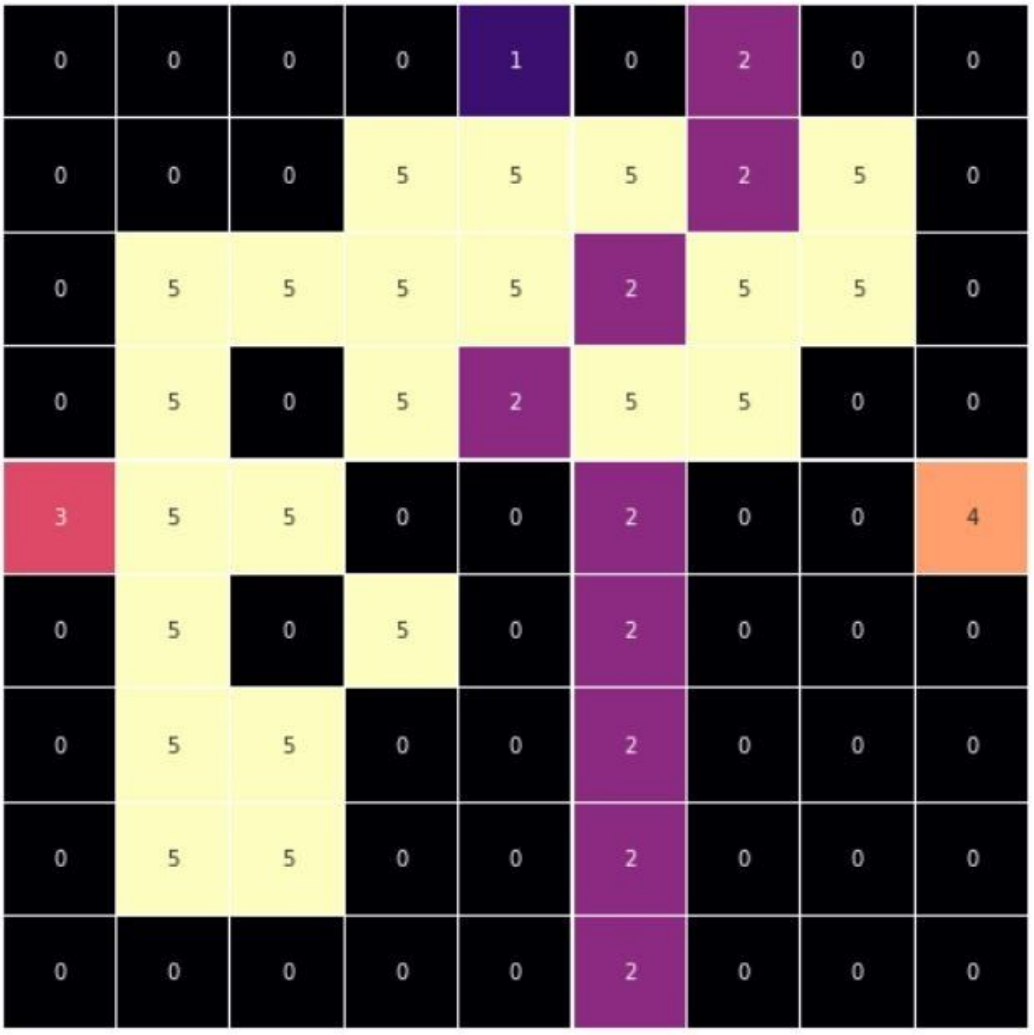



```
El tiempo es =>  576.8256187438965 segundos
```

Resultado #2

```
------------------DATOS GENERALES---------------------------
La posicion del primer jugador es =>  (0, 4)
La posicion del segundo jugador es =>  (8, 5)
La posicion del tercero jugador es =>  (4, 0)
La posicion del cuarto jugador es =>  (4, 8)
El tamanio del tablero es =>  9
El destino del primer jugador es =>  (8, 4)
El destino del segundo jugador es =>  (0, 6)
El destino del tercero jugador es =>  (6, 8)
El destino del cuarto jugador es =>  (5, 0)


----------EL GANADOR ES EL SEGUNDO JUGADOR--------------------
La cantidad de movimientos realizados es:  8


----------------------MOVIMIENTOS REALIZADOS---------------------------
Movimientos realizados por el primer jugador =>  RPHHDDDDD
Movimientos realizados por el segundo jugador =>  UUUUJKKU
Movimientos realizados por el tercer jugador =>  DDDPRRRKRRK
Movimientos realizados por el cuarto jugador =>  LLHHHHLJUU
```

```
------------------------------CAMINO RECORRIDO-----------------------------
```

