Tetris: ¿El juego infinito?

Juan Ignacio Jiménez Gutiérrez
dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
juajimgui1@alum.us.es juan12_rubio@hotmail.com

Carlos Jesús García Borrego
dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
cargarbor@alum.us.es garciaborrego.carlos@gmail.com

Nuestro objetivo principal será el desarrollo de un asistente en lenguaje Python [1] por el cual simulemos el comportamiento del reconocido juego de Tetris [2].

Mediante algoritmos de espacio de estados [3], probaremos distintas estrategias para alcanzar desde una posición determinada de un Tetrominó [4], hasta la mejor posición posible dado una situación particular del mapa de juego.

Los métodos resultantes de nuestro desarrollo nos permiten introducir una pieza con una posición inicial predefinida y simular el comportamiento del juego original.

Vamos a utilizar el entorno interactivo web Jupyter Notebook para el desarrollo y ejecución del código desarrollado.

Python, Tetris, tetrominó, espacio de estados.

I. Introducción

Todo el mundo conoce el Tetris como juego, pero nuestro primer impulso cuando se nos encomendó este desarrollo fue preguntarnos, ¿De dónde procede? ¿Cómo funciona realmente este juego? ¿Dónde está su complejidad? Pues bien, originariamente el Tetris fue diseñado y programado por Alekséi Pàzhitnov [5]. Fue lanzado el l6 de junio de 1984 en la Unión Soviética.

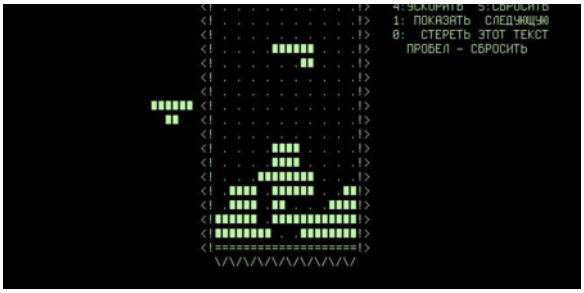

Fig 1. Imagen de Tetris obtenida en https://www.xataka.com/videojuegos/tetris-cumple-30-anos-y-sigue-como-el-primer-dia-imagen-de-la-semana

Consiste en la caída y posicionamiento de una serie de piezas llamadas tetrominós que se van apilando en una pantalla vertical de 20 filas por 10 columnas. Cada fila ocupada en su totalidad por estas piezas se eliminará y las que están por encima caerán ocupando la anterior. Existen 7 tipos distintos de tetrominós, compuestos a su vez por 4 bloques cuadrados unidos de forma ortogonal. Durante la caída de las piezas el jugador tiene la posibilidad de rotarlas 90º y desplazarlas a izquierda y derecha. El juego acaba cuando las piezas se acumulan llegando hasta arriba de la ventana, superando la altura de las 20 filas.

Una vez comprendemos su funcionamiento, enfocamos el desarrollo de este juego como un problema de espacio de estados definiendo cada acción posible con sus restricciones y efectos correspondientes. Una vez definido lo anteriormente comentado, utilizamos algoritmos de búsqueda (informada [6] y no informada [7]) para obtener los distintos resultados que analizaremos en los siguientes apartados.

Siguiendo con la estructura de los problemas de espacio de estados, las acciones que implementaremos serán: desplazamiento a la izquierda, desplazamiento a la derecha, desplazamiento hacia abajo, giro a la derecha y giro a la izquierda. Estas acciones tendrán sus respectivas restricciones, como, por ejemplo, que la pieza no pueda salir del mapa o la colisión con otras piezas ya colocadas.

Trataremos cada pieza como una agrupación de 4 celdas con coordenadas (fila, columna) que se irán introduciendo en la parte central superior del mapa.

Utilizaremos una serie de métodos auxiliares para el borrado de línea, generar piezas aleatorias, introducirlas en el mapa, etc.

