TetriSolve: resolvedor de Tetris mediante problemas de espacios de estados

Mario Bizcocho González
dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
mario.biz.gon@gmail.com; marbizgon@alum.us.es

Alejandro Monteseirín Puig
dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
alexmonteseirin@gmail.com; alemonpui@alum.us.es

***Resumen*— El objetivo de este trabajo es la implementación en Python de un juego de Tetris, así como un algoritmo que permita resolverlo automáticamente. Para ello, primero tomará un estado inicial con un tablero vacío y una lista de fichas; posteriormente, para cada ficha, buscará la posición óptima en la que colocarla siguiendo la lógica del juego y, por último, efectuará los desplazamientos oportunos para colocarla en esa posición.**

La jugada resultante del algoritmo debe estar entre las mejores posibles para el conjunto de piezas que se le pasa. Esto se garantiza por medio del enfoque de resolución basado en problemas de espacios de estados, lo que significa un acercamiento a la solución mediante la consideración de un estado final deseado, un estado inicial y un camino para conseguirlo que puede implementar una heurística, la cual satisfará el requisito de idoneidad del camino escogido.

Palabras Clave—Inteligencia Artificial, espacios de estados, Tetris, juego, algoritmo, búsqueda.

I. INTRODUCCIÓN

El ámbito de los videojuegos ofrece un amplio campo de aplicación para la inteligencia artificial. Títulos como *Grand Theft Auto* utilizan una gran variedad de algoritmos para ofrecer rutas a distintos destinos tanto al jugador como a peatones y vehículos pertenecientes a su compleja simulación urbana [1]. Otros, como *Hearts of Iron IV*, la usan para hacer un cálculo en tiempo real de las prioridades militares estratégicas de los países beligerantes en la II Guerra Mundial en su simulación de este periodo histórico, teniendo en cuenta factores como la importancia estratégica de cada región, sus capacidades militares o los recursos de que dispone[3]. Otros, como *Battlefield I*, manejan a los soldados controlados por el ordenador para que se pongan a cubierto si están heridos, ataquen si tienen línea de fuego con un enemigo o lancen una granada si ven que hay una gran concentración de enemigos en una zona[4]. Estos tres ejemplos son de videojuegos muy recientes, con algoritmos enormemente complejos desarrollados por estudios de desarrollo con un gran volumen de personal, tiempo, dinero y experiencia. Sin embargo, los conceptos de la inteligencia artificial se pueden aplicar a videojuegos que preceden a estos modernos ejemplos por varias décadas.

El Tetris es un videojuego desarrollado en la antigua Unión Soviética por el ingeniero informático Alekséi Leonídovich Pázhitnov. Se lanzó en junio de 1984, y su nombre deriva del griego *tetra* (cuatro, por los cuatro lados de los cuadrados que componen las piezas de Tetris) y el tenis, deporte por el que Pázhitnov tenía gran afición[2].

El juego consiste en colocar piezas compuestas de cuatro cuadrados (se las conoce como *tetrominós*) en un tablero de 23 filas (dos de ellas ocultas, en la que aparecen los nuevos tetrominós) y 10 columnas que además está afectado por la gravedad, de forma que las piezas van cayendo de forma automática. El jugador puede controlar, sin embargo, la rotación de las piezas, y también puede desplazarlas en el eje horizontal. Cuando una pieza se posa sobre otra que ya estuviera colocada, o sobre el fondo del tablero, esta se queda fija y ya no se ve afectada ni por la caída automática ni por el control del jugador. Cuando esto sucede, el juego coloca una nueva ficha en las filas superiores del tablero y se repite el proceso. En caso de que una de las filas quedara completamente llena, esta desaparece y las filas que hubiera arriba se desplazan hacia abajo, añadiéndose una vacía más en el nivel superior. Los tetrominós se pueden rotar mientras permanezcan en el aire y al rotarlos no choquen con las paredes u otras piezas.

Nuestro problema lanza la pregunta de cómo se podría, usando algoritmos de búsqueda y planificación, optimizar una partida dada una lista de tetrominós predefinida. Esto presenta una serie de retos interesantes en, por ejemplo, cuál es el mejor criterio que aplicar para decidir dónde colocar una determinada ficha. Debemos intentar completar filas, pero también se debe intentar que los tetrominós que coloquemos no incrementen demasiado la altura total de la estructura o nos arriesgamos a perder la partida. Otros elementos que pueden presentar conflictos son las estrategias de acercamiento para llevar el tetrominó a su posición final, ya que existen varios algoritmos para realizar esta tarea y cada uno tiene sus particularidades que lo hacen más o menos apto para esto, sobre todo en cuanto a la eficiencia en el tiempo y la posibilidad de refinar heurística.

Pare resolver el problema hemos tomado un acercamiento a través de búsqueda de estados. Primero tomamos una lista de fichas e iteramos sobre ella, pasando cada una a un algoritmo *ad hoc* para decidir dónde colocarla que usa elementos del sistema de búsqueda de estados, si bien no utiliza un algoritmo de búsqueda propiamente dicho (probamos a poner el tetrominó en todas las posiciones del tablero, y luego iteramos entre las opciones válidas para tomar la que tenga mejor heurística). Posteriormente, aplicamos una búsqueda con A* para encontrar

el conjunto de movimientos más eficiente para llevar a esa ficha a la posición designada por el anterior paso.