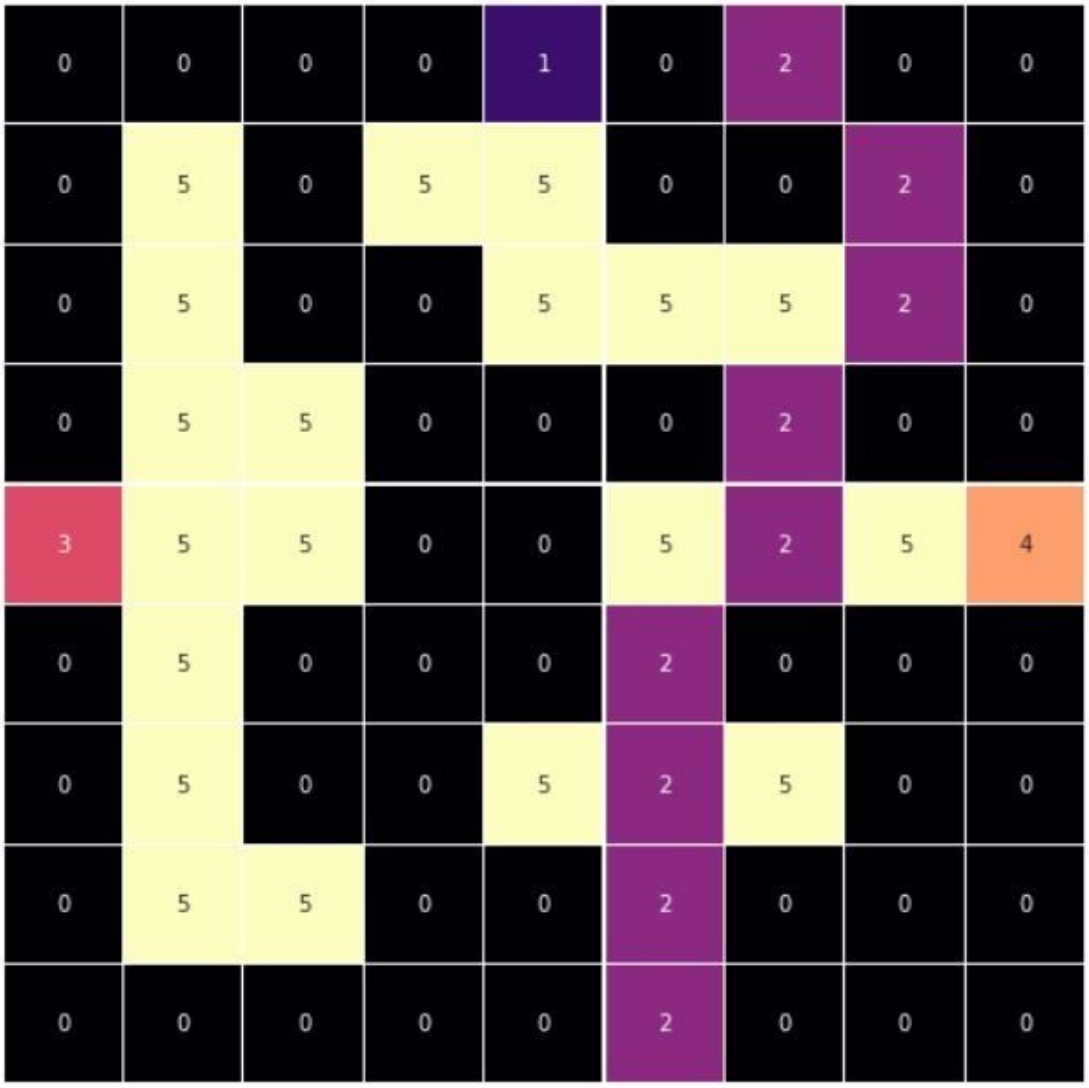



```
El tiempo es =>  562.0458126068115 segundos
```

Resultado #3

```
------------------DATOS GENERALES---------------------------
La posicion del primer jugador es =>  (0, 4)
La posicion del segundo jugador es =>  (8, 5)
La posicion del tercero jugador es =>  (4, 0)
La posicion del cuarto jugador es =>  (4, 8)
El tamanio del tablero es =>  9
El destino del primer jugador es =>  (8, 4)
El destino del segundo jugador es =>  (0, 6)
El destino del tercero jugador es =>  (6, 8)
El destino del cuarto jugador es =>  (5, 0)


----------EL GANADOR ES EL SEGUNDO JUGADOR--------------------
La cantidad de movimientos realizados es:  8


----------------------MOVIMIENTOS REALIZADOS---------------------------
Movimientos realizados por el primer jugador =>  RPPDHDHDH
Movimientos realizados por el segundo jugador =>  UUUKUKUJ
Movimientos realizados por el tercer jugador =>  DDDPRRRKRRK
Movimientos realizados por el cuarto jugador =>  HLHHLHLJUU
```