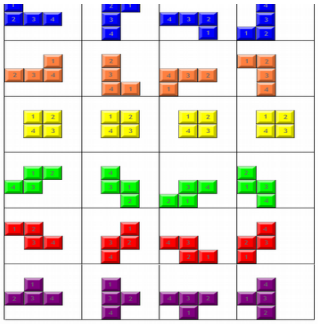

Fig 2. Imagen de tetrominós obtenida de https://gamedevelopment.tutsplus.com/tutorials/build-an-as3-tetris-game-from-the-ground-up--active-7977

II. PRELIMINARES

Como se ha comentado anteriormente, utilizaremos algoritmos de problemas de espacios de estado y de búsqueda informada y no informada para este desarrollo.

A. Métodos empleados

- Problemas de espacio de estados: consideramos la resolución de un problema de espacio de estado como el proceso que, partiendo de una situación inicial y utilizando un conjunto de procedimientos/reglas/acciones seleccionados a priori, es capaz de explicitar el conjunto de pasos que nos llevan a una situación posterior que llamamos solución.
- Búsqueda informada: aplicar conocimiento al proceso de búsqueda para hacerlo más eficiente. El conocimiento vendra dado por una función que estima la “bondad” de los estados.
- Búsqueda no informada: no cuenta con ningún conocimiento sobre cómo llegar al objetivo.

B. Trabajo Relacionado

Durante la fase de búsqueda de documentación, encontramos referencias muy interesantes a estudios sobre el uso del Tetris como terapia para superar traumas o adicciones [8] [9].

III. METODOLOGÍA

Al empezar con el desarrollo se decidió que usaríamos una estructura de datos compuesta de una matriz 22x10 de ceros como mapa inicial, tal como mostramos a continuación.

```
mapa_inicial = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

Y que los tetrominós se definirían como un array de 4 elemento y, a su vez, cada uno de ellos contendrían una coordenada (x,y) para definir su posición. Como **[[0, 3], [1, 3], [1, 4], [1, 5]]** que corresponde a al tetrominó J.

Para el tratamiento de este mapa y los tetrominós, se han desarrollado diferentes métodos, entre los cuales se encuentra introducir_en_mapa(pieza) que, como su nombre indica, nos permite introducir el tetrominó con su posición inicial en el mapa.

```
Introducir_en_mapa:
Entradas:
        pieza (lista de 4 coordenadas (x,y)), mapa
Salidas:

Algoritmo:
        para i en rango longitud(pieza)
                fila = pieza[i][0]
                columna = pieza[i][1]
                mapa[fila][columna] = 1
```

Una vez tenemos definidos la estructura que va a seguir el mapa y los tetrominós y el método para introducir estos últimos en el mapa, nos centramos en enfocar el tetris como un problema de espacio de estados. Para ello, debemos definir las distintas acciones que van a poder llevarse a cabo un tetrominó durante el juego. Identificamos claramente 5 acciones distintas, que son: movimiento a izquierda, movimiento a derecha, caída hacia abajo, giro a izquierda y giro hacia derecha. A su vez, cada acción contara con una serie de restricciones y una vez se ejecuten contaran con una serie de efectos sobre el tetrominó. Por ejemplo, observamos el método es_aplicable(self,estado) para el movimiento a la izquierda:

```
es_aplicable movimiento izquierda
Entradas:
        estado
Salidas:
        res (boolean)
Algoritmo:
        res = false
        contador = 0
        para i en rango longitud(estado)
                si estado[i][1] > -1 y estado[i][1]-1 > -1 y mapa.celdas[estado[i][0]][estado[i][1]-1] == 0
                        contador += 1
                si contador == 4
                        res = true
        devolver res
```

En los métodos de es_aplicable se tendrán en cuenta distintas restricciones como que no sobrepase los bordes del mapa o las colisiones con piezas ya colocadas.

A su vez, nos encontramos con algoritmos más complejos para acciones como los giros. Pondremos como ejemplo el método para la rotación a la izquierda.

