Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Algoritmo de Estimación de Distribuciones para Resolver el Cubo de Rubik 3x3x3 Estándar

Autor: Alberto Montero Santos
Tutor(a): Juan Antonio Fernandez del Pozo

Madrid, Enero 2026

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática

Título: Algoritmo de Estimación de Distribuciones para Resolver el Cubo de Rubik 3x3x3 Estándar

Enero 2026

Autor: Alberto Montero Santos
Tutor: Juan Antonio Fernandez del Pozo
Departamento de Inteligencia Artificial
Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid

Resumen

Este Trabajo Fin de Grado presenta el estudio y desarrollo de un sistema de inteligencia artificial orientado a la resolución del cubo de Rubik 3x3x3 mediante técnicas de optimización combinatoria. El proyecto se centra en el diseño e implementación de un solver basado en un modelo evolutivo mixto, cuyo núcleo es un Algoritmo de Estimación de Distribuciones (EDA), complementado con distintos mecanismos heurísticos y un simulador propio que permite evaluar de forma controlada el comportamiento del sistema.

El documento comienza con una introducción al problema, en la que se define el cubo de Rubik como un reto combinatorio de gran complejidad y se presentan los EDAs como una alternativa a otros enfoques clásicos de búsqueda y optimización. A partir de esta base, se establecen las hipótesis de trabajo, el ámbito y alcance del proyecto, así como los objetivos perseguidos, centrados en analizar la viabilidad de este tipo de algoritmos para la resolución automática del cubo y en estudiar el impacto de distintas mejoras prácticas sobre su rendimiento.

A continuación, se desarrolla un marco teórico que recoge los fundamentos necesarios para el proyecto y un análisis del estado del arte. En este apartado se abordan tanto los aspectos teóricos del cubo de Rubik —incluyendo su estructura, notación, simetrías y estrategias de resolución— como los principales conceptos de la optimización combinatoria y los algoritmos evolutivos, con especial énfasis en los EDAs, sus variantes basadas en modelos gráficos y las estrategias híbridas empleadas en la literatura.

El núcleo del trabajo lo constituye el diseño e implementación del modelo del cubo y del solver. Se presenta una representación formal del cubo que permite simular sus estados y movimientos, así como mecanismos de simplificación, canonización, eliminación de ciclos y explotación de simetrías. Sobre esta base se construye el solver evolutivo, definiendo la representación de los individuos, las funciones de fitness basadas en medidas de similitud y otros criterios alternativos, y el proceso de evolución. El algoritmo se enriquece progresivamente mediante técnicas como elitismo, mutación, búsqueda local y poda, dando lugar a un modelo evolutivo mixto que combina exploración probabilística y explotación heurística.

Finalmente, se presentan y analizan los resultados obtenidos a partir de distintas pruebas experimentales, evaluando el grado de cumplimiento de los objetivos planteados y comparando el comportamiento de las distintas configuraciones del

algoritmo. El trabajo concluye con una discusión sobre las principales conclusiones extraídas, las limitaciones del enfoque propuesto y las posibles líneas de mejora futuras, incluyendo la exploración de modelos probabilísticos alternativos y el uso más profundo de herramientas matemáticas, como la teoría de grupos.

En conjunto, este TFG aporta un estudio detallado y experimental sobre la aplicación de algoritmos de estimación de distribuciones y estrategias evolutivas híbridas a un problema combinatorio complejo, ofreciendo una base sólida para futuras extensiones en el ámbito de la inteligencia artificial y la optimización.

Palabras clave: Cubo de Rubik, Inteligencia artificial, Algoritmos evolutivos, Algoritmos de estimación de distribución, Optimización combinatoria, Simulación, Solver

Abstract

This Bachelor's Thesis presents the study and development of an artificial intelligence system aimed at solving the 3x3x3 Rubik's Cube using combinatorial optimization techniques. The project focuses on the design and implementation of a solver based on a mixed evolutionary model, whose core is an Estimation of Distribution Algorithm (EDA), complemented by several heuristic mechanisms and a custom simulator that allows the system's behavior to be evaluated in a controlled manner.