----------CAMINO RECORRIDO----------

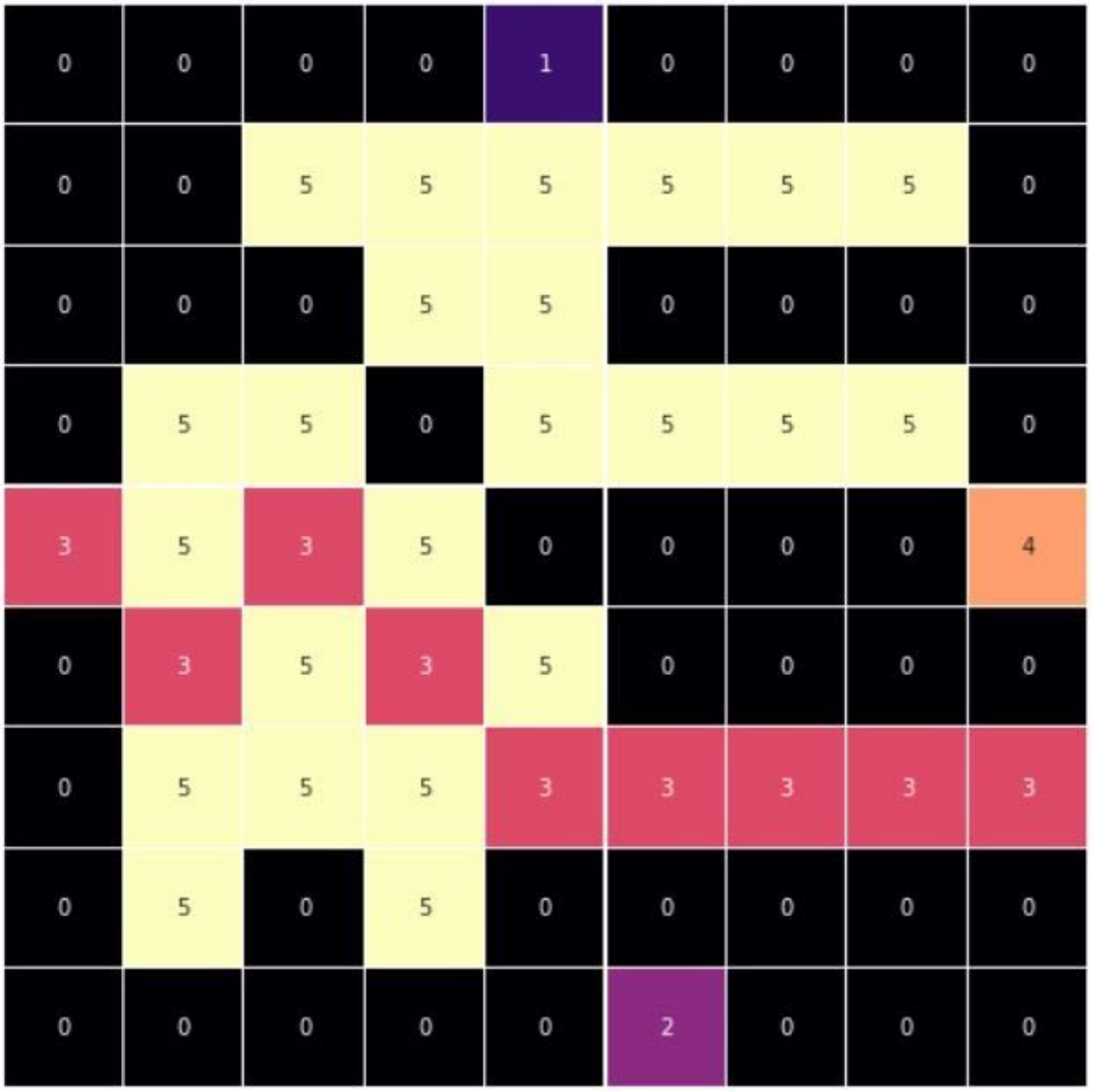


El tiempo es => 576.2994289398193 segundos

Resultado #4

```
------------------DATOS GENERALES----------------------------
La posicion del primer jugador es =>  (0, 4)
La posicion del segundo jugador es =>  (8, 5)
La posicion del tercero jugador es =>  (4, 0)
La posicion del cuarto jugador es =>  (4, 8)
El tamanio del tablero es =>  9
El destino del primer jugador es =>  (8, 4)
El destino del segundo jugador es =>  (0, 6)
El destino del tercero jugador es =>  (6, 8)
El destino del cuarto jugador es =>  (5, 0)


----------EL GANADOR ES EL TERCER JUGADOR--------------------
La cantidad de movimientos realizados es:  8


----------------------MOVIMIENTOS REALIZADOS---------------------------
Movimientos realizados por el primer jugador =>  LLHPPHPPDD
Movimientos realizados por el segundo jugador =>  UUKKKUUJL
Movimientos realizados por el tercer jugador =>  PKPPRRRR
Movimientos realizados por el cuarto jugador =>  LLLLHJHL
```

----------CAMINO RECORRIDO----------



El tiempo es => 585.4685306549072 segundos

Github:

https://github.com/HectorEgocheaga/TF-Quoridor

Conclusiones:

En conclusión, se pudo complir con los objetivos planeados porque la eficiencia del algoritmo utilizado, BFS, en el mayor de los casos tendrá un resultado muy similar, siempre y cuando no se haya realizado algún cambio en aquella función. Esto se puede evidenciar en los resultados realizados en la parte inferior donde se puede visualizar el tiempo de cada ejemplo.

También se consiguió cumplir con los objetivos por completo ya que se utilizó los algoritmos y técnicas de programación aprendidas en clase para resolver los ámbitos en los que era necesario demostrar el conocimiento aprendido.

Bibliografía

ARBOLERA, D(18 de marzo del 2016) Anaconda, suite para la Ciencia de datos con Python desde Ubuntu, Recuperado de: https://ubunlog.com/anaconda-datos-python-ubuntu/

