Creación de inteligencia artificial que completa el juego de snake a través del algoritmo de Dijkstra

Herramientas computacionales
Grupo 5

Integrantes:

Andrés Felipe Vargas
Alejandro Ezequiel Celis
Santiago Andrés Acosta Díaz

---

1. Descripción del problema y motivación

Desde el nacimiento de los videojuegos se han empleado varias técnicas y herramientas para la diversificación del entretenimiento, historia y jugabilidad de los mismos. Dentro de estas herramientas, las llamadas IA (Inteligencias Artificiales) han jugado un papel importante, haciendo que el desarrollo de dicho campo se vea perfeccionado a un punto inimaginable.

Es a partir de esto y nuestra cuasi-eterna pasión por la programación, algoritmos y videojuegos, que hemos decidido implementar nuestra propia inteligencia artificial, remontándonos a la era de los juegos arcade que nacieron en el siglo pasado.

La elección popular apeló al famoso videojuego llamado “snake”. En el juego, *el jugador o usuario controla una larga y delgada criatura, semejante a una serpiente, que vaga alrededor de un plano delimitado, recogiendo alimentos (o algún otro elemento), tratando de evitar golpearse contra su propia cola o las "paredes" que rodean el área de juego. Cada vez que la serpiente se come un pedazo de comida, la cola crece más, provocando que aumente la dificultad del juego. El usuario controla la dirección de la cabeza de la serpiente (arriba, abajo, izquierda o derecha) y el*

cuerpo de la serpiente la sigue. Además, el jugador no puede detener el movimiento de la serpiente, mientras que el juego está en marcha.

A partir de lo anterior, se buscó implementar un algoritmo de tal forma que jugara e intentara completar el videojuego de forma automática.

2. Metodología

Preliminares:

Se requiere tener instalado python en el sistema que ejecutará el juego. En caso de no tenerlo, sólo hace falta descargar el instalador desde la página https://www.python.org/downloads/ y ejecutarlo en el sistema.

El código utiliza dos librerías de python que no se encuentran por defecto al instalar python, las cuales son *Turtle* y *pygame*. En caso de no tenerlas instaladas sólo se necesita correr un comando en la consola por cada una:

Instalar pygame:

```
pip install pygame
```

Instalar Turtle:

```
pip install turtle
```

Teniendo estas librerías instaladas se puede correr el código abriendo la terminal en la carpeta donde se encuentra alojado, revisando que se encuentren los archivos snake.py, util.py, beep.wav y diomedes.mp3. Para ejecutar el juego se utilizan los siguientes comandos:

```
py snake.py
python snake.py
```

Dependiendo la versión de python instalada.

Algoritmo de Dijkstra

El algoritmo de Dijkstra, también llamado algoritmo de caminos mínimos, es un algoritmo para la determinación del camino más corto, dado un vértice origen, hacia el resto de los vértices en un grafo que tiene pesos (distancias) en cada arista.

Pasos del algoritmo

Inicialización:

Sea V un conjunto de vértices de un grafo.

Sea C una matriz de costos de las aristas del grafo, donde en C[u,v] se almacena el costo de la arista entre u y v.

Sea S un conjunto que contendrá los vértices para los cuales ya se tiene determinado el camino mínimo.

Sea D un arreglo unidimensional tal que D[v] es el costo del camino mínimo del vértice origen al vértice v.

Sea P un arreglo unidimensional tal que P[v] es el vértice predecesor de v en el camino mínimo que se tiene construido.

Sea v_inicial el vértice origen.

Paso 1. *S ← {v_inicial} //Inicialmente S contendrá el vértice //origen*
Paso 2. *Para cada v∈V, v ≠ v_inicial, hacer*
2.1. D[v] ← C[v_inicial, v] //Inicialmente el costo del //camino mínimo de v_inicial a v es lo contenido en //la matriz de costos
2.2. P[v] ← v_inicial //Inicialmente, el //predecesor de v en el camino mínimo construido //hasta el momento es v_inicial
Paso 3. *Mientras (V – S ≠ ∅) hacer //Mientras existan vértices para //los cuales no se ha determinado el //camino mínimo*
3.1. Elegir un vértice w∈(V-S) tal que D[w] sea el mínimo.
3.2. S ← S ∪ {w} //Se agrega w al conjunto S, pues ya se //tiene el camino mínimo hacia w
3.3. Para cada v∈(V-S) hacer
3.3.1. D[v] ← min(D[v],D[w]+C[w,v]) //Se escoge, entre //el camino mínimo hacia v que se tiene //hasta el momento, y el camino hacia v //pasando por w mediante su camino mínimo, //el de menor costo.
3.3.2. Si min(D[v],D[w]+C[w,v]) = D[w]+C[w,v] entonces P[v] ← w //Si se escoge ir por w entonces //el predecesor de v por el momento es w
Paso 4. *Fin*