A continuación, detallaremos con la debida profundidad los pasos que hemos llevado a cabo para desarrollar esta solución. La sección II (PRELIMINARES) plantea los detalles de las técnicas que hemos empleado y los trabajos que hemos encontrado que plantean también un acercamiento a este problema – si bien no hemos utilizado ninguno de estos planteamientos en nuestro trabajo, en parte porque solo los encontramos una vez que ya teníamos planteado el acercamiento que queríamos tomar. Posteriormente, en la sección III (METODOLOGÍA), plantearemos el detalle de nuestra solución, empezando por la colocación de un tetrominó dado en una posición específica, siguiendo por el algoritmo de decisión para elegir la posición final deseada, y finalizando con la función que toma la lista de fichas y ejecuta la partida. En la sección IV (RESULTADOS) exponemos los experimentos que hemos realizado para verificar el correcto funcionamiento de nuestro algoritmo, así como los resultados obtenidos de ellos y un análisis de los resultados que extraemos. Por último, en la sección V (CONCLUSIONES), argumentamos sobre la sección anterior, el conocimiento extraído del desarrollo del trabajo y una serie de ideas de mejora para una posible reanudación de este trabajo.

II. PRELIMINARES

A continuación exponemos los métodos empleados para el desarrollo de este trabajo, sus particularidades y sus puntos fuertes para el campo de aplicación concreto del Tetris. Asimismo, visitaremos trabajos relacionados con el actual, que pueden ayudarnos a apreciar otros acercamientos a este problema y formas en las que otras personas han resuelto los retos que plantea.

A. *Métodos empleados*

El método utilizado para la resolución del problema es la búsqueda en espacios de estados. Esta técnica se basa en la consideración de configuraciones o *estados* sucesivos de una instancia, con la intención de encontrar un *estado objetivo* con la propiedad deseada. Modelar un problema como un espacio de estados, por tanto, consiste en representar los distintos estados en los que puede hallarse nuestro problema y formar un grafo en el que dos estados están conectados si existe una operación que pueda realizarse para transformar el primer estado en el segundo[5].

En la implementación que usamos, tomada de la práctica 2 del curso (reutilizando los archivos búsqueda_espacio_estados.py y problema_espacio_estados.py), los estados aparecen definidos como tableros de Tetris, es decir, matrices bidimensionales de 25 filas y 14 columnas. Las paredes, para evitar problemas a la hora de comprobar los bordes con las piezas más grandes, aparecen representadas con doble ancho.

- Los muros aparecen representados como elementos de la matriz de valor **7**.
- Las piezas ya asentadas se representan como elementos de valor **3**.
- El cuadrado central de la pieza activa se representa como elemento de valor **2**.
- El resto de los cuadrados de la pieza activa se representan con elementos de valor **1**.

Un ejemplo de una pieza con forma de T encajada entre piezas asentadas se puede apreciar en la **Figura 1**.

```
[[7. 7. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 7. 7].
 [7. 7. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 7. 7].
 [7. 7. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 7. 7].
 [7. 7. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 7. 7].
 [7. 7. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 7. 7].
 [7. 7. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 7. 7].
 [7. 7. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 7. 7].
 [7. 7. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 7. 7].
 [7. 7. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 7. 7].
 [7. 7. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 7. 7].
 [7. 7. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 7. 7].
 [7. 7. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 7. 7].
 [7. 7. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 7. 7].
 [7. 7. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 7. 7].
 [7. 7. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 7. 7].
 [7. 7. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 7. 7].
 [7. 7. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 7. 7].
 [7. 7. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 7. 7].
 [7. 7. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 7. 7].
 [7. 7. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 7. 7].
 [7. 7. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 7. 7].
 [7. 7. 3. 3. 3. 3. 1. 2. 3. 3. 3. 3. 7. 7].
 [7. 7. 3. 3. 3. 3. 3. 1. 3. 3. 3. 3. 7. 7].
 [7. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7].
 [7. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7]
]
```

Fig. 1. Ejemplo de un estado (matriz bidimensional). Imagen de producción propia.

La forma de relacionar un estado con otro se define mediante acciones. Estas tienen una serie de métodos definidos por defecto, concretamente es_aplicable (que debe ser sobreescrita a la hora de construir la acción), aplicar (también debe ser sobreescrita) y coste_de_aplicar (puede ser sobreescrita, si no lo es siempre devuelve 1).

- **es_aplicable**: esta función comprueba si, con el estado sobre el que se intenta aplicar la acción, esta puede realizarse o no. Sirve para dejar de calcular estados que, en el grafo de estados, dependan del actual.
- **aplicar**: define la acción que se debe tomar para aplicar la acción sobre el estado. Se realiza justo después que es_aplicable si esta última devuelve un resultado positivo.
- **coste_de_aplicar**: define un valor numérico fijo o variable para el coste de aplicación de una acción. Hay algoritmos de búsqueda que tienen en cuenta esta métrica para encontrar un camino óptimo.

Asimismo, a la hora de declarar un problema de espacio de estados se le debe pasar la lista de acciones posible completa, así como el estado inicial y los estados finales. El problema se considerará resuelto si el estado que resulta de aplicar el estado que se está evaluando está incluido en la lista de estados finales,

y deberá ser recorrido usando un algoritmo de búsqueda, como búsqueda en anchura, búsqueda en profundidad o búsqueda A estrella. En la única implementación completa de búsqueda que hemos realizado, la búsqueda de los movimientos necesarios para llegar a una determinada posición con el tetrominó activo, hemos usado A estrella. Lo explicaremos más en detalle en la sección III.

