



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA



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

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

Índice general

Índice general	IV
Introducción	VII
0.1 Problema a resolver	VII
0.2 Dificultad del problema	VII
1 Codificación	1
1.1 Movimientos posibles y tipos de genotipo	1
1.2 Conversión genotipo a fenotipo	3
1.3 Función objetivo	4
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 Algoritmos	9
3.1 Algoritmo genético	9
3.1.1 Población inicial	9
3.1.2 Selección	10
3.1.3 Cruce	11
3.1.4 Mutación	12
3.1.5 Reemplazo	13
3.2 Enfriamiento simulado	13
3.2.1 Vecindario	13
3.2.2 Lista Tabu	15
3.2.3 Paciencia y Enfriamiento de la temperatura	15
3.2.4 Temperatura inicial	16
3.2.5 Notas	16
4 Experimentación	17
4.1 Algoritmo genético	17
4.2 Enfriamiento simulado	18
4.3 Duración experimentos	18
5 Resultado	19
5.1 Algoritmo genético	19
5.2 Enfriamiento simulado	19
5.3 Evolución	19
6 Conclusiones	21
<hr/>	
Apéndices	
A Tablas de movimientos	23
A.1 Sets de movimientos	23
A.2 Rangos de valores	23
B Resultado de la experimentación	25

B.1	Algoritmo genético	25
B.1.1	Evolución general	25
B.1.2	Efecto del tipo de individuo (movimientos)	25
B.1.3	Efecto de la probabilidad de mutación	26
B.1.4	Efecto del tamaño de población	27
B.2	Enfriamiento simulado	28
B.2.1	Evolución general	28
B.2.2	Efecto del tipo de individuo (movimientos)	28
B.2.3	Efecto del tamaño de la lista tabú	29
B.2.4	Efecto del factor de enfriamiento	29

Introducción

0.1 Problema a resolver

Minimización de la cobertura con restricciones

0.2 Dificultad del problema

Escala con el número de piezas. 10 es 'fácil', 20 'medio' y 30 'difícil'. Más no por limitaciones temporales y computacionales.

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 que puede hacer un jugador son:

- Mover la pieza a la izquierda.
- Mover la pieza a la derecha.
- Rotar la pieza en el sentido de las agujas del reloj.
- Rotar la pieza en sentido contrario a 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 las versiones originales, 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 para abordar el problema. 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:

Tipo	Secuencia de movimientos
Simple	1. Mover la pieza 2. Rotar la pieza
Double	1. Mover la pieza 2. Rotar la pieza 3. <i>Dejar caer la pieza</i> 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. <i>Dejar caer la pieza</i> 5. Mover la pieza 6. Rotar la pieza

Tabla 1.1: Sets de movimientos posibles para los individuos.

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. Los rangos permitidos para cada tipo de movimiento se detallan en la Tabla 1.2.

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). Una vez esta haya caído, el rango de movimientos a considerar ha de ser mayor, ya que la pieza puede estar en cualquier posición del tablero. Por ello, el rango de movimientos laterales tras dejar caer la pieza se ha establecido entre -9 y 9.

Movimiento	Rango y descripción
Intercambiar la pieza actual	$\{0, 1\}$ 0 si no se quiere intercambiar, 1 si se quiere intercambiar.
Mover la pieza (inicial)	$\{-5, \dots, 5\}$ 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 (inicial)	$\{0, 1, 2, 3\}$ Un entero entre 0 y 3, indicando cuántas veces se rotará la pieza en el sentido de las agujas del reloj.
<i>Dejar caer la pieza</i>	No configurable La pieza caerá hasta que toque el suelo o otra pieza.
Mover la pieza (final)	$\{-9, \dots, 9\}$ Igual que el movimiento lateral inicial.
Rotar la pieza (final)	$\{0, 1, 2, 3\}$ Igual que la rotación inicial.

Tabla 1.2: Rangos de valores permitidos para cada tipo de movimiento.

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:

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

Esta función se pretende **MAXIMIZAR**. 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. También se omiten los T-spins [**<empty citation>**], de nuevo por no aportar nada a la calidad de la solución.
- **Penalización:** es una penalización que se aplica en caso de que el genotipo contenga movimientos que no hagan nada 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. La idea principal tras estas, es codificar el objetivo de 'minimización' del espacio ocupado y guiar la búsqueda hacia soluciones buenas.

- **Bloques ocupados (Blocks):** Suma absoluta de la cantidad de bloques ocupados que hay en el tablero.
- **Altura Ponderada (Weighted Blocks):** Similar a la anterior, pero las filas más altas tienen un peso mayor. Esto penaliza de forma más severa la creación de picos o torres altas en el tablero. De esta forma, los bloques en la primera fila tienen un peso de 1, los de la segunda fila un peso de 2, y así sucesivamente.

- **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, más que a la solución actual.
- **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 columnas con agujeros. Un 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 (en la misma columna). 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 respecto a agujeros (Pit Hole Percent):** Porcentaje de 'hoyos' contra agujeros. Un hoyo se define como una columna que tiene bloques más altos en ambas columnas adyacentes. Este porcentaje se calcula como:

$$\frac{\text{Hoyos}}{\text{Hoyos} + \text{Agujeros}}$$

La idea tras esta métrica es penalizar tableros con muchos agujeros que no sean 'hoyos', ya que los hoyos son más fáciles de rellenar. Es también una métrica de dispersión de los agujeros en el tablero.

- **Hoyos más Profundo (Deepest Well):** La profundidad de la columna más profunda de todas. Sería el mínimo del máximo (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.

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:

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

Tabla 1.3: Pesos de los componentes principales de la función objetivo.

A su vez, los pesos para cada una de las heurísticas individuales, que componen el término de heurísticas, se detallan en la Tabla 1.4.

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

Tabla 1.4: Pesos para cada heurística individual.

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 individuos con movimientos innecesarios. Ya que simulamos el juego, NO tenemos individuos inválidos, pero siempre preferimos un genotipo que use todos sus movimientos a aquel que no se intenta 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 en Python. Estos logs contienen información relevante como la puntuación obtenida, el tiempo de ejecución, los parámetros usados, etc.

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 algoritmos.

La Implementación se ha realizado de la forma más óptima 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.