FUNCIONES IMPLEMENTADAS:

El funcionamiento de la IA se puede dividir en dos partes generales:

1. La discretización del mapa de juego.
2. La implementación del algoritmo de Djriashka.

Discretización del mundo:

Este bloque son funciones que permiten representar el juego y sus elementos como una matriz nxn. Es necesario para la implementación de la IA al necesitar esta una estructura sólida que represente eficientemente el juego.

Esta parte se puede apreciar en las líneas 151 a la línea 250 del código, en donde se implementan las siguientes funciones:

1. inicializar_matriz(n)

   Esta función recibe como parámetro un entero *n* el cual representa el largo y ancho del entorno de juego. Retorna una matriz vacía en forma de una lista de listas con elementos "-" (que representan un espacio vacío) de tamaño nxn.

   Hace uso de dos bucles for que iteran desde 0 a n, el primero representando las columnas de la matriz y el segundo las filas, llenando cada lista de filas con el elemento vacío "-" y agregando a la matriz principal la lista recién creada.

```python
# Función que devuelve una matriz del tamaño del tablero de juego
def inicializar_matriz(n):

    # matriz estado de juego
    game_state = list()

    # por cada fila (total de n) creo una columna con n espacios
    for i in range(n):

        row = list()

        for j in range(n):

            row.append(EMP)

        game_state.append(row)

    return game_state
```

2. in_matriz(obj)

   Esta función recibe como parámetro un objeto de la clase *Turtle*. Retorna una tupla ordenada que representa las coordenadas del objeto como índices de la matriz. El juego al haber sido implementado con un sistema de coordenadas escalado y trasladado con respecto a la matriz, se toman ambas coordenadas del objeto *Turtle* y se le suma el desfase que tienen, para luego ser dividido por el tamaño de cada celda del juego, devolviendo así una tupla con los respectivos índices de la matriz que representan la posición del objeto.

```python
# Función que me devuelve el índice de la matriz dependiendo las coordenadas
def in_matriz(obj):
    return (int((obj.ycor() + 300) / 20), int((obj.xcor() + 300) / 20))
```

3. discretizar_mundo(cabeza, snake, comida):

   Esta función recibe como parámetros un objeto *Turtle* cabeza, que es la cabeza de la serpiente; una lista de objetos *Turtle* snake, que son los segmentos del cuerpo de la serpiente; y un objeto *Turtle* comida, que es la comida de la

serpiente. Retorna una matriz nxn en donde están representados los diferentes elementos del juego con símbolos, “O” para la cabeza, “=” para el cuerpo de la serpiente, “%” para la comida y “-” para celdas vacías; las coordenadas de la cabeza y las coordenadas de la comida como tuplas.

Empieza llamando a la función *inicializar_matriz()* para obtener una matriz vacía nxn, y luego, para cada la cabeza, comida y cada elemento de la lista snake obtiene sus coordenadas llamando a la función *in_matriz()* y sobreescribiendo el respectivo símbolo en los índices correspondientes de la matriz.