ENSZINK, J(16 de octubre del 2019). Quoridor. Enszink Board Game: QUORIDOR. Recuperado de: https://enszink.blogspot.com/2019/10/quoridor.html

MAGNET(9 de Febrero del 2016) La muy, muy larga pero imprescindible historia de los juegos de mesa. Recuperado de: https://magnet.xataka.com/en-diez-minutos/la-larga-historia-de-los-juegos-de-mesa

GLENDENNING, L(2005). Mastering Quoridor(Tesis Bachillerato de Ciencias de la Computación). The University of New Mexico. Albuquerque, New Mexico. Recuperado de: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.100.5204&rep=rep1&type=pdf

MESSAGUÉ, V & BROWN, J(2018). Monte-Carlo-Tree-Search-for-Quoridor. Innopolis University. Kazan, Tatarstan, Russia. Recuperado de: https://www.researchgate.net/profile/Joseph_Brown7/publication/327679826_Monte_Carlo_Tree_Search_for_Quoridor/links/5b9e5b7692851ca9ed0f6eb9/Monte-Carlo-Tree-Search-for-Quoridor.pdf

MESSAGUÉ, V & BROWN, J(2018). Monte-Carlo-Tree-Search-for-Quoridor. Innopolis University. Kazan, Tatarstan, Russia. Recuperado de: https://upcommons.upc.edu/bitstream/handle/2117/127676/131875.pdf?sequence=1&isAllowed=y

MERTENS, P(21 de junio del 2006). A Quoridor-Playing Agent. Recuperado de: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.134.5605&rep=rep1&type=pdf

LUTZ, M (2001). Programming Python. Recuperado de: https://books.google.com.pe/books?hl=en&lr=&id=c8pV-TzyfBUC&oi=fnd&pg=PR11&dq=python&ots=n53A7NUXSS&sig=3fk9iG5kxmIaOB3Vb4KbuHTeXx0&redir_esc=y#v=onepage&q=python&f=false

BORGE, F(14 de agosto del 2011). Introducción a la librería NumPy de Python - Parte 1. Recuperado de: https://aprendeia.com/introduccion-a-numpy-python-1/

SÁNCHEZ, R(20 de marzo del 2015). Visualización de datos en Python con Seaborn. Recuperado de: https://www.analyticslane.com/2018/07/20/visualizacion-de-datos-con-seaborn/#:~:text=Seaborn%20es%20una%20librer%C3%ADa%20para,defecto%20en%20la%20distribuci%C3%B3n%20Anaconda.

TRELLES, S(5 de febrero del 2012). Introducción a la librería Matplotlib de Python. Recuperado de: https://aprendeia.com/libreria-pandas-de-matplotlib-tutorial/