B. *Trabajo Relacionado*

El problema de desarrollar un jugador de Tetris ideal mediante inteligencia artificial ha sido afrontado en multitud de ocasiones por personas de todo el mundo. Un caso especialmente bien documentado es el del bot (casi) perfecto[6], según sus propias palabras, de Yiyuan Lee, estudiante de Ciencias de la Computación en la Universidad de Singapur. Tiene en cuenta un tablero con las mismas dimensiones que el nuestro, con las mismas reglas para hacer aparecer piezas y con todos los tetrominós. Además de esto permite al algoritmo conocer cuál será su próxima pieza, usa los *Wall Kicks* (comportamiento especial cuando se gira una pieza y golpea contra algo, que nosotros no hemos podido implementar por falta de tiempo), y rota los tetrominós de manera distinta a la nuestra. Aún así, en general, la implementación del juego base es muy parecida a la nuestra.

La gran diferencia, sin embargo, radica en la heurística y el tipo de algoritmo. Lee ha usado para este trabajo algoritmos genéticos, que quizá se ajusten más a la naturaleza del Tetris por el hecho de no tener que definir desde el principio un conjunto de estados finales que puede ser considerablemente grande. Además, su heurística es más compleja, teniendo en cuenta 4 factores:

- Altura agregada: se calcula como la suma de las alturas de cada columna. Se pretende minimizar.
- Líneas completas: la cuenta de líneas que se completan en la jugada. Se pretende maximizar.
- Agujeros: cuenta de espacios vacíos con al menos un bloque sobre ellos. Se pretende minimizar.
- Irregularidad: suma de las restas absolutas de las alturas de las columnas. Se pretende minimizar, ya que es indeseable que las columnas tengan alturas demasiado dispares porque, si una muy vacía acaba cubierta, crea un gran número de agujeros. Esta métrica se explica en la Figura 2.

El señor Lee usó algoritmos genéticos para definir unos parámetros *a*, *b*, *c*, *d*, que configuran un vector en $\mathbb{R}^4$. El algoritmo simulaba 100 partidas para cada gen, cada una con como máximo 500 tetrominós, y se le asignaba un fitness igual al número de filas que consiguiese vaciar. Se usaba selección por torneo para decidir los genes que conseguían reproducirse y, además, se usaban medidas para que los genes hijos se parecieran más a su gen progenitor con mayor fitness. Cuando la población alcanzaba cierto tamaño se eliminaba el 30% más débil de la población y se repetía el cruce para rellenar este vacío con nuevos cruces, y cada individuo tenía una pequeña posibilidad de mutar.

El vector de 4 elementos resultante de este proceso es posteriormente usado para decidir, en cada caso, cuál es la mejor alternativa en cada movimiento, multiplicándolo para ello con el vector resultante de las cuatro métricas arriba mencionadas, usando para ello información sobre la situación del tablero, la ficha actual y la siguiente ficha que va a aparecer y calculando un número suficiente de pasos futuros. La alternativa óptima es elegida y ejecutada, y se recalcula la mejor alternativa cada vez que se toma una nueva pieza.

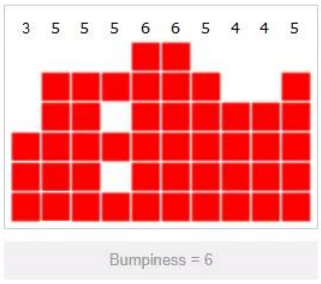

Fig. 2. Explicación de la métrica de irregularidad. Captura de pantalla de https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/, propiedad de Yiyuan Lee.

El desarrollador de este algoritmo asegura que, para su conjunto de reglas, el algoritmo estuvo ejecutándose durante dos semanas y llegó a vaciar más de 2 millones de filas antes de que lo tuviera que apagar debido a sobrecalentamiento de la máquina en la que lo estaba ejecutando. Probablemente habría podido continuar indefinidamente.

También existen otros ejemplos, como el Tetris AI del usuario de GitHub *fthomasmorel*[7], desarrollado en Python pero mucho peor documentado, o el del usuario de GitHub *allenfrostline*[8], también en Python y basado en pygame. Hemos preferido ilustrar más en detalle el ejemplo del señor Lee, sin embargo, debido a su excelente documentación y a la interesante heurística que ha utilizado.

III. Metodología

Para explicar la metodología que hemos empleado, iremos en el mismo orden en el que realizamos el trabajo. Comenzaremos por el algoritmo de búsqueda de movimientos para colocar el tetrominó en su sitio, seguiremos con el que encuentra la posición óptima para colocarlo y terminaremos con el que ejecuta la partida.

1) *Búsqueda de movimientos*

El algoritmo de búsqueda de movimientos es un ejemplo puro de búsqueda de estados. Comenzamos con un estado inicial consistente en un tablero de Tetris, que es el resultado de colocar la pieza anterior (o, en el caso del comienzo de la partida, un tablero vacío). Al construir el problema de espacio de estados también le pasamos un estado final, consistente en el tablero resultante de los movimientos que queremos obtener. Este tablero se habrá conseguido mediante la ejecución del algoritmo de búsqueda de posición, que explicaremos tras este.

Al construir el problema de estados también debemos definir la colección de acciones que contemplamos en su ejecución. En concreto, estas acciones son Mover Abajo, Mover Izquierda, Mover Derecha, Rotar y Rotar Inverso.