Esta función también tiene en cuenta el teletransporte que es capaz de hacer la serpiente, que son los bloques de las líneas 194 a 199 y 217 a 224.

```
188 def discretizar_mundo(cabeza, snake, comida):
189
190     game_state = inicializar_matriz(tamaño) #Tamaño de la matriz (Tamaño del area donde la serpiente se moverá)
191
192     cab_coords = in_matriz(cabeza)
193
194     if (cab_coords[0] > 29):
195         cabeza_x, cabeza_y = (29, cab_coords[1])
196     elif (cab_coords[1] > 29):
197         cabeza_x, cabeza_y = (cab_coords[0], 29)
Remote Explorer lse:
199         cabeza_x, cabeza_y = cab_coords
200
201     comida_x, comida_y = in_matriz(comida)
202
203     #print(f"Cabeza coords: {cabeza_x, cabeza_y}")
204     #print(f"Comida coords: {comida_x, comida_y}")
205
206     game_state[cabeza_x][cabeza_y] = CAB
207     game_state[comida_x][comida_y] = COM
208
209     # hago lo mismo para cada segmento del cuerpo
210     for segmento in snake:
211
212         segmento_coords = in_matriz(segmento)
213
214         if (segmento_coords[0] == cabeza_x) and (segmento_coords[1] == cabeza_y):
215             continue
216
217         if (segmento_coords[0] > 29):
218             segmento_x, segmento_y = (29, segmento_coords[1])
219         elif (segmento_coords[1] > 29):
220             segmento_x, segmento_y = (segmento_coords[0], 29)
221         else:
222             segmento_x, segmento_y = segmento_coords
223
224         game_state[segmento_x][segmento_y] = CUE
225
226     return (game_state[::-1], (cabeza_x, cabeza_y), (comida_x, comida_y))
```

Es importante notar que la matriz se reversa al momento de retornar, ésto se debe a que el sistemas de coordenadas utilizado fue (vertical, horizontal).

4. corpse(GameState)

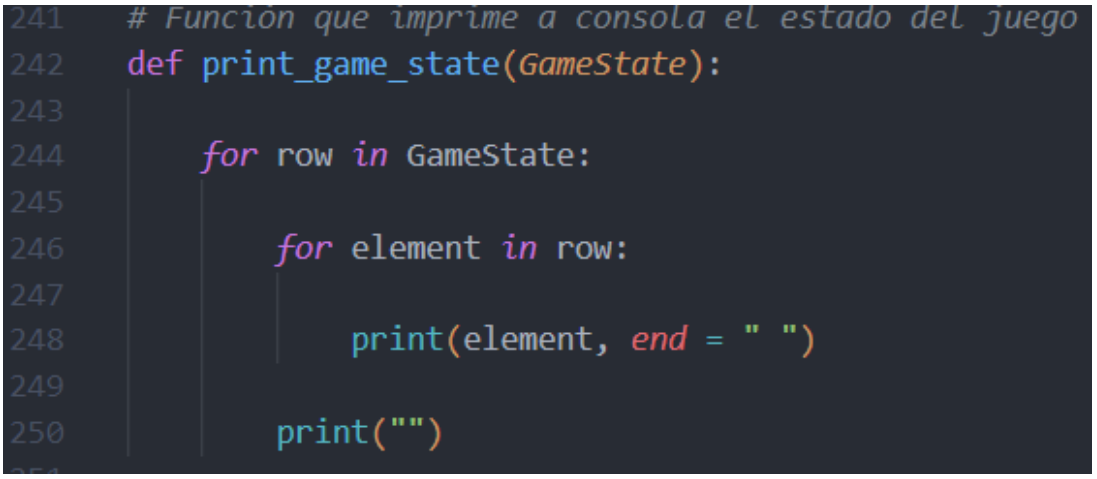
   Esta función recibe como parámetros la matriz del estado del juego. Retorna un set que contiene tuplas con las coordenadas de cada segmento del cuerpo de la serpiente. Esta función se utiliza en la implementación de la IA para hacer que el algoritmo de búsqueda de caminos tenga en cuenta el cuerpo de la serpiente para evitar que esta colapse.

   Esta función itera por cada elemento y su índice de la matriz del estado del juego, y aquellas entradas cuyo símbolo represente el cuerpo de la serpiente, son agregadas al set *corpse_segments()* en forma de una tupla de coordenadas.

```
228  def corpse(GameState):
229
230      corpse_segments = set()
231
232      for cIn, column in enumerate(GameState):
233          for rIn, row in enumerate(column):
234
235              if GameState[cIn][rIn] == CUE:
236                  corpse_segments.add((29 - cIn, rIn))
237
238      #print(f"Corpse: {corpse_segments}")
239      return corpse_segments
```

5. print_game_sate(GameState)

   Esta función recibe como parámetros la matriz del estado del juego y no tiene retorno. Imprime la matriz a consola. Es una función de debugación, no afecta el resto del código.



```
241  # Función que imprime a consola el estado del juego
242  def print_game_state(GameState):
243
244      for row in GameState:
245
246          for element in row:
247
248              print(element, end = " ")
249
250          print("")
```

IA, algoritmo de búsqueda del camino más corto.