The document begins with an introduction to the problem, defining the Rubik's Cube as a highly complex combinatorial challenge and presenting EDAs as an alternative to classical search and optimization approaches. Based on this foundation, the working hypotheses, the scope of the project, and the main objectives are established, with a particular emphasis on analyzing the feasibility of these algorithms for the automatic resolution of the cube and on studying the impact of different practical enhancements on their performance.

Subsequently, a theoretical framework and a review of the state of the art are developed. This section addresses both the theoretical aspects of the Rubik's Cube—including its structure, notation, symmetries, and resolution strategies—and the main concepts of combinatorial optimization and evolutionary algorithms, with special emphasis on EDAs, their variants based on graphical models, and hybrid strategies reported in the literature.

The core of the thesis consists of the design and implementation of the cube model and the solver. A formal representation of the cube is presented, enabling the simulation of its states and movements, along with mechanisms for simplification, canonization, cycle elimination, and exploitation of symmetries. On this basis, the evolutionary solver is constructed by defining the representation of individuals, fitness functions based on similarity measures and alternative criteria, and the evolutionary process itself. The algorithm is progressively enhanced through techniques such as elitism, mutation, local search, and pruning, resulting in a mixed evolutionary model that combines probabilistic exploration with heuristic exploitation.

Finally, the results obtained from different experimental tests are presented and analyzed, evaluating the degree to which the proposed objectives are achieved and comparing the behavior of the different algorithm configurations. The thesis concludes with a discussion of the main findings, the limitations of the proposed

approach, and possible future lines of improvement, including the exploration of alternative probabilistic models and a deeper use of mathematical tools such as group theory.

Overall, this project provides a detailed experimental study on the application of estimation of distribution algorithms and hybrid evolutionary strategies to a complex combinatorial problem, offering a solid foundation for future extensions in the field of artificial intelligence and optimization.

Keywords: Rubik's Cube, Artificial Intelligence, Evolutionary Algorithms, Estimation of Distribution Algorithms (EDAs), Combinatorial Optimization, Simulation, Solver

Tabla de contenidos

1. Introducción **1**
1.1. Definición 1
1.1.1. El cubo 1
1.1.2. Los Algoritmos de Estimación de Distribución (EDAs) 2
1.1.3. Hipótesis 3
1.2. Ámbito y Alcance 4
1.3. Motivación y objetivos 4
1.4. Descripción del documento por capítulos 5

2. Fundamentos y Estado del Arte **7**
2.1. El cubo de Rubik 7
2.1.1. Introducción: El cubo de Rubik como problema combinatorio 7
2.1.2. Definiciones 8
2.1.3. Estructura física y teórica 8
2.1.4. Notación de movimientos 9
2.1.5. Ciclos 10
2.1.6. Simetrías 10
2.1.7. Resolución por Humanos 11
2.1.8. Resolución Automática 12
2.1.9. El “Número de dios” (God’s Number) 14
2.2. Optimización Combinatoria 15
2.2.1. Algoritmos Evolutivos 15
2.2.2. Otras metaheurísticas 16
2.2.3. Algoritmos de Estimación de Distribución (EDAs) 17
2.2.4. EDAs con modelos gráficos 18
2.2.5. Estrategias híbridas y mejoras prácticas 19
2.2.6. Implementaciones 20

3. Modelo e Implementación del Cubo **23**
3.1. Definiciones 23
3.2. Estructura 23
3.2.1. Coordenadas 24
3.2.2. Las Piezas 25
3.2.3. Representación Gráfica 25
3.3. Secuencias de Movimientos 26
3.3.1. Simplificación y Canonización 26