De igual forma, el cálculo de la función objetivo y las heurísticas se ha optimizado al máximo, evitando cálculos redundantes y almacenando resultados intermedios cuando es posible. Por ejemplo, en el algoritmo genético se evitaba re-calcular la función objetivo para individuos que no hayan sido reemplazados entre generaciones, más detalles en el capítulo de su Implementación.

Con todo esto, se ha conseguido una Implementación eficiente y robusta del Tetris, que permite simular el juego de forma rápida y precisa de forma independiente a los algoritmos. Todo esto, facilita la experimentación con los algoritmos metaheurísticos propuestos y proporciona una base sólida para futuras mejoras o reutilización de partes del proyecto gracias a su modularidad.

CAPÍTULO 3

Algoritmos

En esta sección, se presentan los algoritmos implementados y las decisiones de diseño tomadas durante su desarrollo. La sección se centra en las técnicas y operadores utilizados de cada algoritmo, más que en la descripción de estos.

El proceso de desarrollo incluyó una exploración manual inicial de parámetros y técnicas no documentada, motivada por las limitaciones temporales del curso. Esta fase preliminar ha servido para acotar la experimentación exhaustiva que se ha hecho más tarde, lo que ha permitido identificar los enfoques que prometen los mejores resultados.

Las decisiones finales sobre los algoritmos y el razonamiento de su elección se detallan aquí, mientras que los parámetros específicos que se usaron en la experimentación exhaustiva se explican en la sección de Experimentación. No se pretende omitir todo lo experimentado, sino que se hablará del porqué de las decisiones junto con las alternativas consideradas.

Además, ya que esta memoria tiene como principal objetivo el ser evaluada por el profesor responsable de la asignatura, no se entrará en detalles básicos sobre el funcionamiento de los algoritmos, sino que se explicarán la forma en la que ha implementado cada parte de ellos.

3.1 Algoritmo genético

El algoritmo genético (Genetic Algorithm, GA) [**<empty citation>**] se ha implementado siguiendo los principios básicos de esta técnica de optimización. Siguiendo lo mencionado anteriormente, se habla de como se llevan a cabo los pasos del algoritmo, en concreto: de la Población inicial, el Cruce, la Mutación, la Selección y el Reemplazo.

3.1.1. Población inicial

En esta sección se habla de los pasos a seguir para generar individuos nuevos. En la solución final propuesta, solo se generan nuevos individuos al inicio. Sin embargo, se ha experimentado incluyendo nuevos individuos en cada generación, pero los resultados no han sido concluyentes por falta de pruebas más exhaustivas. Es por ello que se ha optado por una población inicial fija que cambia únicamente vía cruce.

El número de individuos generados es un hiperámetro y se mantendrá constante durante toda la ejecución del algoritmo. A su vez, hay que definir el tipo de individuo a generar (el set de movimientos) junto con el número de piezas con las que se hará la optimización.

Una vez definidos estos parámetros, se generan los N individuos de forma independiente rellenando sus sets de movimientos. Vamos a indicar como se haría para el tipo de individuo `SwapDouble`, el resto de tipos son iguales pero con menos movimientos.

1. Para cada individuo i desde 1 hasta N :
 - a) Crear un nuevo *individuo_i* vacío
 - b) Para cada pieza j desde 1 hasta P :
 - 1) Generar $swap \leftarrow$ entero aleatorio en $\{0, 1\}$
 - 2) Generar $movLateralInicial \leftarrow$ entero aleatorio en $\{-5, \dots, 5\}$
 - 3) Generar $rotacionInicial \leftarrow$ entero aleatorio en $\{0, 1, 2, 3\}$
 - 4) Generar $movLateralFinal \leftarrow$ entero aleatorio en $\{-9, \dots, 9\}$
 - 5) Generar $rotacionFinal \leftarrow$ entero aleatorio en $\{0, 1, 2, 3\}$
 - 6) Añadir todos estos movimientos a *individuo_i* para la pieza j
2. Devolver la población con los N individuos generados

Recordamos que cada movimiento está acotado a un rango concreto para evitar la mayor cantidad de movimiento inválidos posibles A. Aún así, como se ha mencionado en la sección de la función objetivo, se penalizan los *no-ops* para evitar que los individuos generen movimientos innecesarios.

3.1.2. Selección

Para la selección se ha optado por probar tanto el método elitista como el método de tarta con probabilidad.

Para el método elitista se probaron varios tamaños de élite, pero a la larga no se conseguían buenas soluciones, quedando el algoritmo rápidamente estancado en óptimos locales.

Por otro lado, el método de tarta parecía llegar a buenos resultados de forma consistente, por lo que se ha optado por usarlo en la solución final. En la selección por tarta, se seleccionan dos padres para el cruce de forma proporcional a su fitness. Es decir, cuanto mejor sea el fitness de un individuo, más probabilidad tendrá de ser seleccionado como padre.

A todos los individuos se les asigna una probabilidad usando la función Softmax [**<empty citation>**]. Esta función transforma los valores de fitness en probabilidades entre 0 y 1, de forma que la suma de todas las probabilidades sea 1.

Al inicio se ha optado por usar la Softmax 'pura', pero se ha visto que los individuos con mejor fitness acaparaban casi toda la probabilidad de ser seleccionados, por lo que se ha optado por escalar con la versión 'temperatura' de la Softmax. Esta versión introduce un parámetro de temperatura T que controla la 'suavidad' de la distribución de probabilidades. A mayor T , más uniforme será la distribución (más exploración), y a menor T , más se acentuará la diferencia entre los individuos, seleccionando solo los mejores (más explotación).

Haciendo varias pruebas se ha optado por una escala de decremento de la temperatura con decremento logarítmico. Esta sigue la formula:

$$temp = \max(0, 1, \frac{InitialTemp}{\log(generationI + 2)})$$

Se han probado distintas temperaturas iniciales, pero al final se ha optado por una temperatura MUY alta que favorece la exploración de $InitialTemp = 100$.