Este bloque está dedicado a la implementación del algoritmo de Dijkstra, el cual busca el camino más corto que puede seguir la serpiente hacia su comida.

Esta parte se puede ver en las líneas 253 a la línea 422 y consta de las siguientes funciones:

1. tele_ia(coords)

   Esta función toma como parámetros una tupla de coordenadas representando una celda del tablero de juego, y retorna una lista de tuplas de coordenadas que representan las celdas adyacentes a la celda input.

   Esta función crea una lista que es llenada con tuplas de enteros, las cuales son las coordenadas de la celda de input más las tupla (-1, 0), (1, 0), (0, -1), (0, 1). Así mismo, la serpiente al poder teletransportar, se revisa si la celda input se encuentra en alguno de los bordes del mapa y con ello se hace el respectivo cálculo para tomar la celda opuesta del mapa, permitiendo el cálculo de caminos que usen la característica del teletransporte.

```python
# Funciòn que permite el teletransporte
def tele_ia(coords):

    cells = list()

    # Borde izquierdo
    if coords[0] == 0:
        cells.append((29, coords[1]))
    else:
        cells.append((coords[0] - 1, coords[1]))

    # Borde derecho
    if coords[0] == 29:
        cells.append((0, coords[1]))
    else:
        cells.append((coords[0] + 1, coords[1]))

    # Borde superior
    if coords[1] == 29:
        cells.append( (coords[0], 0) )
    else:
        cells.append((coords[0], coords[1] + 1))

    # Borde inferior
    if coords[1] == 0:
        cells.append((coords[0], 29))
    else:
        cells.append((coords[0], coords[1] - 1))


    return cells[::][::-1]
```

Notar que al devolverse la lista, se reversa cada uno de sus elementos, esto gracias al sistema de coordenadas que se está utilizando, que no es el usual.

2. cell_neighbors(GameState, node, explored, corpse)

   Esta función recibe como parámetros una tupla de coordenadas node que representa la celda que se está explorando, un set de tuplas de coordenadas explored que representa las celdas ya exploradas y un set de tuplas coordenadas corpse, que representa los segmentos del cuerpo de la serpiente.

   Esta función crea un set que será llenado con tuplas de coordenadas que representan las celdas a las que se puede mover a partir de la celda input. Llama a la función *tele_ia()* para obtener las celdas adyacentes a la celda input y luego itera por cada una de estas celdas comprobando que sea una celda apta para explorar, es decir, que no haga parte del cuerpo de la serpiente y que no se haya

explorado anteriormente, si estas condiciones se cumplen, se agrega esta celda al set y luego se devuelve este set de tuplas de coordenadas.

```python
# Función que me devuelve los posibles movimientos dada una celda como cabeza
def cell_neighbors(node, explored, corpse):

    neighbors = list()

    # node = node[::-1]

    instant_cells = tele_ia(node)

    #print(f"Explored set: {explored}")

    for cell in instant_cells:

        if (cell not in corpse) and (cell not in explored):
            neighbors.append(cell)
            explored.add(cell)


    return neighbors
```

3. path(GameState, cab_coords, com_coords)

Esta función recibe como parámetros la matriz del estado de juego GameState, las coordenadas de la cabeza cab_coords y las coordenadas de la comida com_coords. Retorna una lista de tuplas de coordenadas que representan el camino que seguirá la serpiente para llegar a la comida.

Esta función hace uso las clases *Nodo* y *PathFinder* que se encuentran en el módulo *util.py.* Las cuales son las siguientes:

3.1 Class Nodo()

Esta clase representa un nodo en la búsqueda del camino. Tiene dos atributos, *coords,* una tupla con las respectivas coordenadas de la celda que este nodo representa; y *parent,* un puntero que apunta hacia el nodo previo.

```python
class Nodo():
    def __init__(self, coords, parent):
        self.coords = coords
        self.parent = parent
```

3.2 Class PathFinder()

Esta clase tiene un sólo atributo *path,* el cual es una lista que representará la fila de celdas a explorar. Es importante mencionar que es una fila del tipo *first in first out,* que por consiguiente permitirá una búsqueda radial que encontrará el camino más corto.

