UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escuela Técnica Superior de Ingeniería Informática
Universidad Politécnica de Valencia

Tetris Solver
Jugando al Tetris con técnicas metaheurísticas

TRABAJO TÉCNICAS METAHEURÍSTICAS

Máster Universitario en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

Autor: Miquel Gómez

Resumen

```python
def generate_prompt(df_train: pd.DataFrame, df_batch: pd.DataFrame, genres: str)
    ↪ -> str:
    prompt = f"""
    Classify the following movie in any of these genres. More than one genre can
    ↪ be assigned.
    Genres: {genres}

    ============================================
    """

    for _, row in df_train.iterrows():
        prompt += f"Movie title: {row['movie_name']}\n"
        prompt += f"Plot: {row['description']}\n"
        prompt += f"Actual genres: {row['genre']}\n"
        prompt += "-------------------\n"

    prompt += """
    ============================================
    Now, classify the following movies returning a structured JSON
    response with the movie names and their genres.
    """

    for _, row in df_batch.iterrows():
        prompt += f"Movie title: {row['movie_name']}\n"
        prompt += f"Plot: {row['description']}\n\n"

    return prompt
```

Palabras clave: Clasificación de textos; Películas; Transformers; LLMs; Machine Learning

Índice general

Índice general IV
Introducción V
0.1 Problema a resolver . . . V
1 Codificación **1**
1.1 Movimientos posibles y tipos de genotipo . . . 1
1.2 Conversión genotipo a fenotipo . . . 3
1.3 Función objetivo . . . 3
1.3.1 Heurísticas . . . 4
1.3.2 Pesos de la función objetivo . . . 5
1.3.3 Notas . . . 6
2 Implementación **7**
2.1 Tecnología de Implementación . . . 7
2.2 Implementación del Tetris . . . 7
3 Experimentación **9**
3.1 Algoritmo genético . . . 9
3.2 Enfriamiento simulado . . . 9
4 Resultado **11**
4.1 Algoritmo genético . . . 11
4.2 Enfriamiento simulado . . . 11
4.3 Evolución . . . 11
5 Conclusiones **13**

Introducción

0.1 Problema a resolver

CAPÍTULO 1
Codificación

En este apartado se habla de como se han codificado los individuos para abordar el problema. Recordar que el objetivo es encontrar, para un set de piezas concreto, la posición y orientación óptima de cada pieza para minimizar el espacio ocupado. A eso hay que añadirle dos reglas del juego: primero que al completar una línea esta se limpia, y segundo que las piezas no pueden estar flotando al colocarse.

Con esto, se parte de la idea de 'jugar' al Tetris. La propuesta tras esta premisa no es jugar al juego de verdad, sino conseguir una codificación que permita cumplir las restricciones impuestas, consiguiendo soluciones que minimicen el espacio ocupado sin necesidad de descartar individuos inválidos.

La forma de atacar este problema de cobertura, será definiendo una serie de movimiento posibles. Se codifica la posición final de cada pieza como una secuencia de estos. Así pues, cada genotipo será una secuencia de movimientos, de forma que si hay x movimientos válidos y se dispone de n piezas, la codificación de un genotipo será una secuencia $x \times n$ movimientos.

Al codificar los genotipos de esta forma, los individuos resultantes serán siempre válidos, ya que cada pieza se colocará en el tablero siguiendo las reglas del juego. Esto implica también que deberemos ser capaces de simular el juego para poder evaluar cada individuo, ya que el fenotipo de cada individuo será el estado final del tablero tras colocar todas las piezas siguiendo los movimientos indicados en el genotipo.

También, habrá movimientos que resulten en '*no-op*', como intentar mover una pieza a la izquierda cuando ya está en el borde izquierdo del tablero. Más abajo vemos como se gestionan estos casos con la función objetivo.

1.1 Movimientos posibles y tipos de genotipo

Cuando decimos codificar los individuos como una secuencia de movimientos, es necesario definir cuales son estos. Si nos fijamos en el juego original, los movimientos posibles son:

- Mover la pieza a la izquierda.
- Mover la pieza a la derecha.
- Rotar la pieza en el sentido de las agujas del reloj.
- Dejar caer la pieza.
- Bloquear la pieza.

- Intercambiar la pieza actual por la siguiente (o por una ya cambiada anteriormente)

En el juego original cuando se dejaba caer una pieza y esta toca el suelo, se bloqueaba al instante. Sin embargo, en versiones más modernas, esto no es así y aún habiendo tocado el suelo, se permite mover la pieza con unas ciertas reglas. En este caso, se hablaría hablando de un Tetris moderno como el Tetris 99 [**<empty citation>**], donde se permite que las piezas se muevan con más libertad y por tanto, se da más capacidad de representación al jugador. Es por todo esto que se ha elegido usar esta versión del juego y no la clásica. El objetivo de esta decisión es dotar a los individuos de más capacidad de representación lo que potencialmente, debería permitirnos llegar a mejores soluciones.