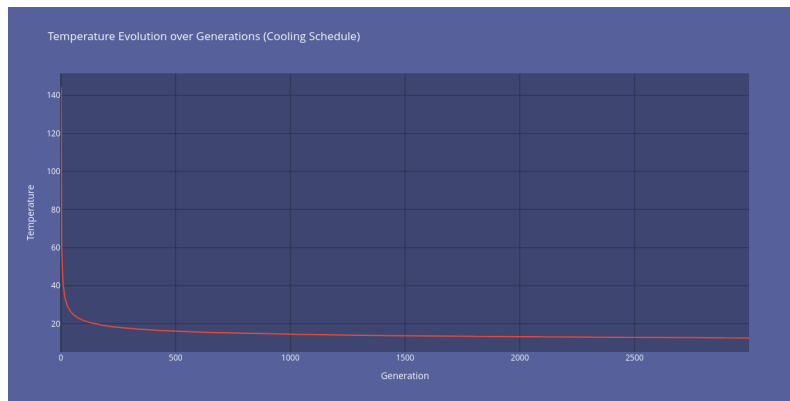


Figura 3.1: Evolución de la temperatura a lo largo de las generaciones.

Como vemos en la gráfica, siempre usamos valores muy elevados de temperatura. A lo largo de la memoria se hará más incapié en este punto, pero la EXPLORACIÓN es lo que más ha beneficiado a la generación de buenas soluciones. En esa línea se seleccionan más parámetros en los siguientes puntos.

3.1.3. Cruce

El cruce parte de poder usar la función de selección para elegir dos padres. Con ellos, se generan dos hijos usando el operador que mejores resultados nos ha dado. Este operador junta la mitad de las piezas de cada padre para generar un hijo. El proceso es el siguiente:

1. Seleccionar dos padres $padre_1$ y $padre_2$ usando la función de selección.
2. Crear hijo $hijo_1$ usando la primera mitad de 'piezas' del $padre_1$ y la segunda mitad de 'piezas' de $padre_2$.
3. Crear hijo $hijo_2$ usando la primera mitad de 'piezas' del $padre_2$ y la segunda mitad de 'piezas' del $padre_1$.
4. Mutar con probabilidad $p_{mutacion}$ hijo $hijo_1$ (ver sección de mutación).
5. Mutar con probabilidad $p_{mutacion}$ hijo $hijo_2$ (ver sección de mutación).
6. Devolver $hijo_1$ y $hijo_2$.

Cuando se dice 'piezas', se hace referencia a todos los movimientos asociados a cada pieza concreta. Por ejemplo, en el tipo de individuo Simple, cada pieza tiene dos movimientos asociados (movimiento lateral inicial, rotación inicial). Por lo tanto, si se está optimizando con diez piezas al hacer el cruce, se copian todos los movimientos asociados a las piezas $\{1 \dots 5\}$ del $padre_1$ y las $\{6 \dots 10\}$ del padre $padre_2$ para generar $hijo_1$, mientras que se copian los movimientos de las piezas $\{1 \dots 5\}$ del $padre_2$ y las $\{6 \dots 10\}$ del $padre_1$ para generar $hijo_2$.

Ya que ambos padres se codifican como una secuencia de enteros, se podría explicar como

$$\begin{aligned} padre_1 &= [m_{1,1}, m_{1,2}, m_{2,1}, m_{2,2} \dots, m_{10,1}, m_{10,2}] \\ padre_2 &= [m'_{1,1}, m'_{1,2}, m'_{2,1}, m'_{2,2} \dots, m'_{10,1}, m'_{10,2}] \\ hijo_1 &= [m_{1,1}, m_{1,2}, \dots, m_{5,1}, m_{5,2}, m'_{6,1}, m'_{6,2} \dots, m'_{10,1}, m'_{10,2}] \\ hijo_2 &= [m'_{1,1}, m'_{1,2}, \dots, m'_{5,1}, m'_{5,2}, m_{6,1}, m_{6,2} \dots, m_{10,1}, m_{10,2}] \end{aligned}$$

Por la naturaleza del problema, es apropiado hacer un cruce a nivel de **piezas continuas**, y no elegir piezas alternadamente de un padre y de otro. La razón tras esto es la dependencia que hay entre una pieza y todas aquellas colocadas a continuación. Si se mezclan piezas de ambos padres de forma alternada, se puede romper la coherencia entre las piezas y generar soluciones peores con mayor probabilidad.

Esto se ha descubierto tras probar ambos métodos, y ver que el cruce por piezas continuas generaba mejores soluciones de forma consistente.

También se ha probado creando solo un hijo en lugar de dos y / o NO usando ningún padre (es decir solo mutar). A la larga ninguna otra combinación ha dado mejores resultados que la propuesta, por lo que se ha optado por esta.

3.1.4. Mutación

La mutación se aplica al individuo resultante del cruce de dos padres con una probabilidad determinada. De nuevo, este será un hiperámetro a definir en la experimentación y cada hijo resultante de un cruce se selecciona para ser mutado de forma independiente con probabilidad $p_{mutacion}$.

Una vez un individuo ha sido elegido para ser mutado, se hace lo siguiente:

1. Seleccionar una pieza al azar del individuo.
2. Seleccionar un movimiento al azar dentro de esa pieza.
3. Mutar el movimiento seleccionado generando un nuevo 'valor aleatorio total' dentro del rango permitido para ese movimiento.

Como ya se ha comentado anteriormente, para cada tipo de movimiento hay un rango permitido (véase apéndice A). Se han planteado dos formas de elegir el nuevo valor del movimiento:

- **Mutación aleatoria total:** generar un nuevo valor aleatorio dentro del rango permitido para ese movimiento.
- **Mutación cercana:** generar un nuevo valor cercano al valor actual del movimiento.

Cuando se dice 'mutación cercana', se hace referencia al hecho de NO variar el valor actuar en más de uno y solo elegir 'vecinos inmediatos'. Por ejemplo, si el movimiento es un desplazamiento lateral de 3, solo se mutaría a 2 o 4 en la versión cercana, pero en la aleatoria total, se generaría un nuevo valor aleatorio entre -5 y 5 (en este caso).

La razón de aplicar la estrategia de mutación aleatoria total, es aumentar la diversidad de la población y poder escapar de óptimos locales con mayor probabilidad. Se han probado ambos formatos y la generación de soluciones buenas se veía muy ralentizada al aplicar mutaciones cercanas (costaba salir de óptimos locales). Es por ello que se ha optado por usar mutaciones aleatorias totales en la solución final.

3.1.5. Reemplazo

El reemplazo, como ya se sabe, consiste en decidir qué individuos de la generación actual pasan a la siguiente y cuales son reemplazados.

Sin muchas sorpresas, se ha optado por un reemplazo generacional parcial, donde la mitad peor de individuos es totalmente reemplazada por aquellos creados mediante el cruce y la mutación.