Esta clase tiene tres métodos: add(node), la cual agrega un objeto de la clase Nodo a path; empty(), la cual retorna True o False dependiendo si la fila se encuentra vacía; y remove(), la cual retorna el primer nodo de la lista path y elimina este primer índice de la lista.

```
6   class PathFinder():
7       def __init__(self):
8           self.path = []
9
10      def add(self, node):
11          self.path.append(node)
12
13      def empty(self):
14          return len(self.path) == 0
15
16      def remove(self):
17          if self.empty():
18              raise Exception("empty frontier")
19          else:
20              node = self.path[0]
21              self.path = self.path[1:]
22              return node
```

Esta función empieza creando un nodo primordial, aquel que representa la cabeza de la serpiente, y luego crea un objeto PathFinder, que será la fila en donde se almacenarán los nodos a explorar. Se obtienen las coordenadas de los segmentos de la serpiente con la función *corpse()* y luego empieza el bucle de búsqueda.

En el bloque de búsqueda se revisa que la fila no esté vacía, y si lo está, esto significa que no se pudo encontrar ningún camino que conecte la cabeza con la comida, es decir, que la serpiente se encerró en su propio cuerpo. Si esto no ocurre, el siguiente paso es tomar el primer nodo en la fila usando el método *remove()* de la clase PathFinder, se agregan las coordenadas de ese nodo al set explored y se revisa si esas coordenadas coinciden con las coordenadas de la comida, si ese es el caso, significa que ya se encontró el camino a seguir, por lo que el paso a seguir es crear una lista donde se almacenan las coordenadas de todos los nodos padre del nodo encontrado, lo cual construye el camino reverso, por lo que se retorna la lista reversa; en tal caso que las coordenadas del nodo no coincidan con la comida, se obtienen las celdas libres adyacentes al nodo explorado y se itera por cada una de ellas, agregándolas ordenadamente a la fila de la instancia de PathFinder.

```python
# Función que implementa la el algoritmo de Djistkra (no sé cómo se escribe xD)
def path(GameState, cab_coords, com_coords):

    start = Nodo(cab_coords, None)  # El nodo inicial es la cabeza del snake
    goal = com_coords               # La meta es la celda que contiene la comida

    pf = PathFinder()   # pf abreviación de PathFinder
    pf.add(start)       # Agregamos el nodo inicial al PathFinder


    explored = set()    # Set en donde guardaremos las celdas ya explorados
                        # para optimizar la búsqueda

    corpse_segments = corpse(GameState)
    #print(f"corpse: {corpse_segments}")
    # input()

    # Se empieza la búsqueda del path
    while True:

        # Si no hay nada en el path finder devolvemos None
        if (pf.empty()):
            return None

        # Tomamos un nodo de la fila
        node = pf.remove()

        explored.add(node.coords)

        # print(f"Explored cell:{node.coords}")

        # Si el nodo el la meta, devolvemos el path que llegó a él
        if (node.coords == goal):

            path = list()

            # Construimos el path
            while (node.parent != None):

                path.append(node.coords)
                node = node.parent

            print("Find path")
            print(f"head: {cab_coords}")
            return path[::-1]

        # Si no hemos llegado a la meta, expandimos los path
        else:

            # Exploramos cada una de los posibles caminos por turno
            for cell in cell_neighbors(node.coords, explored, corpse_segments):
                # Agregamos al PathFinder ese nodo para que sea explorado
                # En siguientes iteraciones
                #print(f"Added neigbor: {cell}")
                pf.add(Nodo(cell, node))
```

4. IA_MOV(cab_coords, cell_to_mov)

   Esta función recibe como parámetros dos tuplas de coordenadas cab_coords y cell_to_move, que representan la posición de la cabeza y la posición de la celda a la que esta se debe mover en la siguiente iteración, y un objeto *Turtle*, la cabeza de la serpiente, la cual modifica su atributo .direction para hacer que la IA tome las decisiones de su movimiento. No tiene retorno.

```python
# Función que devuelve el movimiento a seguir por el snaje
def IA_mov(cab_coords, cell_to_move):

    y_mov = cell_to_move[0] - cab_coords[0]
    x_mov = cell_to_move[1] - cab_coords[1]

    print(f"x move: {x_mov}")
    print(f"y move: {y_mov}")

    direction = None

    if (x_mov == 1) or (x_mov == -29):
        direction = "right"
    if (x_mov == -1) or (x_mov == 29):
        direction = "left"
    if (y_mov == -1) or (y_mov == 29):
        direction = "down"
    if (y_mov == 1) or (y_mov == -29):
        direction = "up"

    return direction
```

5. IA(game_state, cab_coors, com_coors, cabeza)

   Esta función recibe como parámetros la matriz del estado del jeugo game_state, y las tuplas de coordenadas de la cabeza y la comida, cab_coords y com_coords respectivamente.