- Mover Abajo:
    - **Es_aplicable**: siempre devuelve verdadero.
    - **Coste_de_aplicar**: devuelve 0, ya que entendemos que esta es una acción que no tiene coste porque el descenso del tetrominó activo sucede de forma automática.
    - **Aplicar**: Este aplicar tiene dos casos, en función del resultado de la siguiente comprobación: Si las caras inferiores de la pieza están en contacto con al menos una pieza asentada o el borde inferior del tablero se realizará el caso 1 y en caso negativo se realizará el caso 2
        1. Caso 1: Debido a que la ficha no puede desplazarse hacia abajo debido a que toca con otra ficha o con el borde inferior del tablero convertimos todos los 1 y el 2 de la pieza en 3, para hacerla fija. Tras esto se hace una comprobación adicional, se comprueba si hay que eliminar alguna fila del tablero debido a que se haya completado una fila, en ese caso la fila se elimina y se añade una nueva vacía arriba del tablero, esto provoca que todas las filas que estaban encima de esta "caigan" hacia abajo.
        2. Si debajo de la ficha no hay nada sencillamente hacemos descender todos sus cuadrados una fila.

Aplicar Mover Abajo
Entrada:
- Un estado E (tablero de Tetris)

Salidas:
- Un estado NE resultado de la aplicación del movimiento

Algoritmo:
1. NE = copia profunda(E)
2. Si chocaría con suelo o pieza asentada(E)
    a. Para cada columna en una lista invertida(E) = C
        i. Para cada elemento en columna(C)=ELEM
            1. Si ELEM==1: NE[ELEM] = 3
            2. Si ELEM==2: NE[ELEM] = 3
    b. NE = eliminar filas llenas(E)
3. Sino
    a. Para cada columna en una lista invertida(E) = C
        i. Para cada elemento en columna(C)=ELEM
            1. Si ELEM==1:
                a. NE[ELEM]=0
                b. NE[ELEM fila inferior]=1
            2. Si ELEM==2:
                a. NE[ELEM]=0
                b. NE[ELEM fila inferior]=2
4. Devolver NE

- Mover Derecha:
    - **Es_aplicable**: verdadero si ninguno de los cuadrados tiene muro o pieza asentada a su derecha, falso en otro caso.
    - **Coste_de_aplicar**: devuelve 1, ya que es una acción que debe ser realizada manualmente si el jugador es humano.
    - **Aplicar**: recorre la matriz de derecha a izquierda y va desplazando los 2 y los 1 una casilla hacia la derecha.
- Mover Izquierda:
    - **Es_aplicable**: verdadero si ninguno de los cuadrados tiene muro o pieza asentada a su izquierda, falso en otro caso.
    - **Coste_de_aplicar**: devuelve 1, ya que es una acción que debe ser realizada manualmente si el jugador es humano.
    - **Aplicar**: recorre la matriz de izquierda a derecha y va desplazando los 2 y los 1 una casilla hacia la izquierda.
- Rotar:
    - **Es_aplicable**: verdadero si la pieza, al ser rotada, no ocupa ninguna casilla con 3 o con 7. Falso en caso contrario.
    - **Coste_de_aplicar**: devuelve 1, ya que es una acción que debe ser realizada manualmente si el jugador es humano.
    - **Aplicar**: Para cada elemento de la matriz original en un radio de 2 elementos desde el centro, copia los 1 y los 2 a la posición rotada en la nueva matriz.
- Rotar Inverso:
    - **Es_aplicable**: verdadero si la pieza, al ser rotada, no ocupa ninguna casilla con 3 o con 7. Falso en caso contrario.
    - **Coste_de_aplicar**: devuelve 1, ya que es una acción que debe ser realizada manualmente si el jugador es humano.
    - **Aplicar**: Para cada elemento de la matriz original en un radio de 2 elementos desde el centro, copia los 1 y los 2 a la posición rotada en la nueva matriz.

Estas acciones son recogidas en una colección e introducidas en un problema de espacio de estados, junto con el tablero del estado inicial y el tablero del estado final. En una función recursiva que hemos llamado CalculoMovimientos construimos estos problemas de estados. Para recorrerlos, usamos el algoritmo de búsqueda A estrella, conocido por su eficacia a la hora de encontrar caminos óptimos en la menor cantidad posible de tiempo. Para ello hay que proporcionarle una función heurística que, en nuestro caso, ayuda a acercar el centro del tetrominó a la posición de este elemento en el estado final.

Esta función heurística toma el nodo actual y primero encuentra la posición del centro de la pieza en el estado del nodo. Posteriormente, toma la posición del centro de la pieza en el estado final. Si en alguno de los dos no ha encontrado centro de pieza (es decir, en una cadena de movimientos en la que la pieza se haya asentado sin llegar al estado final), devuelve 1000 como heurística, lo cual, teniendo en cuenta que es una heurística para minimizar, es un resultado pésimo.

Si no es el caso, la heurística devuelve la suma de las diferencias absolutas entre la posición original del centro de la pieza en el estado perteneciente al nodo y en el estado final, siguiendo la fórmula 1:

$$heurística = |x - xFinal| + |y - yFinal| \qquad (1)$$

La búsqueda A estrella, una vez que se activa, recorrerá los nodos priorizando los nodos frontera que tengan la menor suma de coste de aplicar y heurística, ya que, como se ha establecido, la heurística es un valor que minimizar.

2) Búsqueda de estado final

Para la implementación de la búsqueda de estado final hemos tenido que implementar un algoritmo de búsqueda *ad hoc* que nos permite calcular una lista ordenada de estados de acuerdo con su idoneidad según nuestra heurística. Para ello, definimos una nueva "acción", PruebaPoner, que intenta colocar una pieza determinada en una posición de un tablero dado.