3.3.2. Eliminación de Ciclos 27
3.3.3. Simetrías 28

4. Modelo e Implementación del Solver **29**
4.1. Concepto Base 29
4.2. Fitness 29
4.2.1. Similitud como Fitness 30
4.2.2. Medidas alternativas de fitness 32
4.3. Procesamiento de individuos 33
4.3.1. Canonización 33
4.3.2. Sub-cadenas 33
4.3.3. Uso de EDAspy 34
4.4. Evolución del algoritmo 35
4.4.1. EDA puro 35
4.4.2. Elitismo 36
4.4.3. Mutación 37
4.4.4. Búsqueda local 39
4.4.5. Poda 42

5. Resultados y Conclusiones **45**
5.1. Resultados 45
5.1.1. Notas sobre las pruebas 45
5.2. Evaluación de los objetivos del trabajo 48
5.3. Conclusiones 48
5.3.1. Fitness 48
5.3.2. Generalización vs Optimalidad 49
5.3.3. Elementos añadidos 49
5.4. Lineas de mejora 49
5.4.1. Fitness 49
5.4.2. Redes Bayesianas vs Cadenas de Markov 50
5.4.3. Uso de Teoría de Grupos 50
5.5. Análisis de Impacto y ODS 50
5.5.1. Análisis de impacto 51
5.5.2. Alineamiento con los Objetivos de Desarrollo Sostenible . . . 51

Bibliografía **53**

Anexos **57**

A. Primer anexo: Repositorio del proyecto **57**

Índice de figuras

1.1. El cubo de Rubik clásico 2
1.2. Diferentes tipos de cubo de rubik 2
1.3. Ejemplo de una Red Bayesiana simple. Fuente: Elaboración propia. 3

2.1. Nomenclatura de las caras del cubo [1] 9
2.2. El resultado de aplicar "U" en el cubo resuelto [2] 9
2.3. Esquema del funcionamiento de un algoritmo evolutivo. Fuente: elaboración propia 16
2.4. Esquema del funcionamiento de un algoritmo EDA. Fuente: elaboración propia 17

3.1. Desarrollo en plano del cubo en estado resuelto 25
3.2. Desarrollo en plano del cubo tras aplicar "U" 26

4.1. Gráfico mostrando las diferentes medidas 31
4.2. Diagrama del algoritmo como EDA puro 36
4.3. Diagrama del algoritmo con Elitismo 36
4.4. Diagrama del algoritmo con mutación 38
4.5. Diagrama del algoritmo con mutación y mutación en élites 39
4.6. Diagrama que ilustra el proceso de búsqueda local 40
4.7. Diagrama del algoritmo con búsqueda local 41
4.8. Diagrama del algoritmo con poda 43

5.1. Gráfica que muestra los resultados de las pruebas: número de generaciones necesaria para resolver cada caso probado por cada longitud de scramble 46
5.2. Gráfica que muestra los resultados de las pruebas: diferencia en longitud entre la solución y el scramble 47

Índice de cuadros

2.1. Configuraciones posibles del cubo de rubik para cada distancia desde el estado resuelto [3] 14

4.1. Resultados de las diferentes medidas de similitud propuestas . . . 30

Capítulo 1

Introducción

En este capitulo se introduce el proyecto, explicando a nivel introductorio los aspectos principales, y definiendo el ámbito, el alcance, la motivación, y los objetivos del mismo.

1.1. Definición

El objetivo es crear un prototipo de un sistema con Inteligencia Artificial para resolver el cubo de Rubik 3x3x3, utilizando un algoritmo evolutivo de tipo EDA (Estimation of Distribution Algorithms), a fin de exhibir su utilidad para problemas reales. Se implementarán tanto el sistema solver, como el sistema simulador del cubo.