Ya que por cada cruce se generan dos hijos, se realizan $N/4$ cruces para generar $N/2$ nuevos individuos que reemplazan a la mitad peor de la población actual.

Cabe destacar que además, a la hora de evaluar las fitness de la nueva generación y ordenar los individuos, nos ahorramos la mitad de los cálculos al haber mantenido la mitad de la generación anterior. Esto es, solo se evalúan los $N/2$ nuevos individuos y se combinan con los $N/2$ individuos de la generación anterior, cuyos fitness ya se conocen.

3.2 Enfriamiento simulado

De forma similar a como se ha hecho en el apartado anterior, el Enfriamiento simulado (Simulated Annealing, SA) [**<empty citation>**] se ha implementado siguiendo los principios básicos de esta técnica de optimización.

Siguiendo lo mencionado en la introducción a este apartado, se habla de como se llevan a cabo los pasos del algoritmo, en concreto: la definición de Vecindario, la lista Tabu, la Paciencia y el Enfriamiento de la temperatura.

Primero que todo mencionar que, a diferencia del algoritmo genético, el Enfriamiento simulado parte de un único individuo inicial y va explorando soluciones vecinas a este individuo. Por lo tanto, no hay una población de individuos, sino un único individuo que se va actualizando a lo largo de la ejecución del algoritmo.

Este individuo es generado de forma aleatoria al inicio, similar a como se ha hecho en la sección de Población inicial del algoritmo genético. Se plantea más abajo el usar como individuo inicial el mejor individuo encontrado por el algoritmo genético, pero por limitaciones de tiempo no se ha podido implementar.

Recordar también que el algoritmo de Enfriamiento simulado tiene un parámetro de temperatura que controla la probabilidad de aceptar soluciones peores a la actual. Si una mejor solución aparece SIEMPRE la elegimos, pero si no, se acepta con una probabilidad que depende de la temperatura, de forma que se pueda escapar de óptimos locales. Ya que estamos maximizando, la probabilidad de aceptar en estos casos se calcula como:

$$P(acceptar) = \min(1, e^{\frac{\Delta fitness}{temp}})$$

Donde $\Delta fitness$ es la diferencia de fitness entre la solución actual y la nueva solución vecina, y $temp$ es la temperatura actual del sistema.

3.2.1. Vecindario

Una vez tenemos este individuo inicial, es necesario definir cómo se generan soluciones vecinas a partir de él.

Dada la naturaleza del problema y la codificación elegida, no hay otra forma de definir el vecindario que no sean aquellos individuos cuyos valores en el genotipo difieran

en un único movimiento. Es decir, se generan soluciones vecinas mutando un único movimiento del individuo actual.

La primera implementación de este vecindario se hizo usando la mutación cercana, es decir, generando un nuevo valor cercano al valor actual del movimiento. Sin embargo, tras varias pruebas se vio que este método no generaba soluciones vecinas suficientemente diversas, por lo que se optó por usar la mutación aleatoria total, generando un nuevo valor aleatorio dentro del rango permitido para ese movimiento. *Se observa lo mismo que antes, la exploración limitada de mutaciones cercanas no permite escapar de óptimos locales.*

Con algo más de experimentación se vio que este método generaba buenas soluciones, pero parecía saltarse algunas posibles intermedias o se quedaba estancado sin saltar de individuo. Por ello, se ha terminado optando por generar una lista de todos los movimientos posibles para cada pieza (según el tipo de individuo) e iterar sobre todas ellas para generar soluciones vecinas de forma ordenada. Cuando todas las soluciones de una pieza han sido exploradas, se pasa a la siguiente pieza y se repite el proceso hasta haber explorado todas las piezas.

La forma en la que se generan esta lista es simple y compleja a la vez, ya que se hace optimizando el espacio y no generando de verdad todas las combinaciones. Partimos de saber todos los valores posibles para cada movimiento (ver Apéndice A), así pues es posible indexar en una lista los valores que puede tener cada movimiento. Por ejemplo, si estamos en SwapSimple, los posibles valores son

$$[0, 1, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 0, 1, 2, 3]$$

Donde los primeros dos valores corresponden al intercambio de pieza, los siguientes once al movimiento lateral inicial, y los últimos cuatro a la rotación inicial.

Ya que estos M movimientos son fijos, y se tienen N piezas, se puede iterar de 0 a $N \times M - 1$ con un índice idx y calcular a qué pieza y movimiento corresponde dicho valor, intercambiarlo generando un nuevo individuo y evaluarlo. El cálculo es el siguiente:

```

1 Genotype getGenotypeFromIndex(Genotype baseGenotype, int idx) {
2     // Create a neighbor by mutating the current genotype
3     // First compute indices for piece, movement type for that piece and value
4     int pieceIndex = idx % nPieces;
5     int moveTypeIndex = (idx / nPieces) % allMovements.Length;
6     int valueIndex = idx / (nPieces * allMovements.Length);
7
8     // Ensure valueIndex is within bounds
9     valueIndex = valueIndex % allMovements[moveTypeIndex].Length;
10
11    // Generate new genotype by mutating the selected movement
12    return baseGenotype.mutateAtCopy(
13        pieceIndex,
14        moveTypeIndex,
15        allMovements[moveTypeIndex][valueIndex]
16    );
17 }
```

Con esto, cada vez que se itera con idx , se obtiene la pieza y el movimiento a modificar, junto con el nuevo valor que se asignará. Con esos tres datos, se pueden generar nuevos genotipos mutando el genotipo base.

Así pues, se consigue explorar todas las soluciones vecinas de forma ordenada. Si al terminar de explorar todas los vecinos NO se ha encontrado una mejor solución, se

reinicia el índice idx a 0 y se vuelve a empezar con una temperatura mayor (ver sección de Enfriamiento de la temperatura).

También, mencionar que para añadir estocasticidad a la exploración, no se empieza por $idx = 0$, sino que se genera un valor aleatorio entre 0 y $N \times M - 1$ para empezar la exploración desde ahí. De esta forma, se evita explorar siempre las mismas soluciones en el mismo orden, lo que podría llevar a estancamientos.

3.2.2. Lista Tabu

Debido a la naturaleza del vecindario y la forma en la que se exploran las soluciones vecinas, es posible que el algoritmo vuelva a explorar soluciones ya visitadas. Además, por las dependencias que se crean entre las primeras y últimas piezas, podemos intuir que el espacio de soluciones está lleno de valles gigantes en los que se puede ver estancado el algoritmo.