```
Rotacion_izq
Entradas:
        estado (pieza)
Salidas:
        posiciones_finales (pieza rotada a izquierda)
Algoritmo:
        x_menor = 99
        y_menor = 99

        para i en rango longitud(estado)
                para j en rango 2
                        si estado[i][0] < x_menor
                                x_menor = estado[i]
[0]
                        si estado[i][1] < x_menor
                                x_menor = estado[i]
[1]

        nuevaMatriz = matriz de ceros 3x3
        estadoAuxiliar = copia de estado

        para i en rango longitud(estadoAuxiliar)
                para j en rango 2
                        si j == 0
                                estadoAuxiliar[i][0] =
estadoAuxiliar[i][0] - x_menor
                        si j == 1
                                estadoAuxiliar[i][1] =
estadoAuxiliar[i][1] - y_menor


        para i en rango longitud(estadoAuxiliar)
                para j en rango
longitud(estadoAuxiliar[i])
                        fila = estadoAuxiliar[i][0]
                        columna = estadoAuxiliar[i][1]
                        nuevaMatriz[fila][columna] =
4

        rot1 = rotamos
        posiciones_finales =[[0,0],[0,0],[0,0],[0,0]]

        contador = 0
        para i en rango 3
                para j en rango 3
                        si rot1[i][j] == 4
                                var_x = i + x_menor
                                var_y = j + y_menor

        posiciones_finales[contador][0] = var_x

        posiciones_finales[contador][1] = var_y
                                contador = contador
+ 1
        devolver posiciones_finales
```

Este método será llamado dentro de la acción de girar a la izquierda, si se cumplen las restricciones de aplicabilidad. Una vez cumplidas estas condiciones, se ejecutará el método aplicar(estado).

```
Aplicar
Entradas:
        estado
Salidas:
        nuevo_estado
Algoritmo:
        nuevo_estado = copia de estado
        nuevo_estado2 = rotacion_izq(nuevo_estado)
        devolver nuevo_estado2
```

Cabe mencionar que para cada una de las acciones se ha decidido utilizar un coste de aplicar de 1.

Una vez definidos los métodos para cada una de las acciones necesarias para los problemas de espacio de estados, solo nos queda el desarrollo de los distintos problemas que se nos presentan. Este apartado lo dividiremos en 3 partes en función del problema a resolver.

1. Dado un estado de la pantalla, obtener la secuencia de movimientos para colocar la pieza desde su lugar actual hasta un lugar específico.
2. Dada una situación de la pantalla y una pieza, decidir el lugar adecuando donde colocarla y llevar a cabo el movimiento correspondiente para hacerlo.
3. Dada una partida, dada por una secuencia de piezas determinada, determinar la secuencia de movimientos para ir colocando las piezas lo mejor posible.

Ahora bien, para el **apartado 1** nos bastará con introducir la posición de una pieza como estado inicial y la posición final donde queremos colocarla como posición final. La forma final de la creación del problema será:

```
acciones = [Abajo(), Izquierda(), Derecha(), Girar_izquierda(),
Girar_derecha()]
estado_inicial = [[0, 2], [1, 2], [1, 3], [1, 4]]
estado_final1 = [[6, 4], [7, 4], [7, 5], [7, 6]]
estados_finales = [estado_final1]
Juego_Mejor_Camino =
probee.ProblemaEspacioEstados(acciones, estado_inicial,
estados_finales)
```

Hecho esto, solo nos quedaría decidir qué tipo de búsqueda queremos utilizar para resolver el problema.

Implementaremos un método para una mejor legibilidad de la solución al problema.