1.1.1. El cubo

El cubo de Rubik [4] es un rompecabezas complejo con un amplio espacio combinatorio y numerosas restricciones. Fue inventado en 1974 y ha sido relevante durante muchos años en la cultura popular; y ha llegado a ocupar pequeños rincones de la sociedad con una gran variedad de diseños y torneos, capturando a miles de aficionados que coleccionan diferentes cubos y compiten en la rápida resolución de los mismos. La figura 1.1 muestra una imagen del cubo de Rubik 3x3x3 clásico.

Existen numerosos tipos de cubos de rubik [4], con distintos conceptos: desde simples alteraciones, cómo cambiar el número de filas y/o columnas (2x2x2, 4x4x4, 3x3x2, etc); pasando por la adición de restricciones; modificando la forma de las piezas en vez de usar colores (mirror, ghost, etc); utilizando diferentes poliedros como bases en lugar de cubos (Pyraminx, Petaminx, etc); e incluso sistemas completamente distintos (Square-1, Skewb, etc) que solo guardan en común el espíritu y estética del original. Algunos de estos se muestran en la figura 1.2. En este proyecto, nos centraremos exclusivamente en el cubo original, el 3x3x3, a fin de acotar y simplificar el problema.

Los humanos suelen resolver cubos [4] de forma metódica y sistemática, jerar-

Figura 1.1: El cubo de Rubik clásico

Figura 1.2: Diferentes tipos de cubo de rubik

quizando el problema en varios sub-niveles más sencillos que el problema completo. Para ello, mezclan el uso de "soltura" (moviendo las piezas con intuición y lógica entrenada con práctica) para llegar a estados en los que la combinatoria es reducida, donde aplican algoritmos que han aprendido previamente para pasar a la siguiente fase, hasta que el cubo está resuelto. Esto hace que sea mucho más sencillo para un humano realizarlo, pero se pierde mucha eficiencia (los mejores humanos suelen utilizar entre 100 y 60 movimientos). Este sistema es muy diferente al que se pretende alcanzar en este proyecto y que suelen utilizar las IAs: en vez de resoluciones sistemáticas y metódicas con divisiones del problema, utilizaremos la alta capacidad computacional para conseguir soluciones mucho más cortas, pero posiblemente poco intuitivas para las personas.

1.1.2. Los Algoritmos de Estimación de Distribución (EDAs)

Los algoritmos de estimación de distribución (EDAs)[5] son algoritmos de tipo *evolutivo*, lo cual quiere decir que parten de una "población" inicial y, a partir de ella, aprenden de los "individuos" considerados *mejores* y crean una nueva generación a partir de estos, que en principio sería mejor que la anterior. Tras esto, repiten el proceso, mejorando progresivamente. A diferencia de otros

algoritmos evolutivos más conocidos, como los *algoritmos genéticos*, que crean nuevas generaciones utilizando ciertas operaciones, como el cruce o la mutación; los EDA generan un **modelo probabilístico** a partir del cual construyen la nueva generación. Esto les otorga ciertas ventajas con respecto a los algoritmos genéticos.

Existen diferentes enfoques para construir el modelo probabilístico que utiliza el EDA, pero uno de los más relevantes es el uso de *modelos gráficos*, en particular de redes bayesianas[6]. Estos son grafos en los que los nodos representan variables y resultados, y están conectados entre sí mediante arcos dirigidos que representan relaciones probabilísticas condicionales. En su totalidad, la red representa la probabilidad conjunta de las variables conforme a las dependencias que recoge el grafo. Estas características proporcionan una gran flexibilidad y adaptabilidad al modelo al añadir o quitar nodos, añadir o quitar aristas, o modificar las probabilidades entre estos. Además, permiten modificaciones locales, y presentan una mayor interpretabilidad que otras soluciones. La figura 1.3 muestra un ejemplo simple de una red bayesiana a nivel conceptual.