   Esta función es un vestigio del proceso de depuración. Llama a la función *path()* para obtener el camino a seguir y lo devuelve.

```python
# Función que encuentra el path y lo retorna
def IA(game_state, cab_coors, com_coors, cabeza):

    #print(f"Head coords: [{cab_coors[0]}, {cab_coors[1]}]")
    #print(f"Food coords: [{com_coors[0]}, {com_coors[1]}]")

    path_to_take = path(game_state, cab_coors, com_coors)

    #if (path_to_take == None):
        #print(f"No path found \n Head: {cab_coors} \n Food: {com_coors}")

    #print(f"path found: {path_to_take}")

    return path_to_take
```

6. IA_make_move(path, head)

   Esta función recibe como parámetros el camino a seguir de la serpiente path, y las coordenadas de la cabeza cab_coords. Retorna el camino a seguir sin el primer elemento si aún hay celdas por recorrer o False en el caso contrario.

   La función llama a *IA_mov()* para obtener la dirección a la que se debe mover la serpiente, y como herramienta de control, en caso de no ser devuelto ningún movimiento, la serpiente ejecutará el último movimiento que hizo.

```
409  # Función que hace el siguiente movimiento del path
410  def IA_make_move(path, cab_coords):
411
412      global prev_mov
413
414      movToMake = [IA_mov(cab_coors, path[0]) if (path != [] and path != None) else prev_mov][0]
415
416      prev_mov = movToMake
417
418      cabeza.direction = movToMake
419
420      #print(f"Move to make: {movToMake}")
421
422      return (path[1::] if (path != None or path != []) else False)
```

7. Bucle en el main loop del juego

   Este bucle se encarga de corroborar que la IA sólo tome un camino por cada comida que aparece, de esta forma optimiza la búsqueda.

   Se tiene un valor booleano IA_SEARCH_PATH, el cual decide cuándo la IA debe buscar un nuevo camino. En caso que deba, se guarda el path que retorna la función *IA()* y se niega el valor booleano para evitar una futura evaluación. La variable *path_flag* guarda el retorno de IA_make_move(), el cual es el path sin el primer elemento o False en caso que se haya completado el camino. Luego, dependiendo del valor de *path_flag*, se acepta el valor de *IA_SEARCH_PATH* o se guarda el path modificado en *IA_FOUND_PATH*. Por último, se hace el movimiento de la serpiente.

3. Descripción de pruebas

Las pruebas se dieron en diferentes fases, cada una representando un error ocurrido durante la implementación del juego.

1. Primera fase: desarrollo del juego base

   En esta fase se implementó el código base del juego, en donde la serpiente se movía utilizando input del teclado. Las pruebas consistían en asegurar que el funcionamiento del juego fuera el correcto, realizándose los siguientes ítems:

   - La serpiente se mueve correctamente
   - La serpiente interactúa con la comida correctamente
   - La serpiente interactúa con su cuerpo correctamente
   - El teletransporte funciona correctamente

2. Segunda fase: discretización del juego

   En esta fase se implementó el primer boceto de la IA. Lo que se buscaba con esto era la correcta discretización del mundo, teniendo en cuenta los siguientes ítems:

   - La matriz del estado del juego (MEJ) tiene el tamaño correcto
   - Aparecen representados los elementos del juego en la MEJ.
   - Los elementos del juego aparecen en el lugar correcto.
   - La MEJ se actualiza de forma correcta antes de llamar a la IA.