```
transforma_solucion
Entradas:
        solución
Salidas:
        res (string creado a partir de la solución)
Algoritmo:
        res = “”
        para palabra en solucion
                si palabra == “Abajo”
                        res+= “A ”
                        continua
                si palabra == “Izquierda”
                        res+= “I ”
                        continua
                si palabra == “Derecha”
                        res+= “D ”
                        continua
                si palabra == “Girar_izquierda”
                        res+= “G_izq ”
                        continua
                si palabra == “Girar_derecha”
                        res+= “G_der ”
        devuelve res
```

Utilizaremos como ejemplo la búsqueda en anchura y pintaremos la solución obtenida.

```
anchura = búsqee.BúsquedaEnAnchura()
solucion = anchura.buscar(Juego_Mejor_Camino)


si solucion
        pinta(transforma_solucion(solucion))
sino
        pinta(“No se encontró solución”)
```

En la sección de resultados analizaremos y las distintas soluciones a los distintos problemas que hemos desarrollado.

Para el **apartado 2** se añade un poco más de complejidad. Debemos realizar un método con el que encontrar de forma la mejor posición para el tetrominó correspondiente.

Para ello, deberemos definir una heurística [10] que nos ayude a determinar dicha posición dada una situación del mapa determinada.

Definiremos 5 métodos que nos ayudaran evaluar 5 aspectos distintos para dar valor a los distintos nodos a raíz de un estado determinado y que nos ayude a quedarnos con el mejor. Los métodos serán: numeroCeldasLlenas(mapa), maximaAlturaAlcanzada(mapa), buscarHuecos(mapa), alturaPonderada(mapa) y buscarLineas(mapa).

Definiremos el método de la heurística como:

```
Heuristica
Entradas:
        mapa
Salida:
        score
Algoritmo:
        h0 = numero de celdas llenas
        h1 = maxima altura alcanzada
        h2 = numero de huecos encontrados
        h3 = altura ponderada
        h4 = buscar lineas
        c0 = 1
        c1 = 1
        c2 = 20
        c3 = 5
        c4 = 1
        score = c0*h0 + c1*h1 + c2*h2 + c3*h3 + c4*h4
        devuelve score
```

Una vez definida la heurística, implementaremos un método que nos devuelva el valor determinado dado un mapa y una pieza en función de los parámetros definidos anteriormente.

```
Calculo_heuristica
Entradas:
        mapa, pieza
Salida:
        res
Algoritmo:
        estado = copia de pieza
        mapaPrueba = copia del mapa
        introducimos en el mapaPrueba la pieza

        res = -1
        res = heuristica del mapaPrueba
        devuelve res
```

Para cada pieza que se va a introducir en el mapa, obtendremos las distintas posiciones finales posibles y con el método de mejorPosicion(l) obtendremos la mejor posición posible dada nuestra heurística.

Una vez obtenida la mejor posición para la pieza que queremos colocar, solo nos queda volver a definir el problema y la búsqueda que queremos usar.

```
mejor_posicion
Entradas:
        pieza
Salida:
        coordenadas
Algoritmo:
        mejorValor = 999
        coordenadas = []
        para I en rango longitud(pieza)
                valor = calculo la heuristica dado un mapa
y las partes de la pieza
                si valor < mejorValor
                        mejorValor = valor
                        coordenadas = pieza[i]
        devuelve coordenadas
```

```
acciones = [Abajo(), Izquierda(), Derecha(), Girar_izquierda(),
Girar_derecha()]
estado_inicial = [[0, 4], [0, 5], [1, 3], [1, 4]]
estado_final1 = mejorPosicion(Lista)
estados_finales = [estado_final1]
Juego_Mejor_Camino =
probee.ProblemaEspacioEstados(acciones, estado_inicial,
estados_finales)
```

Y mostraremos los resultados con el algoritmo de anchura como en el apartado anterior.

En el **apartado 3** desarrollamos un método que englobe todo el desarrollo anterior, donde además incluiremos el borrado de líneas para las líneas que estén llenas.