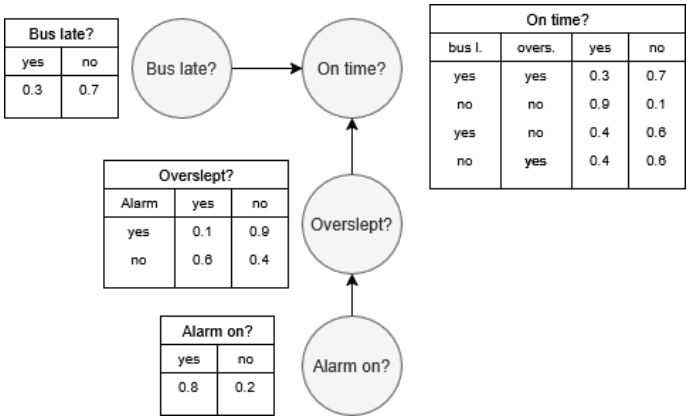

Figura 1.3: Ejemplo de una Red Bayesiana simple. Fuente: Elaboración propia.

1.1.3. Hipótesis

En este proyecto, se pretende diseñar un EDA basado en redes bayesianas para resolver el cubo de rubik estándar 3x3x3. Para ello, se utilizarán grandes muestras de secuencias de movimientos aleatorios con una longitud acotada que tomen como estado inicial el cubo hecho y lo deshagan. El sistema aprenderá las secuencias de movimientos que *deshacen* el cubo y, a partir de ello, utilizando las secuencias inversas, debería ser capaz de resolverlo, usando como métrica objetivo la comparación de los estados deshechos con el estado inicial que se desea resolver. Se aplicarán también técnicas híbridas de optimización local y conocimiento del dominio para optimizar el modelo. Así mismo, es posible que los resultados del trabajo permitan extraer conocimiento sobre el funcionamiento de solvers basados en EDAs, y de las estrategias aplicables en su diseño.

1.2. Ámbito y Alcance

El alcance de este trabajo se limita a la aplicación de EDAs básicos utilizando redes bayesianas, aplicadas al cubo de Rubik 3x3x3 clásico. No se discutirán otros modelos del cubo de Rubik ni otros tipos de algoritmos evolutivos más allá de un nivel introductorio en la sección de Estado del Arte.

Aunque se pretende utilizar un modelo de EDA "básico", es decir, que siga el modelo estándar de un EDA; sí se pretende usar conocimiento del dominio y optimizaciones locales para mejorar el solver, aunque tampoco se descarta el uso de propiedades de otros paradigmas como puedan ser los algoritmos genéticos. Existen ciertas propiedades del cubo que permitirían ganar mucha eficiencia con relativamente poco esfuerzo. Entre estas, cabe destacar la eliminación de *ciclos* (aquellos movimientos que, desde un estado A, acaban volviendo a ese mismo estado A) y varios tipos de equivalencias, como re-coloraciones o simetrías (todo lo cual se discutirá en mayor profundidad mas adelante), así como pequeñas optimizaciones locales.

Es importante mencionar que no se pretende implementar un sistema óptimo (ni en cuanto al número de movimientos, al tiempo de ejecución, ni a otras medidas), sino un sistema *funcional* que utilice EDAs. Existen varios algoritmos específicos para los cubos de rubik que son óptimos en cuanto al número de movimientos y que son también computacionalmente baratos (especialmente en comparación con un EDA) que se discutirán más adelante.

1.3. Motivación y objetivos

En pleno auge de las tecnologías de Inteligencia Artificial, nuevos modelos, algoritmos y paradigmas surgen constantemente. Entre ellos, los algoritmos evolutivos se han establecido como una herramienta popular y potente, simples a nivel conceptual, y fáciles de implementar, pero capaces de resolver problemas y optimizaciones complejas, a menudo encontrando soluciones casi óptimas, con bajos costes computacionales (en comparación con sistemas exactos para cada dominio, que a menudo son muy caros).