- PruebaPoner:
    - **Es_aplicable**: verdadero si la pieza, al ser colocada en esa posición, no choca con muros o con piezas asentadas. Falso en caso contrario.
    - **Coste_de_aplicar**: Se tienen en cuenta tres parámetros, cada uno de ellos valorado de forma distinta. Por cada fila que se consiga eliminar colocando la ficha en esa posición se restan 10000 puntos. Por cada hueco que tape, se suman 1000 puntos (ya que se crea una situación en la que estas casillas se vuelven inaccesibles, como se puede ver en el trabajo del señor Lee[6]). También se restan puntos mientras más abajo se coloque la pieza a razón de 1 punto por fila. Estas operaciones se realizan sobre una puntuación base de 10000 puntos. Es posible obtener heurísticas negativas, aunque no influye en el resultado del algoritmo.
    - **Aplicar**: Para cada elemento de la pieza a colocar, se recorre el estado y se copian los componentes de la pieza al tablero objetivo.

Heurística A*
Entrada:
- Un nodo N (tablero de Tetris con función heurística y conectado a otros elementos de un grafo)

Salidas:
- Un valor numérico H

Algoritmo:
1. hayEstado = 0
2. Para cada columna en una lista (N.estado) = C
    a. Para cada elemento en columna(C)=ELEM
        i. Si ELEM==2:
            1. hayEstado=1
            2. xFicha=x
            3. yFicha=y
3. Para cada columna en una lista (N.estado) = C
    a. Para cada elemento en columna(C)=ELEM
        i. Si ELEM==2:
            1. Sumar 1 a hayEstado
            2. xFinal=x
            3. yFinal=y
4. H=1000
5. Si hayEstado==2
    a. H=absoluto(xFicha-xFinal)+absoluto(yFicha-yFinal)
6. Devolver H

Inicialmente se inició el desarrollo del calculo de estado final creando una función que dado un estado de tablero y una pieza te devolviera de entre todas las posibilidades la de menor coste, que tal como esta planteada es por así decir la Heurística de esta acción , esto tenia un inconveniente ya que el mejor estado no tiene por qué ser accesible(en la sección IV.RESULTADOS,se explica más detalladamente esta posible situación), así que fue cambiado a una lista ordenada de los mejores estados. En el caso de que el estado más optimo sea imposible de alcanzar, pasara al siguiente.

El algoritmo comienza declarando todas las posibles rotaciones de la pieza que estamos intentando colocar en el tablero. Tras esto, las recoge en una colección y declara una acción para cada posible posición y rotación de la pieza en el tablero. Declaramos una colección vacía de estados finales y entonces, para cada acción (si es aplicable), la realizamos sobre el estado inicial. Cuando hemos recorrido todas las posibles acciones, devolvemos la lista ordenada con la función Python *sorted*, con la que podemos ordenar la lista resultante mediante, en este caso, la función PruebaPoner.coste_de_aplicar, para cada uno de los estados. Esto resulta en una lista de estados finales ordenados de mejor a peor.

Por último, esta lista ordenada se suministra a la función CalculoMovimientos, de naturaleza recursiva. Esta función, de

la que ya hablamos cuando expusimos la forma en la que buscamos los movimientos, utiliza a esta para determinar de forma voraz si es posible o no colocar la pieza en la posición que hemos seleccionado más allá de lo que dice la función es_aplicable de PruebaPoner a causa de, por ejemplo, un bloqueo en las filas superiores que impida al tetrominó llegar a la posición objetivo. Si es posible colocarla, tomará la sucesión de movimientos y la devolverá; si no es posible, imprimirá en la consola que no ha podido colocar la ficha y volverá a intentarlo con el siguiente estado procedente del proceso anterior.

En más profundidad la función calculaMovientos recibe un estado al que debe llegar ,una pieza y un número entero "intentos"(la cual usaremos para la recursividad principalmente) ,la función tiene un try catch entonces intentara colocar la pieza desde la posición inicial mediante la búsqueda A* explicada anterior mente utilizando las acciones que tiene, si lo consigue devolverá una lista de acciones que tras realizarlas llegaras al sitio final además de un número que representa en que intento se consiguió, en caso de que no lo consiga devuelve una excepción que es capturada, una vez capturada se decide si intentarlo de nuevo, en el caso de que haya mas posiciones finales posibles se pasa a la siguiente, y a la siguiente de manera recursiva hasta dar con un resultado, si ya se han probado todas las posibilidades y no se puede llegar a ninguna se devuelve un resultado de error y se imprime por pantalla que es imposible.

Cuando se dé el caso de que no se pueda llegar a ninguno de los estados que aparecen recogidos en la lista de estados significará que se ha perdido, ya que todos los estados están o bien bloqueados o bien bajo un bloque que impida acceder a la posición objetivo. Si esto pasa, consideramos que el algoritmo ha terminado y se finaliza la ejecución. En la sección IV. RESULTADOS exponemos los resultados de la prueba extensiva del algoritmo y en qué punto de la ejecución se llega a la situación de derrota.