```
Borrado de lineas
Entradas:
        mapa_inicial
Salidas:

Algoritmo:
        resultado = 2
        mientras resultado == 2 o resultado == 1
                resultado = 2
                lineaBorrada =  -1

                para i en rango 22
                        cont = 0
                        si 1 en mapa_inicial[i]
                                para r en rango 10
                                        si mapa_inicial[i]
[r] == 1
                                                cont =
cont +1

                                si cont == 10
                                        resultado = 1
                                         mapa_inicial[i] =
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
                                        lineaBorrada = i
```

Nuestro algoritmo nos preguntara para cada iteración si queremos continuar con la siguiente pieza y solo aceptara por respuesta SI para continuar y NO para detener el juego. Para cada iteración mostraremos la pieza que aparece, su mejor posición, los pasos realizados para llegar a dicha posición y cómo quedaría el mapa resultante.

IV. RESULTADOS

A continuación, vamos a detallar los distintos experimentos realizados para mostrar el correcto funcionamiento de los algoritmos desarrollados. Utilizaremos capturas de pantalla para apoyar las explicaciones.

Este será el mapa inicial con el que realizaremos las pruebas del **apartado 1.**

```
#Matriz creada mediante el mapa

mapa =  Mapa([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
              [1, 1, 1, 1, 1, 0, 0, 0, 0, 0]])
```

Crearemos el problema con los valores mostrados a continuación utilizando los algoritmos de problema de espacio de estados.

```
# Creación del problema

acciones = [Abajo(), Izquierda(), Derecha(), Girar_izquierda(), Girar_derecha()]

estado_inicial =  [[0, 3], [1, 3], [1, 4], [1, 5]]
estado_final1 = [[17, 4], [18, 4], [19, 3], [19, 4]]
estados_finales = [estado_final1]
Juego_Mejor_Camino = probee.ProblemaEspacioEstados(
    acciones, estado_inicial, estados_finales)
```

Una vez creado el problema, utilizaremos varios algoritmos de búsqueda de estado, tanto informados como no informados.

```
# Búsqueda en Anchura:
anchura = búsqee.BúsquedaEnAnchura()
solucion = anchura.buscar(Juego_Mejor_Camino)

if solucion:
    print(transforma_solucion(solucion))
else:
    print("No encontró solución ")
```

```
A A A A A A A A A A A A A A A A A G_izq
```

```
# Búsqueda A*: hay que proporcionar la función de estimación del coste h.

a_estrella = búsqee.BúsquedaAEstrella(h)  # La función h debe estar definida previamente y estar relacionada con este problema
solucion = a_estrella.buscar(Juego_Mejor_Camino)

if solucion:
    # print(solucion)
    print(transforma_solucion(solucion))
else:
    print("No encontró solución :(")

# La solución obtenida se ve más natural ya que además del coste, la función h prioriza el estado final más cercano
# creando una linea "recta" entre ellos.
```

```
A A A A A A A A A A A A A A A A A G_izq
```

Como podemos observar, para un problema tan sencillo como el que se nos plantea

Por último, vamos a hacer una comparativa de los tiempos de ejecución de los distintos algoritmos utilizados.

```
#Tiempos de ejecución:

print("Anchura:",time.timeit('busquedaAnchuraTime()' ,setup="from __main__ import busquedaAnchuraTime",number = 1))
print("Primero el Mejor:",time.timeit('primeroElMejor()' ,setup="from __main__ import primeroElMejor",number = 1))
print("Optima:",time.timeit('Optima()' ,setup="from __main__ import Optima",number = 1))
print("A estrella:",time.timeit('AEstrella()' ,setup="from __main__ import AEstrella",number = 1))
```

```
Anchura: 0.22033703452916598
Primero el Mejor: 0.6455705353946541
Optima: 0.2303794877952896
```

```
# Búsqueda Óptima

optima = búsqee.BúsquedaÓptima()
solucion = optima.buscar(Juego_Mejor_Camino)

if solucion:
    # print(solucion)
    print(transforma_solucion(solucion))
else:
    print("No encontró solución :(")