Ahora, si se tiene algo de experiencia en el juego, se puede ver que no en todas las situaciones, todos los movimientos son necesarios para llegar a una solución concreta. Sin ir más lejos, en casos donde el tablero está casi vacío y no hay piezas creando agujeros, siempre se podrá poner una pieza en cualquiera de las posiciones validas con una *rotación* y un *movimiento lateral*.

Como tampoco queremos eliminar de la experimentación la posibilidad de ver los efectos que tiene el sí hacer más movimientos una vez las piezas han tocado el suelo, se han planteado cuatro sets posibles de movimientos que podrán usar los individuos:

- Simple:
  1. Mover la pieza.
  2. Rotar la pieza.
- Double:
  1. Mover la pieza.
  2. Rotar la pieza.
  3. *Dejar caer la pieza*
  4. Mover la pieza.
  5. Rotar la pieza.
- SwapSimple:
  1. Intercambiar la pieza actual.
  2. Mover la pieza.
  3. Rotar la pieza.
- SwapDouble:
  1. Intercambiar la pieza actual.
  2. Mover la pieza.
  3. Rotar la pieza.
  4. *Dejar caer la pieza*
  5. Mover la pieza.
  6. Rotar la pieza.

Donde el paso de *dejar caer la pieza* NO es configurable por el individuo (es fijo) y el resto de movimientos serán un entero que indicará cuántas veces se realiza ese movimiento. Damos ejemplos de cada uno de los movimientos:

- Intercambiar la pieza actual: $\{0, 1\}$ 0 si no se quiere intercambiar, 1 si se quiere intercambiar.
- Mover la pieza: $\{-5, 5\}$ un entero positivo o negativo. Si es positivo, se moverá esa cantidad de veces a la derecha, si es negativo, se moverá esa cantidad de veces a la izquierda (o hasta que no se pueda mover más).
- Rotar la pieza: $\{0, 1, 2, 3\}$ un entero entre 0 y 3, indicando cuántas veces se rotará la pieza en el sentido de las agujas del reloj.
- *Dejar caer la pieza*: no es configurable, la pieza caerá hasta que toque el suelo o otra pieza.
- Mover la pieza: $\{-9, ..., 9\}$, Igual que el anterior.
- Rotar la pieza: $\{0, 1, 2, 3\}$, Igual que el anterior.

Como optimización en este punto se ha propuesto lo siguiente: dado que las piezas aparecen en cierta posición concreta, el rango de movimientos laterales que se hace al principio se ha limitado a un rango de -5 a 5. Esto es, si una pieza aparece en la columna 5 del tablero, no tendría sentido moverla más de 5 veces a izquierda o derecha (ya que se saldría del tablero).

Por último, mencionar que cuando una pieza termine de ser movida se bloqueará al momento, dejándola caer hasta tocar el suelo u otra pieza y añadiéndola al tablero.

1.2 Conversión genotipo a fenotipo

Como hemos mencionado, la idea es simular el juego a partir de los movimientos establecidos para cada pieza. De tal forma que un genotipo, representado por una secuencia de enteros (movimientos), se convierta en un fenotipo, aplicando cada jugada a la secuencia de piezas predeterminada, siendo que el estado final del tablero tras colocar todas las piezas será el fenotipo asociado.

Antes de definir un genotipo, habrá que definir un set de piezas concreto y qué movimientos se pueden realizar con cada una de ellas. A estos sets o tipos se les ha llamado `Genotype`. Con ambos datos, se podrá simular el juego con los movimientos indicados en el genotipo y el tablero final será el fenotipo asociado.

Este set de piezas será el mismo para todos los individuos, y se generará aleatoriamente al inicio de la ejecución del algoritmo.

1.3 Función objetivo

Una vez sabemos como convertir un genotipo en fenotipo, es necesario definir una función objetivo que nos permita evaluar la calidad de cada solución. Dado que el objetivo es minimizar el espacio ocupado en el tablero, se ha definido la siguiente función objetivo:

$$\begin{aligned}\text{Fitness(genotipo)} = f(g) = &\ \text{factor juego} \times \text{puntuación juego} \\ &+ \text{factor penalización} \times \text{penalización} \\ &+ \text{factor heurísticas} \times \text{heurísticas}\end{aligned}$$

Desglosemos cada uno de los términos:

- **Puntuación juego**: es la puntuación obtenida tras jugar la partida con el genotipo indicado. Esta puntuación se calcula como en el juego original, donde se otorgan puntos por cada línea completada, y se otorgan puntos extra por completar varias líneas a la vez (doble, triple, tetris) [**<empty citation>**]. Se omiten puntos extra por combos o por dejar caer las piezas rápidamente, ya que no aportan nada a la calidad de la solución.
- **Penalización**: es una penalización que se aplica en caso de que el genotipo contenga movimientos inválidos o '*no-op*' como se han definido anteriormente. La penalización será proporcional al número de movimientos inválidos realizados. Estas son calculadas durante la conversión de genotipo a fenotipo, en la simulación del juego.
- **Heurísticas**: son una serie de métricas que evalúan la calidad del tablero final tras colocar todas las piezas. En este punto es donde más se puede influir en la calidad de la solución, ya que la puntuación del juego puede ser similar para tableros muy diferentes, pero las heurísticas permitirán guiar a los algoritmos hacia mejores soluciones sin necesidad de completar muchas líneas. Además, cada heurística tendrá un peso o factor asociado, que permitirá ajustar su importancia en la función objetivo.

1.3.1. Heurísticas

Se han definido las siguientes heurísticas para evaluar la calidad del tablero final. El origen de todas ellas es una mezcla entre heurísticas clásicas usadas en la literatura para jugar al Tetris con IA [**<empty citation>**], videos de youtube [**<empty citation>**] y experiencia personal.

- **Altura Agregada (Blocks)**: Suma absoluta de la cantidad de bloques que hay en el tablero.
- **Altura Ponderada (Weighted Blocks)**: Similar a la anterior, pero las columnas más altas tienen un peso mayor. Esto penaliza de forma más severa la creación de picos o torres altas en el tablero.
- **Líneas Limpiables (Clearable Lines)**: Recompensa el número de líneas completas que se pueden eliminar con una sola pieza 'I' (la línea recta). Es una métrica directa de la puntuación que se obtendría en el juego. Su Implementación está orientada a que soluciones vecinas completen más líneas al explorar.
- **Rugosidad (Roughness)**: Suma de las diferencias de altura absolutas entre columnas adyacentes. Un valor alto indica un tablero no plano, lo que crea una solución peor 'compacta' y que ocupa potencialmente más espacio.
- **Agujeros por Columna (Column Holes)**: Número de huecos columnas con agujeros. Una agujero se define como un espacio vacío que tiene al menos un bloque por encima en la misma columna. Penaliza la creación de agujeros en las soluciones.
- **Agujeros Conectados (Connected Holes)**: Número de agujeros que son adyacentes a otros agujeros. Penaliza la creación de grandes bolsas de aire que son difíciles de rellenar.
- **Bloques sobre Agujeros (Blocks Above Holes)**: Número de bloques que se encuentran directamente encima de un agujero. Penaliza fuertemente los agujeros que están enterrados, ya que dejan grandes cavidades.

- **Porcentaje de Hoyos (Pit Hole Percent)**: Porcentaje de columnas que tienen un 'Hoyo'. Un hoyo se define como una columna que tiene bloques más altos en ambas columnas adyacentes.

- **Hoyos más Profundo (Deepest Well)**: La profundidad de la columna más profunda de todas. Sería el mínimo de entre las alturas del bloque más alto de cada columna.

La combinación de estas heurísticas, cada una con su respectivo factor de ponderación, conforma la puntuación final de la heurística, como se muestra en la siguiente fórmula:

$$\text{Heurísticas} = \sum_{h \in H} w_h \times \text{score}(h)$$

Donde H es el conjunto de todas las heurísticas mencionadas, w_h es el factor de ponderación para la heurística h, y $\text{score}(h)$ es el valor calculado para dicha heurística en el tablero final. Esta función se pretende **MAXIMIZAR**.

1.3.2. Pesos de la función objetivo

Los factores que ponderan cada uno de los términos de la función objetivo, se han establecido tras una serie de pruebas preliminares, referencias en la literatura [**<empty citation>**] y 'a ojo'. Lo ideal sería poder lanzar una serie de experimentos para ajustar estos pesos, pero por limitaciones computacionales y de 'sentido' no ha sido posible.

Se dice 'sentido' porque alrededor de ajustar estos pesos, que al final no dejan de ser hiperparámetros, se podrían crear proyectos enteros. Un ejemplo sería usar técnicas metaheurísticas para ajustar estos pesos, como un algoritmo genético que optimice los pesos de las heurísticas, o incluso alguna técnica bayesiana más moderna.

Es por todo esto que al final hemos decidido fijar unos pesos 'a mano' basándonos lo mencionado anteriormente. En concreto, se han elegido los siguientes pesos:

Tabla 1.1: Pesos de los componentes principales de la función objetivo.