Para mitigar esto, se ha implementado una lista Tabu que almacena los últimos K individuos visitados. De esa forma, cuando se vaya a explorar una nueva solución, se comprobará si ya ha sido visitada anteriormente. En tal caso, se descartará y se generará una nueva solución en su lugar.

El tamaño de la lista Tabu K es un hiperámetro a definir en la experimentación. Se han probado distintos valores para esta lista, en la fase de experimentación se definen cuales han sido los candidatos para experimentar.

3.2.3. Paciencia y Enfriamiento de la temperatura

La idea principal del enfriamiento simulado, es definir un parámetro de exploración / explotación que vaya variando a lo largo de la ejecución. El valor de este parámetro afecta directamente a la probabilidad de aceptar soluciones peores a la actual, de forma que se pueda escapar de óptimos locales y no quedarse atrapado en ellos.

Después de probar distintas formas de variar valor, se ha optado por usar una temperatura inicial baja que vaya decreciendo linealmente a lo largo de la ejecución con un factor de enfriamiento (también hiperparámetros). En cada iteración, el valor de la temperatura se reduce con la fórmula:

$$temp_i = \max(0,1, \quad temp_{i-1} \times (1 - coolingFactor))$$

Ahora, no podemos siempre decrementar este valor, hay que ser conscientes de si el algoritmo se está quedando estancado o no. La forma de medir este estancamiento se controla mediante una 'paciencia'.

Si durante $T_{paciencia}$ iteraciones no se encuentra una mejor solución a la actual (que no mejor solución global), se puede considerar que el algoritmo está estancado y en cada iteración se aumenta la temperatura con la siguiente fórmula:

$$temp_i = \max(0,1, \quad temp_{i-1} \times (1 + coolingFactor))$$

De esta forma, si el algoritmo se queda estancado, se aumenta la temperatura para favorecer la exploración y poder escapar de óptimos locales. Mientras se supere la paciencia, se aumenta la temperatura en cada iteración hasta que se encuentre una mejor solución. En tal punto, se reinicia el contador de paciencia y se vuelve a decrementar la temperatura en cada iteración.

3.2.4. Temperatura inicial

Tras diversas pruebas y ver el comportamiento del algoritmo, se ha visto que con temperaturas medianas / elevadas el algoritmo parece elegir y saltar a soluciones peores con demasiada frecuencia y no consigue 'nada' hasta que el valor no baja de 1.

Es así como se ha optado por elegir $temp_{initial} = 2$, de forma que el algoritmo tenga iteraciones suficientes para explorar, pero no haga una gran cantidad de saltos innecesarios sin encontrar un camino por el que poder avanzar.

3.2.5. Notas

Comentar ciertos aspectos importantes a modo de resumen sobre el enfriamiento simulado:

- Se usan dos mecanismos para evitar estancamientos: la lista Tabu y la paciencia con el aumento de temperatura.
- El vecindario se explora de forma ordenada para evitar saltarse soluciones intermedias y asegurar una exploración completa.
- Durante el algoritmo se guardan dos individuos: el actual y el mejor encontrado hasta el momento. La temperatura afecta a la probabilidad de aceptar soluciones peores que el actual, pero el mejor encontrado se mantiene siempre separado.
- Se han definido dos paciencias, una para el aumento de temperatura y otra para el fin del algoritmo (ver sección de Experimentación). La primera paciencia NO depende de la mejor solución global, sino de la que se esté considerando actualmente. Ambas paciencias son hiperparámetros a definir en la experimentación, de nuevo, consultar la sección de Experimentación para más detalles sobre qué valores se les ha asignado.

CAPÍTULO 4

Experimentación

En este capítulo se describen los experimentos realizados para evaluar el rendimiento de los algoritmos implementados. Se detallan los parámetros probados de cada algoritmo, en que rango se han explorado y las decisiones tomadas para elegir los valores finales usados en la sección de Resultados.

Como se ha mencionado, la primera acotación de los parámetros a probar se hizo de forma manual y no documentada. Sin embargo, esta fase preliminar ha servido para acotar la experimentación exhaustiva que se plantea en este apartado, lo que ha permitido identificar los enfoques que prometen los mejores resultados.

4.1 Algoritmo genético

Una vez definidas las partes principales del algoritmo genético, es necesario definir los parámetros a probar y los valores que se han explorado para cada uno de ellos.

Para cada tipo de dificultad: 10 piezas, 20 piezas y 30 piezas (Fácil, Medio y Difícil), se han probado los siguientes parámetros:

Parámetro	Valores probados
Tipo de individuo	Simple, Double, SwapSimple, SwapDouble
Tamaño de la población	10000, 20000, 30000 individuos
Probabilidad de mutación	5 %, 15 %, 25 %

Tabla 4.1: Parámetros probados para el algoritmo genético

En total, con estos parámetros, se han realizado $4 \times 3 \times 3 = 36$ experimentos para cada tipo de dificultad, por lo que en total han sido, SOLO para el algoritmo genético, $36 \times 4 = 144$ experimentos.

Cada experimento se ha ejecutado durante un máximo de 3000 generaciones y con una paciencia de 400 generaciones. En caso de no conseguir una mejor solución tras 400 generaciones o en caso de superar las 3000, se ha registrado el mejor individuo encontrado hasta el momento como solución final.

Durante cada iteración se ha registrado el mejor fitness de la generación actual, junto con la temperatura en ese momento. De esta forma, se puede analizar la evolución del algoritmo a lo largo del tiempo.

4.2 Enfriamiento simulado

De forma similar a como se ha hecho en el apartado anterior, para el Enfriamiento simulado se han probado los siguientes parámetros para cada tipo de dificultad: 10 piezas, 20 piezas y 30 piezas (Fácil, Medio y Difícil):

Parámetro	Valores probados
Tipo de individuo	Simple, Double, SwapSimple, SwapDouble
Factor de enfriamiento	0,00005, 0,0005, 0,005, 0,05
Tamaño lista tabú	5, 10, 20

Tabla 4.2: Parámetros probados para el enfriamiento simulado

Para la paciencia que maneja el incremento de la temperatura, se ha optado por un valor dinámico que depende del número de piezas y de la cantidad de movimientos por pieza. La fórmula usada es simple:

$$T_{paciencia} = nPiezas \times nMovimientosPorPieza$$

De esta forma, lo que nos aseguramos es de visitar todos los vecinos al menos una vez antes de considerar que el algoritmo está estancado y aumentar la temperatura.