Este trabajo de fin de grado busca mostrar el potencial de los EDAs como parte de la familia de algoritmos evolutivos, y las ventajas que tiene el uso de modelos probabilísticos frente a otros enfoques. Para ello, se ha escogido un problema con cierto nivel de complejidad con el que muchos están familiarizados: el cubo de Rubik, del cual también se discutirán características relevantes para su resolución.

Los objetivos del proyecto son:

1 Simulador del Cubo

- Diseño con bases matemáticas y formales
- Representación gráfica
- Movimientos y cadenas de movimientos

- Medida de distancia entre estados
- Ciclos, equivalencias, simetrías, y otras utilidades de conocimiento del dominio

2 Solver

- Diseño del EDA
- Implementación del EDA
- Modelo gráfico con redes bayesianas
- Optimizaciones locales

3 Conclusiones

- Pruebas
- Comparaciones

1.4. Descripción del documento por capítulos

Este documento se estructura en varios capítulos que abordan de manera detallada los distintos aspectos del proyecto. La organización es la siguiente:

Capítulo 2: Fundamentos y Estado del Arte - En este capítulo se abordarán los principios teóricos y los estudios previos relacionados con los EDAs y la resolución de los Cubos de Rubik.

Capítulo 3: Modelo e Implementación del Cubo - Este capítulo indagará en el problema que representa el cubo de Rubik. Se revisará cómo funciona y se comporta, la nomenclatura estándar, así como ciertas características relevantes a su resolución automática. Se discutirán también el modelo utilizado para la implementación, la medida de ”fitness”, y otros aspectos relevantes.

Capítulo 4: Modelo e Implementación del Solver - En este capítulo se detallará lo relacionado con la parte del solver del proyecto. Se discutirá el modelo e implementación del EDA, así como las optimizaciones aplicadas con conocimiento del dominio y búsqueda local, y demás aspectos del algoritmo.

Capítulo 5: Resultados y Conclusiones - Este capítulo discutirá las pruebas realizadas, los resultados finales obtenidos y las comparaciones con otros datos relevantes. Se analizarán los resultados y se estudiarán métricas de rendimiento a diversos niveles. Finalmente, se llevará una evaluación general del cumplimiento de los objetivos del proyecto, se presentarán posibles mejoras y propuestas, y se reflexionará sobre el desarrollo y la experiencia del proyecto. Además, se analiza el alineamiento del proyecto con los Objetivos de Desarrollo Sostenible (ODS).

Capítulo 2

Fundamentos y Estado del Arte

En este capítulo se discutirán y explorarán los fundamentos teóricos del proyecto, así como trabajos y tecnologías previas relevantes a este.

2.1. El cubo de Rubik

En está sección se discutirán los fundamentos teóricos del problema del cubo de rubik. Se explicará su funcionamiento y estructura (incluidos aspectos relevantes a este proyecto como ciclos y simetrías), se discutirán definiciones, nomenclatura, y notación de movimientos. Además, se revisarán métodos de resolución manuales y automáticos, y se discutirán otros aspectos relevantes del problema.

2.1.1. Introducción: El cubo de Rubik como problema combinatorio

El cubo de Rubik 3×3×3 [7] es un clásico rompecabezas mecánico que, desde su invención por Ernő Rubik en 1974, ha atraído tanto a aficionados como a investigadores debido a su naturaleza simple de entender, pero extremadamente compleja de resolver en su totalidad. En su forma estándar, consta de 6 caras, cada una inicialmente de un color uniforme. Véase la figura 1.1: El cubo de Rubik clásico. Una configuración es alcanzable mediante una secuencia de giros de caras. Se estima que el número de estados posibles del cubo es aproximadamente $4{,}3 * 10^{19}$.

Además de su interés lúdico, el cubo de Rubik constituye un paradigma de *problema combinatorio* con una estructura matemática bien definida y un espacio de búsqueda finito pero inmenso. Por ello, ha sido empleado como caso de estudio en múltiples disciplinas: desde la teoría de grupos y la inteligencia artificial, hasta la optimización heurística y el aprendizaje automático. Esta combinación de simplicidad estructural y complejidad combinatoria lo convierte en un entorno ideal para probar la capacidad de los algoritmos evolutivos y probabilísticos de modelar dependencias y patrones de búsqueda eficientes.