# La solución que dá es una de las mejores, y al no tener en cuenta la función h, el orden en el que pongamos las
# acciones a la hora de definir el problema determinará la solución obtenida:
# solucion(Arriba,Abajo,Izquierda,Derecha) != solucion(Derecha,Abajo,Izquierda,Arriba)
```

```
A A A A A A A A A A A A A A A A A G_izq
```

```
# Primero el mejor prioriza la función h, ignora el coste
primero_el_mejor = búsqee.BúsquedaPrimeroElMejor(h)
solucion = primero_el_mejor.buscar(Juego_Mejor_Camino)

if solucion:
    # print(solucion)
    print(transforma_solucion(solucion))
else:
    print("No encontró solución :(")
```

```
A A A A A A A A A A A A A A A A A G_izq
```

Respecto al **apartado 2**, utilizaremos un mapa inicial distinto para realizar los experimentos.

```
In [112]: #Matriz creada mediante el mapa

mapa =  Mapa([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
              [0, 0, 1, 0, 1, 0, 1, 1, 0, 0],
              [0, 0, 1, 1, 1, 1, 1, 1, 1, 1],
              [1, 0, 0, 0, 0, 1, 0, 1, 0, 1],
              [1, 0, 0, 1, 0, 1, 0, 0, 0, 1],
              [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])
```

Nos serviremos de una serie de métodos auxiliares que nos ayuden a calcular las distintas posiciones finales posibles de la pieza que queremos colocar en el mapa para quedarnos con la mejor según la heurística que tenemos definida. Así pues, el método mejorPosicion(l) tendrá como entrada una lista de posibles posiciones finales de la pieza y finalmente nos devolverá la mejor que a continuación podremos introducir en el mapa.

```
def mejorPosicion(l):
    mejorValor = 999
    coordenadas = []
    for i in range(len(l)):
        valor = calculoHeuristica(mapa_inicial,l[i])
        if valor < mejorValor:
            mejorValor = valor
            coordenadas = l[i]

    print("El valor es:",mejorValor)
    return coordenadas

print("Posición:",mejorPosicion(Lista))
```

```
El valor es: 600
Posición: [[17, 0], [18, 0], [18, 1], [19, 1]]
```

```
introducir_en_mapa(mejorPosicion(Lista),mapa_inicial)
```

```
El valor es: 600
```

Llegados a este punto, solo queda crear el problema de espacio de estados.

```
# Creación del problema

acciones = [Abajo(), Izquierda(), Derecha(), Girar_izquierda(), Girar_derecha()]
estado_inicial = [[0, 4], [0, 5], [1, 3], [1, 4]]
estado_final1 = mejorPosicion(Lista)
estados_finales = [estado_final1]
Juego_Mejor_Camino = probee.ProblemaEspacioEstados(acciones, estado_inicial, estados_finales)
```

```
# Búsqueda en Anchura:
anchura = búsqee.BúsquedaEnAnchura()
solucion = anchura.buscar(Juego_Mejor_Camino)

if solucion:
    print(transforma_solucion(solucion))
else:
    print("No encontró solución ")
```

```
A A A A A A A A A A A A A A A I I I A G_izq A
```

Finalmente, solo queda realizar una búsqueda por anchura para probar que funciona correctamente.

El **apartado 3** se centra en utilizar lo desarrollado en el apartado anterior, introduciendo una secuencia de piezas generadas automáticamente.

Esta vez partiremos de un mapa inicialmente vacío.

```
#Matriz (22x10) creada mediante el mapa

mapa =  Mapa([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
```

Se generará una lista de piezas automáticamente y se irán introduciendo cada vez que el logaritmo nos pregunta si queremos continuar.