En total, con estos parámetros, se han realizado $4 \times 3 \times 3 = 36$ experimentos para cada tipo de dificultad, por lo que en total han sido, SOLO para el enfriamiento simulado, $36 \times 4 = 144$ experimentos.

Cada experimento se ha ejecutado durante un máximo de 100 000 generaciones y con una paciencia de 10 000 generaciones. En caso de no conseguir una mejor solución tras 10 000 generaciones o en caso de superar las 100 000, se ha registrado el mejor individuo encontrado hasta el momento como solución final.

Durante cada iteración se ha registrado el mejor fitness de la generación actual, junto con la temperatura en ese momento. De esta forma, se puede analizar la evolución del algoritmo a lo largo del tiempo.

4.3 Duración experimentos

En total se han realizado $144 + 144 = 288$ experimentos, cada uno con distintas duraciones dependiendo de los parámetros usados.

Todos los experimentos se han ejecutado en mi máquina personal. Las máquina tiene como procesador un CPU Ryzen 9 9000 con 64GB RAM, por lo que se puede considerar una capacidad de cómputo alta. Ahora, el tiempo de ejecución de cada experimento varía mucho dependiendo del tipo de algoritmo, el tipo de individuo y la cantidad de piezas a optimizar.

Por ejemplo, los experimentos del algoritmo genético con 10 piezas tardan entre 1 y 15 minutos en completarse, mientras que los experimentos con 20 y 30 piezas pueden tardar entre 10 y 120 minutos, variando también según la cantidad de individuos y si converge o no antes de llegar al límite de generaciones.

Para el algoritmo de enfriamiento simulado, los tiempos son generalmente más cortos, oscilando entre 1 y 10 minutos para 10 piezas, y entre 5 y 20 minutos para 20 y 30 piezas. Esto se debe a la naturaleza más simple de los movimientos y la estructura del algoritmo en comparación con el algoritmo genético.

CAPÍTULO 5

Resultado

En este apartado mostramos los resultados obtenidos tras la experimentación exhaustiva realizada. Se presentan comparaciones de como cada tipo de parámetro afecta a las soluciones finales (a su fitness). También se presentan los mejores resultados obtenidos para cada tipo de dificultad y se comparan los rendimientos de los dos algoritmos implementados.

No mostraremos todas las gráficas obtenidas en este apartado, estas estarán en el anexo B para aquellos interesados en verlas todas. Aquí se mostrarán las más relevantes y aquellas que ayuden a entender el comportamiento de los algoritmos.

5.1 Algoritmo genético

5.2 Enfriamiento simulado

5.3 Evolución

CAPÍTULO 6

Conclusiones

APÉNDICE A

Tablas de movimientos

En este apéndice se incluyen las tablas completas que detallan los sets de movimientos posibles y los rangos de valores para cada tipo de movimiento. Estas tablas se presentan originalmente en el Capítulo 2 (Codificación), sección 2.2 (Movimientos posibles y tipos de genotipo).

A.1 Sets de movimientos

La Tabla A.1 muestra los cuatro tipos de sets de movimientos que pueden usar los individuos en el algoritmo genético.

Tipo	Secuencia de movimientos
Simple	1. Mover la pieza 2. Rotar la pieza
Double	1. Mover la pieza 2. Rotar la pieza 3. <i>Dejar caer la pieza</i> 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. <i>Dejar caer la pieza</i> 5. Mover la pieza 6. Rotar la pieza

Tabla A.1: Sets de movimientos posibles para los individuos (ver Tabla 1.1 en el Capítulo 2).

A.2 Rangos de valores

La Tabla A.2 detalla los rangos de valores permitidos para cada tipo de movimiento, incluyendo una descripción completa de su comportamiento.

Movimiento	Rango y descripción
Intercambiar la pieza actual	$\{0, 1\}$ 0 si no se quiere intercambiar, 1 si se quiere intercambiar.
Mover la pieza (inicial)	$\{-5, \dots, 5\}$ 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 (inicial)	$\{0, 1, 2, 3\}$ Un entero entre 0 y 3, indicando cuántas veces se rotará la pieza en el sentido de las agujas del reloj.
<i>Dejar caer la pieza</i>	No configurable La pieza caerá hasta que toque el suelo o otra pieza.
Mover la pieza (final)	$\{-9, \dots, 9\}$ Igual que el movimiento lateral inicial.
Rotar la pieza (final)	$\{0, 1, 2, 3\}$ Igual que la rotación inicial.

Tabla A.2: Rangos de valores permitidos para cada tipo de movimiento (ver Tabla 1.2 en el Capítulo 2).

APÉNDICE B

Resultado de la experimentación

En este apéndice se incluyen todas las gráficas obtenidas durante la experimentación exhaustiva realizada. Estas gráficas muestran cómo cada tipo de parámetro afecta a las soluciones finales (a su fitness) para ambos algoritmos implementados.

B.1 Algoritmo genético

B.1.1. Evolución general

A continuación se muestran las gráficas de evolución del algoritmo genético. La Figura B.1 muestra la evolución del fitness a lo largo de las generaciones, mientras que la Figura ?? muestra cómo evoluciona la temperatura del algoritmo.