3) Ejecución de la partida

En este apartado en concreto no se puede comentar una implementación en concreto, sino un patrón general, ya que la función de ejecución de la partida se adapta, en esencia, a la naturaleza del experimento que se quiera realizar con ella. La estructura básica sería la siguiente:

1. Definimos una lista de fichas. Este paso debe ser realizado antes de comenzar la partida debido a nuestra implementación, y debe ser hecho con una función que asegure aleatoriedad para crear una partida justa.
2. Definimos el estado inicial. Para una partida aleatoria debemos pasarle el tablero vacío, aunque podemos pasarle un tablero preparado para probar una característica determinada del algoritmo.
3. Ponemos una ficha en el tablero. Esta debe ser la siguiente en la lista de fichas
4. Obtenemos la lista de estados finales ordenados con la función correspondiente.
5. Usamos la función CalculoMovimientos con la anterior lista de estados finales ordenados de más optimo a menos optimo cómo poner la nueva ficha en el tablero. Nos devuelve un par formado por la lista de movimientos y el intento en el que se consiguió, que equivale al índice del estado final al que logro llegar, si fue en el primer intento pues será el primero de la lista, si a la segunda pues él segundo....
6. Si no se consiguió, CalculoMovimientos nos devolverá un error que se traducirá a "fin de la partida"
7. En caso de que CalculoMovimientos nos devuelva un resultado extraemos el intento en el que lo consiguió (casi siempre será a la primera)
8. Con el intento y el calculador de estados podemos obtener el estado al que logro llegar CalculoMovimientos, que siempre será el mas óptimo de entre los posibles, ese estado es ahora el estado actual.
9. El estado actual tiene la ficha colocada en el lugar optimo según el Calculador de movimientos, ahora realizamos la acción Mover_Abajo para que las fichas se hagan fijas y se eliminen filas en el caso de que se haya completado una fila. Ese estado es ahora el estado actual.
10. Repetimos el ciclo con la siguiente ficha en la lista y el estado del paso 9 indefinidamente, hasta que nos quedemos sin fichas o la partida acabe porque se haya quedado sin soluciones posibles, es decir TetriSolve pierda la partida

IV. RESULTADOS

A continuación, describiremos los experimentos realizados sobre el algoritmo, así como los resultados que ofrecen y las conclusiones que de ellos se derivan:

1) Experimento del camino bloqueado

Este experimento fue creado a raíz de una tutoría con el profesor Luis Valencia. En esta tutoría, el profesor nos sugirió que tuviésemos en cuenta una situación como la que se describe en la **Figura 3**, en la que al tetrominó activo le es imposible llegar a la posición objetivo debido a un bloqueo en el camino.

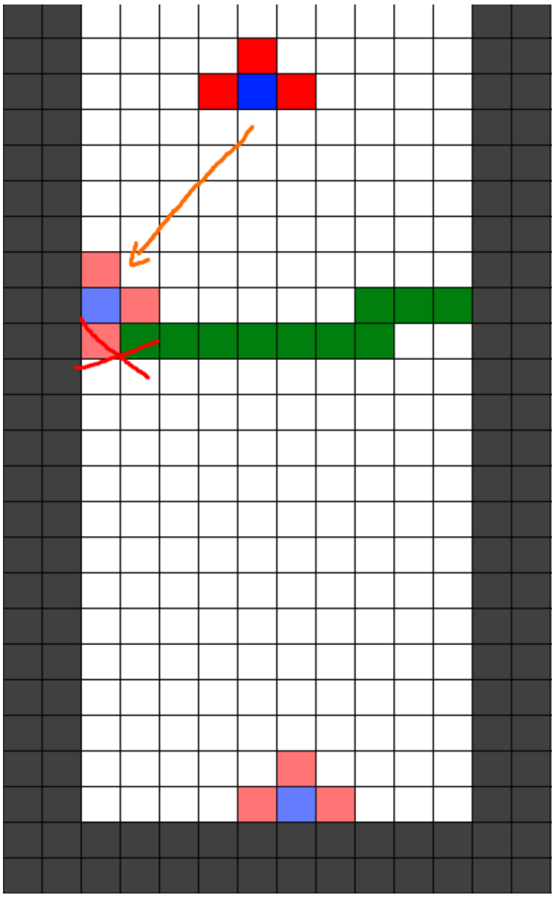

Fig. 3. Explicación de la situación en la que una figura no podría acceder a su posición recomendada (abajo), y la posición alternativa que debería surgir.

- Planteamiento: definir un estado en el que una ficha no pueda llegar a la posición definida como óptima por el algoritmo de búsqueda de estado final y asegurarse de que pueda colocar la ficha en alguna posición si esta falla.
- Resultados: el algoritmo intenta entre 10 y 29 de las 64 posibilidades en el tablero creado con esta trampa (según heurística) y finalmente consigue colocar la ficha sobre el bloqueo.
- Conclusión: la forma que hemos escogido para salvar este problema asegura que siempre se va a poder colocar la ficha. Sin embargo, radica en un incremento masivo del tiempo, ya que tiene que hacer un paso de cálculo completo por cada intento fallido. Una posible forma de mejorar este acercamiento sería la optimización de los pasos subyacentes, para acercarse lo más posible a una ejecución en tiempo real. Se garantiza que, cuando el algoritmo no puede devolver un movimiento válido, significa que se ha perdido la partida.

2) Experimento de rendimiento extensivo

- Planteamiento: definir un estado inicial vacío, proporcionar una lista de fichas de un tamaño con un orden de magnitud de al menos 5 y ejecutar la partida para observar cuántas jugadas es capaz de sobrevivir el algoritmo.
- Resultados: si se toma una lista de fichas consistente de iteraciones consecutivas sobre la lista original de fichas (I, J, L, O, S, T, Z), hemos llegado a conseguir una ejecución consistente en 2378 iteraciones (tetrominós colocados), antes de que el kernel de Jupyter se quedase colgado por insuficiencias del hardware. Sin embargo, en las 4 experiencias con una lista de piezas aleatorizada por completo que hemos llevado a cabo hemos logrado resultados dispares, que oscilan entre 170 iteraciones y 1600 iteraciones.
- Conclusiones: nuestro algoritmo, por la forma en la que está planteado, parece no saber bien qué hacer con las piezas O. Quizá se deba a que tienen el centro en una esquina a pesar de ser regulares, pero cuando se suceden varias piezas de este tipo se crean situaciones que luego el algoritmo no es capaz de remontar.