3. Tercera fase: primer boceto de la IA

   En esta fase se implementó la base de la IA, específicamente, el algoritmo de búsqueda de caminos. Los ítems a revisar fueron:

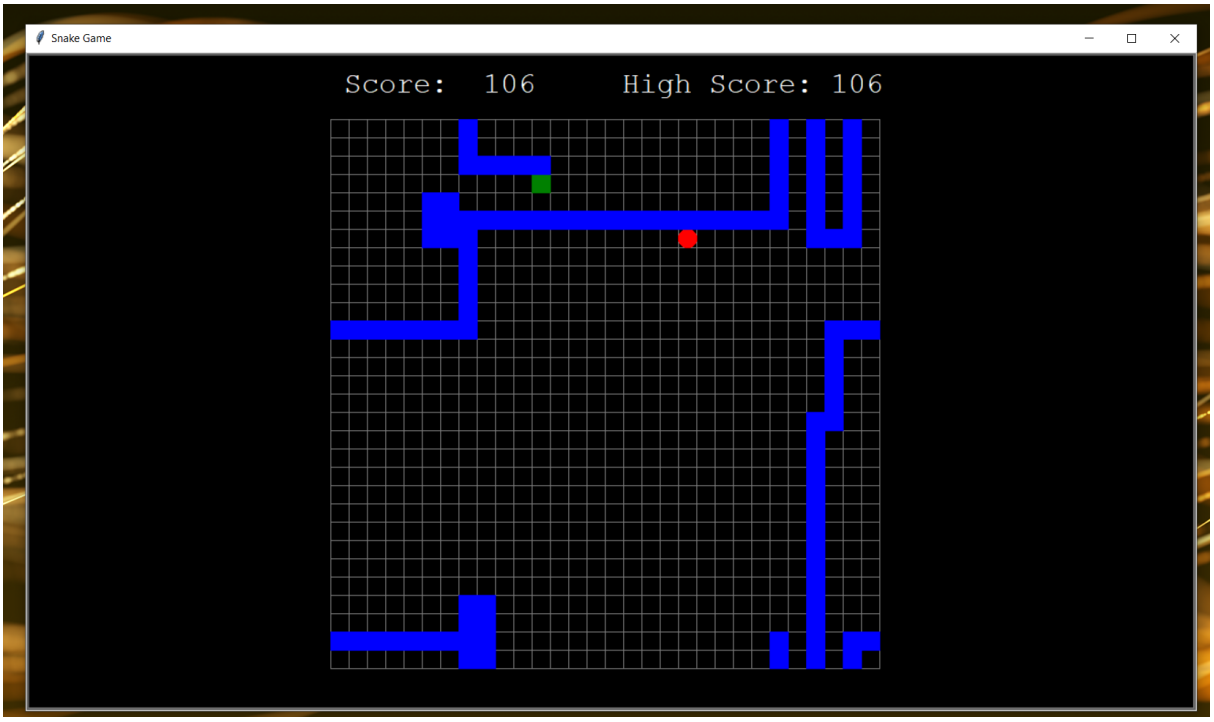
   - La fila se comportaba como una fila FIFO
   - La función *path()* reconoce cuándo terminar la búsqueda
   - El algoritmo converge a un único camino.
   - El algoritmo es eficiente.

   En esta fase no se tiene en cuenta el cuerpo de la serpiente, por lo que ella chocaba consigo misma.

4. Cuarta fase: implementación de la IA
   En esta fase se buscó completar el algoritmo de la IA, haciendo que la búsqueda del camino tuviese en cuenta el cuerpo de la serpiente, la teletransportación en los bordes y la eficiencia de búsqueda. Revisando los siguientes ítems:

- La IA no explora celdas ya exploradas, evitando entrar en bucle
- La IA no toma celdas ocupadas por el cuerpo de la serpiente
- La IA converge a un único camino

4. CONCLUSIONES



Se obtuvo un resultado parcialmente favorable, puesto que la IA es capaz de avanzar y moverse inteligentemente por el mapa, buscando siempre los caminos más eficientes y haciendo uso de la teletransportación.

ERRORES Y FALLAS:

Un error que no se pudo solucionar fueron escenarios donde la serpiente se encerraba con su propio cuerpo al ya tener una longitud bastante grande (generalmente mayor a 120 segmentos).

REPOSITORIO DE GITHUB

https://github.com/acelis-hub/Snake

REFERENCIAS

- https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
- https://www.youtube.com/watch?v=GazC3A4OQTE
- https://www.youtube.com/watch?v=BCpKJrGHBJA
- https://docs.python.org/3/library/turtle.htm
- https://www.pygame.org/news