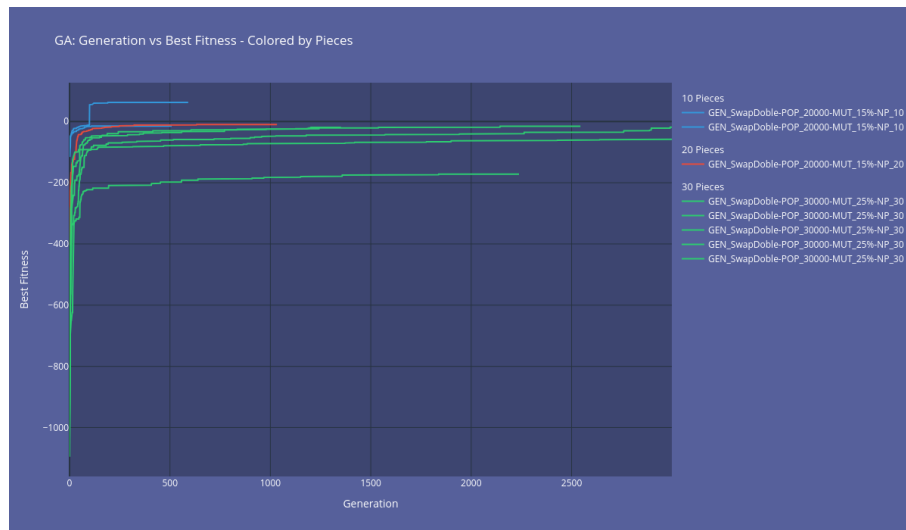
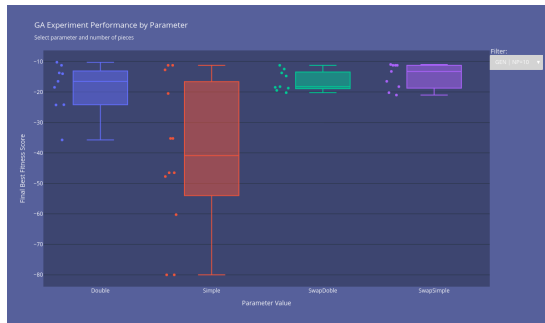


Figura B.1: Evolución del fitness del algoritmo genético a lo largo de las generaciones.

B.1.2. Efecto del tipo de individuo (movimientos)

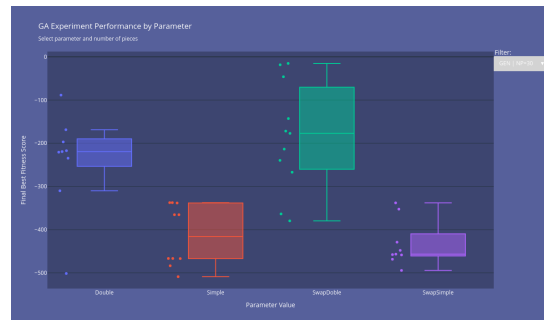
Las siguientes gráficas muestran cómo el tipo de individuo (Simple, Double, Swap-Simple, SwapDouble) afecta al fitness final obtenido para diferentes cantidades de piezas.



(a) 10 piezas (Fácil)



(b) 20 piezas (Medio)

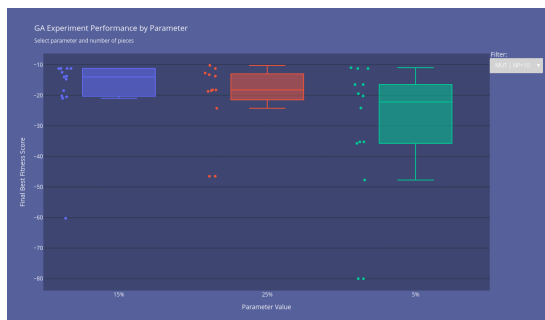


(c) 30 piezas (Difícil)

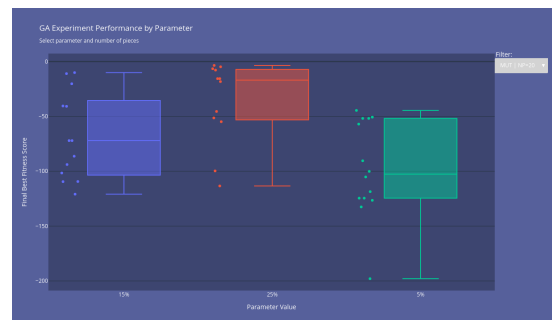
Figura B.2: Efecto del tipo de individuo en el algoritmo genético para diferente N° de piezas.

B.1.3. Efecto de la probabilidad de mutación

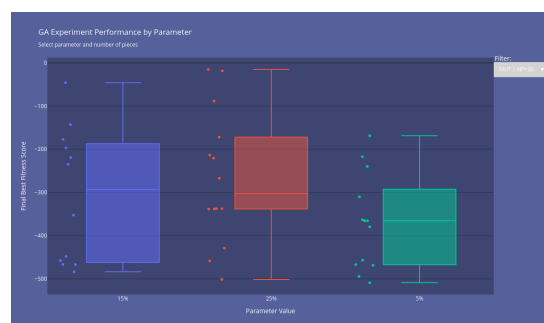
Las siguientes gráficas muestran cómo la probabilidad de mutación (5 %, 15 %, 25 %) afecta al fitness final obtenido para diferentes cantidades de piezas.



(a) 10 piezas (Fácil)



(b) 20 piezas (Medio)

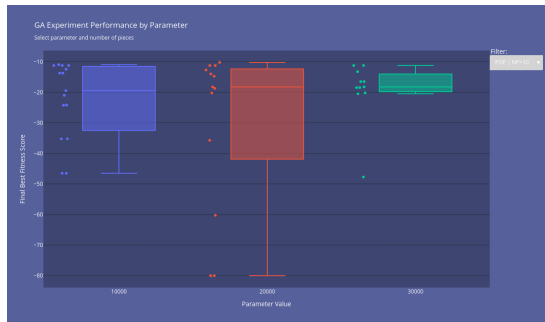


(c) 30 piezas (Difícil)

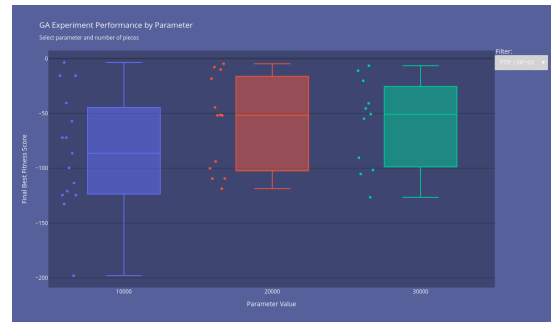
Figura B.3: Efecto de la % de mutación en el algoritmo genético para diferente N° de piezas.

B.1.4. Efecto del tamaño de población

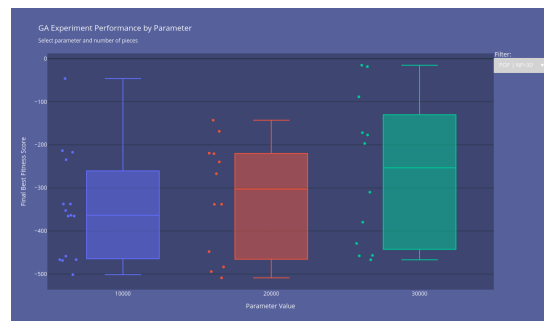
Las siguientes gráficas muestran cómo el tamaño de la población (10000, 20000, 30000 individuos) afecta al fitness final obtenido para diferentes cantidades de piezas.