3) Visualización de los resultados:

Para ver los resultados hemos realizado una visualización utilizando HTML como se puede ver en la Figura4, concretamente una tabla de filas y columnas, para que fuera más cómodo que mostrar la matriz directamente, además se adjuntan distintos experimentos ordenados de mayor nivel de explicación a menor nivel.

7	7	0	0	0	0	0	0	0	0	0	0	7	7
7	7	0	0	0	0	0	0	0	0	0	0	7	7
7	7	0	0	0	0	0	0	1	2	1	0	7	7
7	7	0	0	0	0	0	0	0	1	0	0	7	7
7	7	0	0	0	0	0	0	0	0	0	0	7	7
7	7	0	0	0	0	0	0	0	0	0	0	7	7
7	7	0	0	0	0	0	0	0	0	0	0	7	7
7	7	0	0	0	0	0	0	0	0	0	0	7	7
7	7	0	0	0	0	0	0	0	0	0	0	7	7
7	7	0	0	0	0	0	0	0	0	0	0	7	7
7	7	0	0	0	0	0	0	0	0	0	0	7	7
7	7	0	0	0	0	0	0	0	0	0	0	7	7
7	7	0	0	0	0	0	0	0	0	0	0	7	7
7	7	0	0	0	0	0	0	0	0	0	0	7	7
7	7	0	0	0	0	0	0	0	0	0	0	7	7
7	7	0	0	0	0	0	0	0	0	0	0	7	7
7	7	0	0	0	0	0	0	0	0	0	0	7	7
7	7	0	0	0	0	0	0	0	0	0	0	7	7
7	7	0	0	0	0	0	0	0	0	0	0	7	7
7	7	0	0	0	0	0	0	0	0	0	0	7	7
7	7	0	0	0	0	0	0	0	0	0	0	7	7
7	7	3	3	3	3	0	0	3	3	3	3	7	7
7	7	3	3	3	3	3	0	3	3	3	3	7	7
7	7	7	7	7	7	7	7	7	7	7	7	7	7
7	7	7	7	7	7	7	7	7	7	7	7	7	7

Fig. 4. Muestra del tablero con la visualización Html creada, se puede apreciar una ficha arriba a la derecha en movimiento y las dos ultimas lineas ocupadas por fichas fijas excepto un hueco.

A continuación, se exponen en detalle los distintos experimentos realizados:

- Dado un estado inicial vacío, calcula los movimientos, mueve la ficha al objetivo e inserta la siguiente, El resultado son las distintas tablas HTML mostrando lo que va pasando en el Tetris.
- TetrisVisual: dado un estado inicial y una lista de fichas muy limitada va mostrando los tableros y los movimientos a realizar.Mostrando por pantalla paso a paso
- TetrisLargoConFin: dado un estado y una lista de fichas más larga (301) muestra los estados finales y las fichas que va introduciendo, se puede seguir con facilidad lo que va haciendo
- TetrisLargoOptimizado: Dado un estado inicial y una lista de fichas eterna (más de 10000) simula jugar la partida, esta versión solo muestra por pantalla los datos mas relevantes, evitando mostrar el tablero dada la gran carga que produce tener miles de tablas. Con el estado 0 y la lista de fichas se ha llegado a 1510 fichas. Como se puede ver en la Figura 5

```
iteracion numero 1509 Conseguida
no encontrada solucion, probando de nuevo con otro estado intento numero:1de19
iteracion numero 1510 Conseguida
no encontrada solucion, probando de nuevo con otro estado intento numero:1de9
no encontrada solucion, probando de nuevo con otro estado intento numero:2de9
no encontrada solucion, probando de nuevo con otro estado intento numero:3de9
no encontrada solucion, probando de nuevo con otro estado intento numero:4de9
no encontrada solucion, probando de nuevo con otro estado intento numero:5de9
no encontrada solucion, probando de nuevo con otro estado intento numero:6de9
no encontrada solucion, probando de nuevo con otro estado intento numero:7de9
no encontrada solucion, probando de nuevo con otro estado intento numero:8de9
No Encontrada Solucion
Out[119]: 'fin de la partida'
```

Fig. 5. Resultados por consola finales del tetrisLargoOptimizado, llegando con éxito a 1510 Iteraciones tras las cuales le es inposible al calculador de movimiento llegar a los estados finales posibles,por lo que la partida acaba

- TetrisLargoAleatorio: Similar al anterior, pero con una lista aleatoria de fichas, se obtienen resultados más dispares. Hacemos especial hincapié en mostrar la figura aleatoria por consola.

```
ficha aleatoria:
0 0 0 0 0
0 0 0 1 0
0 1 2 1 0
0 0 0 0 0
0 0 0 0 0
iteracion numero 6 Conseguida
ficha aleatoria:
0 0 0 0 0
0 0 0 1 0
0 1 2 1 0
0 0 0 0 0
0 0 0 0 0
```

Fig. 6. En esta version hacemos especial Incapie en mostrar las fichas aleatorias,como curiosidad, suele perder la partida con una ficha larga, dadas las dificultades para rotarla una vez el tablero esta casi completo

Para ir ajustando la Heurística utilizamos el TetrisLargoConFin, dado que al principio apenas llegaba a 100 jugadas, poco a poco se fue aumentando, el cambio más drástico se vio al añadir a la heurística la decisión de no rellenar huecos, lo cual paso de cientos de jugadas a miles. Conforme las jugadas superaban una cifra más elevada nos vimos obligados a eliminar las impresiones de pantalla del estado del tablero dada la sobrecarga del sistema por tener tantas tablas, de ahí surgió Tetris Largo Optimizado y Tetris Largo Aleatorio, enfocados solo en conseguir un mayor número de fichas colocadas sin perder.