```
Bienvenido a nuestro juego del Tetris

Nuestro mapa inicial es el siguiente:
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0,
0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0,
0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0,
0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0,
0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0,
0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]


¿Deseas continuar el juego?(S/N):S

La pieza que aparece es: [[0, 3], [0, 4], [1, 4], [1, 5]]

La mejor posición para colocar la pieza es: [[20, 1], [20, 2], [21, 2], [21, 3]]

Los pasos que debe realizar son: A A A A A A A A A A A A A A A A A A A A I I



El mapa inicial quedaría de la siguiente manera:
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0,
0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0,
0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0,
0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0,
0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0,
0, 0, 0, 0, 0], [0, 1, 1, 0, 0, 0, 0, 0, 0, 0], [0, 0, 1, 1, 0, 0, 0, 0, 0, 0]]


¿Deseas continuar el juego?(S/N):S

La pieza que aparece es: [[0, 3], [0, 4], [1, 4], [1, 5]]

El mapa inicial quedaría de la siguiente manera:
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0,
0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0,
0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0,
0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0,
0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0,
0, 0, 0, 0, 0], [0, 1, 1, 1, 1, 0, 0, 0, 0, 0], [0, 0, 1, 1, 1, 1, 0, 0, 0, 0]]


¿Deseas continuar el juego?(S/N):S

La pieza que aparece es: [[0, 5], [1, 3], [1, 4], [1, 5]]

La mejor posición para colocar la pieza es: [[20, 8], [21, 6], [21, 7], [21, 8]]

Los pasos que debe realizar son: A A A A A A A A A A A A A A A A A A A A A D D D
```

En cada iteración se colocará una pieza automáticamente en la mejor posición posible. Las líneas completas se irán borrando una vez se vayan llenando y las piezas que quedan encima se irán recolocando.

En conclusión, el algoritmo desarrollado simula correctamente un juego de Tetris con la peculiaridad de que las piezas se posicionan de forma automática hasta que nosotros decidamos parar de jugar.

V. Conclusiones

El proyecto está formado por tres apartados, en el primer apartado nos pedía que dado un estado inicial llegaremos a un estado final, en caso de que fuese posible. Cosa que hemos podido hacer con éxito. En el segundo apartado, mediante métodos heurísticos hemos obtenido la mejor posición de la matriz para colocar una pieza dada. En el último apartado, hemos conseguido que la máquina "juegue" de manera automática, eligiendo la mejor posición posible, penalizando cosas como altura ponderada de las piezas en la matriz o los huecos que dejan dichas piezas al ser colocadas, y premiando que una pieza provoque un borrado de línea.

Nuestro proyecto está abierto a mejoras, como puede ser la introducción de Wall Kicks, para conseguir una mejor posición final de la pieza.
Lo que permitiría seguir con el juego más tiempo antes de perder la partida.

Referencias

[1] Lenguaje Python en encyclopedia libre Wikipedia https://es.wikipedia.org/wiki/Python.

[2] Tetris en enciclopedia libre Wikipedia https://es.wikipedia.org/wiki/Tetris.

[3] Definición de espacio de estados http://www.cs.us.es/~fsancho/?e=33.

[4] Tetrominó en enciclopedia libre Wikipedia https://es.wikipedia.org/wiki/Tetromin%C3%B3.

[5] Biografia de Alekséi Pazhitnov http://tetris.publijuegos.com/biografia_de_alexey_pazhitnov.html.

[6] Algoritmos de busqueda informada http://www.cs.us.es/~fsancho/?e=62.

[7] Algoritmos de busqueda no informada http://www.cs.us.es/~fsancho/?e=65.

[8] Preventing intrusive memories after trauma via a brief intervention involving Tetris computer game play in the emergency department: a proof-of-concept randomized controlled trial https://www.nature.com/articles/mp201723.

[9] Can playing the computer game “Tetris” reduce the build-up of flashbacks for trauma? A proposal from cognitive science https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0004153.

[10] Definición de heuristica https://es.wikipedia.org/wiki/Heur%C3%ADstica_(inform%C3%A1tica)