(a) 10 piezas (Fácil)



(b) 20 piezas (Medio)



(c) 30 piezas (Difícil)

Figura B.4: Efecto del tamaño de población en el algoritmo genético para diferente N° de piezas.

B.2 Enfriamiento simulado

B.2.1. Evolución general

A continuación se muestra la gráfica de evolución del algoritmo de enfriamiento simulado. La Figura B.5 muestra la evolución del fitness a lo largo de las iteraciones.

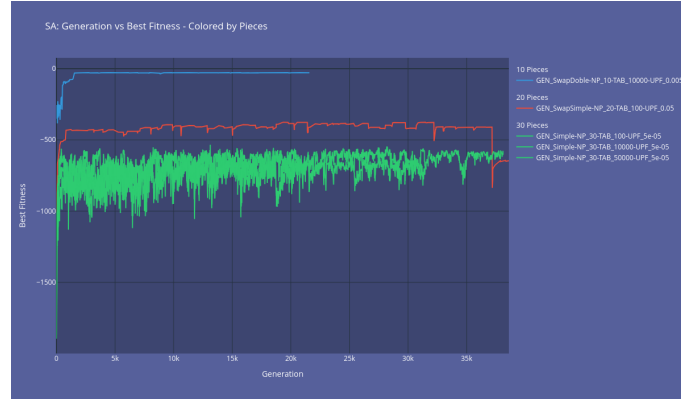


Figura B.5: Evolución del fitness del algoritmo de enfriamiento simulado a lo largo de las iteraciones.

B.2.2. Efecto del tipo de individuo (movimientos)

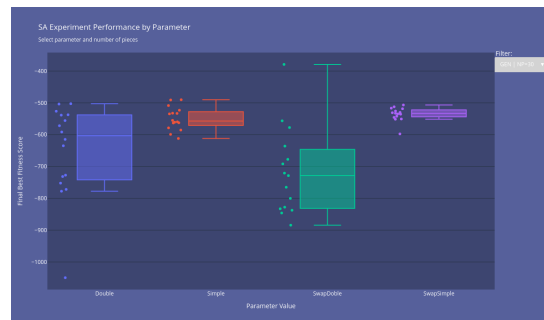
Las siguientes gráficas muestran cómo el tipo de individuo (Simple, Double, SwapSimple, SwapDouble) afecta al fitness final obtenido para diferentes cantidades de piezas.



(a) 10 piezas (Fácil)



(b) 20 piezas (Medio)



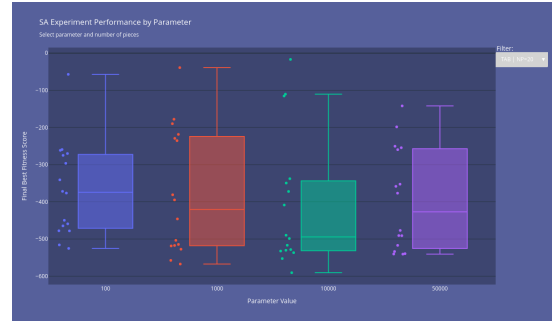
(c) 30 piezas (Difícil)

Figura B.6: Efecto del tipo de individuo en el enfriamiento simulado para diferente N° de piezas.

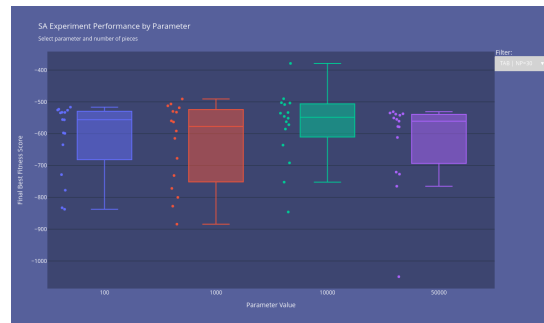
B.2.3. Efecto del tamaño de la lista tabú



(a) 10 piezas (Fácil)



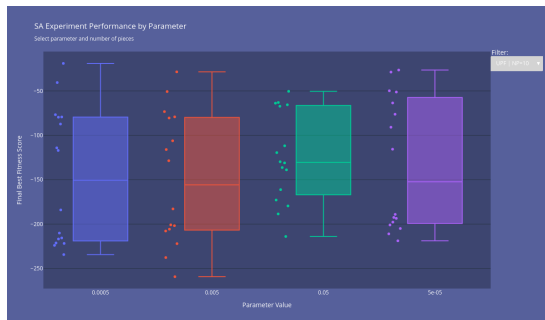
(b) 20 piezas (Medio)



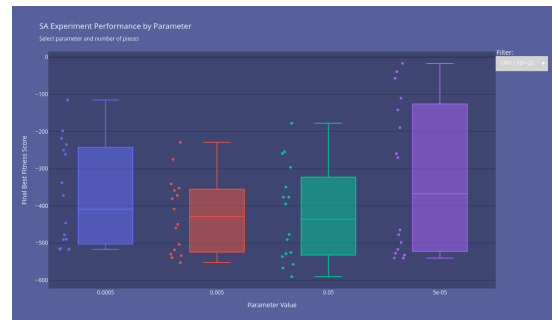
(c) 30 piezas (Difícil)

Figura B.7: Efecto del tamaño lista tabú en el enfriamiento simulado para diferente N° de piezas.

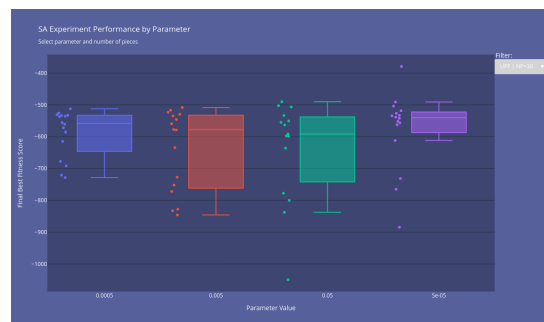
B.2.4. Efecto del factor de enfriamiento



(a) 10 piezas (Fácil)



(b) 20 piezas (Medio)



(c) 30 piezas (Difícil)

Figura B.8: Efecto factor de enfriamiento en el enfriamiento simulado para diferente N° de piezas.