En cuanto al rendimiento del proyecto, nunca ha sido una prioridad, priorizando mejores resultados sobre tiempo de ejecución, esto provoca grandes tiempos de espera en las partidas en las que la lista de fichas es muy grande o infinita, sin embargo, en partidas con un numero de fichas razonable el resultado es más que aceptable pudiendo tardar pocos minutos en realizarlas

V. Conclusiones

Durante este trabajo hemos desarrollado una inteligencia artificial basada en problemas de búsquedas de espacios de estados que, a partir de un estado inicial y una lista de tetrominós, es capaz de producir una partida en la que se garantiza un buen resultado. Se divide en tres componentes: un algoritmo puro de búsqueda de espacios de estados que, dado un estado inicial y uno final, busca la sucesión óptima de movimientos para alcanzar esta posición con el tetrominó activo; otro algoritmo, en este caso desarrollado a medida para este propósito, que se encarga de encontrar la posición óptima para un tetrominó en el tablero que se le indique, y, por último, una función para iterar sobre las distintas fichas y llamar a este último algoritmo *ad hoc* a medida que se va desarrollando la partida, y que puede diferir según la naturaleza del experimento o partida que queramos desarrollar.

Como podemos comprobar, la inteligencia artificial que hemos desarrollado es capaz de enfrentarse a un problema real

de Tetris y resolver situaciones de juego. Los experimentos que hemos llevado a cabo ilustran algunas situaciones en las que cabe margen de mejora, pero, aun así, tenemos la impresión de que hemos desarrollado un buen trabajo dentro de los parámetros que nos define la asignatura. Los resultados de ejecución son similares a los que hemos comentado con otros dos grupos que también les ha tocado este mismo trabajo, habiendo tomado ellos heurísticas bastante distintas a las nuestras hasta donde nosotros sabemos.

Durante el desarrollo del trabajo, hemos tenido oportunidad de darnos cuenta de que el acercamiento por búsqueda de espacios de estados para el problema del Tetris no es el más adecuado por una serie de razones. En primer lugar, este acercamiento obliga a conocer con anterioridad todas las posibles opciones que puede presentar el problema antes de formular un acercamiento, a diferencia de los algoritmos genéticos, que son la tendencia más generalizada entre los proyectos que hemos tenido oportunidad de observar en Internet investigando las obras relacionadas. Estos algoritmos permiten un acercamiento más modular y dinámico, mientras que mediante algoritmos de búsqueda ni siquiera ha sido posible resolver el problema de búsqueda de estados finales. Ha sido necesario implementar un algoritmo a medida, ya que, al estar intentando encontrar precisamente el estado final idóneo para la jugada correspondiente, no se le podían pasar al problema los estados finales posibles.

En efecto, tenemos una serie de ideas de mejora con respecto a este trabajo. Para comenzar, sería conveniente optimizar la ejecución de los algoritmos, ya que caemos en una serie de bucles fácilmente evitables y que podríamos intentar solucionar, no habiéndolo hecho ya meramente por falta de tiempo. También sería conveniente mejorar la heurística para evitar lo indicado en las conclusiones del experimento de rendimiento intensivo. Ciertamente, podemos intentar mejorar también la tecnología que nos permite mostrar la ejecución del Tetris. A medio-largo plazo, nos hemos planteado migrar el proyecto de Jupyter a pygame, con el objetivo de convertir el Tetris en un proyecto ejecutable y jugable como el del señor Lee[6]. De esta forma, se permitiría a un jugador humano proponerle una situación de juego al algoritmo, y sería interesante ver cómo este la resuelve. Esto, además, plantea la posibilidad de exponer la ejecución del programa en tiempo real en cualquier sitio de forma gráfica, y puede ser vistoso en, por ejemplo, exposiciones de arte informático.

REFERENCIAS

[1] Jan Henno Lauinger, Dr. J. Fürnkranz, Q. Hien Dang: Development of Traffic Simulation in a Game Environment, https://www.ke.tu-darmstadt.de/lehre/arbeiten/master/2016/Lauinger_JanHenno.pdf

[2] Varios autores: página de Wikipedia para Tetris. https://es.wikipedia.org/wiki/Tetris. Consultada el 20/06/2018.

[3] Paradox Interactive: HOI4 Dev Diary – Performance and AI, https://forum.paradoxplaza.com/forum/index.php?threads/hoi4-dev-diary-performance-and-ai.1070758/. Consultada el 21/06/2018.

[4] Varios autores: Wikipedia - Artificial Intelligence in Video Games, https://en.wikipedia.org/wiki/Artificial_intelligence_in_video_games#Video_game_combat_AI

[5] Varios autores: Wikipedia – State Space Search, https://en.wikipedia.org/wiki/State_space_search. Consultada el 28/06/2018

[6] Lee, Yiyuan: Tetris AI – the (Near) Perfect Bot, https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/. Consultada el 28/06/2018

[7] Fthomasmorel (usuario de GitHub): Tetris-AI, https://github.com/fthomasmorel/Tetris-AI. Consultada el 28/06/2018

[8] Allenfrostline (usario de GitHub): Tetris AI, https://github.com/allenfrostline/Tetris-AI, consultado el 28/06/2018