Componente	Peso
Factor de Puntuación del Juego	2.5
Factor de Penalización	-1.0
Factor General de Heurísticas	1.0

A su vez, los pesos para cada una de las heurísticas individuales, que componen el término de heurísticas, se detallan en la Tabla 1.2.

Tabla 1.2: Pesos para cada heurística individual.

Heurística	Peso
Blocks	-1.0
Weighted Blocks	-0.75
Clearable Lines	1.0
Roughness	-1.0
Column Holes	-5.0
Connected Holes	-2.0
Blocks Above Holes	-2.0
Pit Hole Percent	-1.0
Deepest Well	-1.0

Como vemos, gran parte de estos factores son negativos. El problema a solucionar es minimización del espacio ocupado, pero la función objetivo está planteada como maximización. La idea tras todo esto es que **estas heurísticas codifiquen el concepto de 'minimización del espacio'** y guíen la búsqueda de soluciones. Por lo tanto, se penaliza todo aquello que aleje al tablero de este ideal y se recompensa lo que lo acerque.

1.3.3. Notas

Comentar ciertos aspectos importantes a modo de resumen sobre la función objetivo:

- Como se ha mencionado, el problema consiste en la minimización del espacio ocupado. La función objetivo **codifica este concepto a través de las heurísticas**, que penalizan tableros con mucho espacio vacío, agujeros, rugosidad, etc.
- La puntuación del juego se incluye para incentivar la eliminación de líneas, que es un objetivo secundario pero relevante en el Tetris y puede ayudar a la búsqueda de soluciones óptimas, ya que al limpiar piezas, se libera espacio en el tablero.
- La penalización por movimientos inválidos, se incluye para evitar penalizar individuos con movimientos innecesarios. Ya que simulamos el juego, NO tenemos individuos inválidos, pero siempre preferimos individuos que usen todos sus movimientos a aquellos que no se intentan mover innecesariamente.

CAPÍTULO 2
Implementación

En este capítulo se describen las tecnologías y herramientas usadas para implementar las soluciones propuestas en este trabajo. Se habla del lenguaje de programación, librerías y frameworks usados, así como de la arquitectura general del sistema.

2.1 Tecnología de Implementación

El desarrollo del proyecto se ha realizado mayoritariamente en C#. Para lo que sería la parte visual se ha utilizado el framework Unity [**<empty citation>**], que permite crear aplicaciones gráficas de forma sencilla y rápida. También se ha usado Python para la visualización de resultados y generación de gráficos.

La implementación del juego se ha hecho mediante C# y Unity de forma que se han podido representar las soluciones de forma visual y dinámica. Unity permite crear escenas 2D y 3D de forma sencilla, y cuenta con una gran comunidad y documentación, por lo que para simular un juego como el Tetris, es una opción muy adecuada.

Respecto a la implementación de los algoritmos metaheurísticos, se ha optado por usar C# para mantener la coherencia con el resto del proyecto. C# es un lenguaje potente y versátil, que permite implementar algoritmos complejos de forma eficiente y sin tener tantas complicaciones técnicas como C or C++.

Los resultados de los experimentos se han almacenado en logs, de forma que se puedan analizar posteriormente.

Por último, para la visualización de estos resultados, su análisis y generación de gráficos, se ha usado Python con librerías como Matplotlib [**<empty citation>**] y Plotly [**<empty citation>**]. Estas librerías permiten crear gráficos de forma sencilla y personalizable, lo que facilita la presentación de los resultados obtenidos en los experimentos.

La Implementación de todo el código se puede encontrar en el repositorio de GitHub [**<empty citation>**].

2.2 Implementación del Tetris

Como ya se ha mencionado, la versión del Tetris utilizada es una versión moderna. Para tener control absoluto del juego y su simulación, se ha implementado el juego desde cero usando la guía oficial para el desarrollo de juegos Tetris [**<empty citation>**].

Para poder hacer la simulación del juego, se ha abstraído todo mediante clases y objetos que representan las piezas, el tablero y las reglas del juego. De esta forma, se puede

simular el juego de forma independiente de la parte visual y la parte lógica de los algoritmo.

La Implementación se ha realizado de la forma más optima posible, eliminando operaciones innecesarias y optimizando el código para que la simulación sea lo más rápida posible. Se planeaba paralelizar los experimentos, pero por limitaciones de Unity el resultado era más lento que la versión secuencial, por lo que se ha optado por dejar el paralelismo para futuros trabajos.

CAPÍTULO 3
Experimentación

3.1 Algoritmo genético

3.2 Enfriamiento simulado

CAPÍTULO 4
Resultado

4.1 Algoritmo genético

4.2 Enfriamiento simulado

4.3 Evolución

CAPÍTULO 5

Conclusiones