Desde la perspectiva de la optimización, el cubo representa un espacio de estados gigantesco, con un movimiento de giro que actúa como un operador de transición entre estados. El objetivo es encontrar una secuencia de giros que transforme una configuración inicial no resuelta en la configuración resuelta (cada cara de un mismo color). Esto lo convierte en un problema de búsqueda en un grafo de gran tamaño, con un coste de exploración enorme si se aplica fuerza bruta.

El carácter combinatorio, estructurado y altamente simétrico del cubo le proporciona ricas *propiedades de grupo* y lo convierte en un excelente banco de pruebas para algoritmos de optimización evolutiva, heurísticos y de distribución probabilística. Por estos motivos, resulta un dominio ideal para explorar algoritmos como los EDA y otros métodos evolutivos, así como también para contrastarlos con algoritmos de resolución óptima o casi óptima desarrollados específicamente para este rompecabezas.

2.1.2. Definiciones

Se considera que el cubo está *resuelto* cuando todas las caras tienen un solo color uniforme; en cualquier otro caso, se considera que el cubo está deshecho o *scrambled*. De ello, a una secuencia de movimientos que toma como estado inicial el cubo resuelto y lo deshace, se le llama *scramble*. Se le denomina *cuber* a las personas aficionadas a los cubos de rubik; y *speedcuber* a aquellos que se especializan en la resolución en el menor tiempo posible de los mismos, a menudo compitiendo en ello. Se les llama *algoritmos* a las secuencias de movimientos o maniobras que realizan permutaciones específicas en el cubo.

2.1.3. Estructura física y teórica

El cubo 3×3×3 está formado por 26 ”cubies” (es decir, piezas): 8 esquinas (cada una con tres colores), 12 aristas (con dos colores) y 6 centros (un color) que permanecen fijos entre sí (aunque las piezas pueden rotar alrededor de ellos). Esta estructura define el diseño de movimientos permitidos y las restricciones de permutación: no todas las permutaciones arbitrarias de las piezas son alcanzables debido a las restricciones de orientación, paridad, y movimiento del mecanismo.

Dado que los centros son fijos entre sí, se puede llamar a cada cara por el color de su centro. Con ello, por lo general, la estructura de colores es la siguiente: cada cara tiene una cara opuesta a pares (blanco y amarillo; rojo y naranja; verde y azul) y, con la cara amarilla encima, la cara verde se encuentra en sentido antihorario respecto a la roja.

Desde el punto de vista formal, el conjunto de todas las configuraciones alcanzables del cubo, junto con el operador de rotación, constituye un *grupo de permutaciones*, denominado Rubik's Group [1]. Cada estado puede representarse como una permutación de las piezas y sus orientaciones. Así, el espacio de búsqueda puede analizarse en términos de la teoría de grupos y álgebra combinatoria, proporcionando una base sólida para los enfoques algorítmicos posteriores. La representación formal que se defenderá más adelante está basada en el álgebra.

2.1.4. Notación de movimientos

La notación estándar [8] para movimientos utiliza letras representativas de la cara para indicar un giro de 90° en el sentido horario. Estas son U, D, F, B, L, R de las caras Up, Down, Front, Back, Left, Right respectivamente, como muestra la figura 2.1. Las letras seguidas de un apóstrofo (U', leído como "U prima" o "U inversa") indican giro antihorario, mientras que las letras seguidas del número 2 (R2) denotan un giro de 180° (equivalente a dos giros de 90°). Esta notación es universal en la comunidad de cubers y en la investigación sobre algoritmos de resolución. La figura 2.2 muestra el resultado de aplicar "U" en el cubo resuelto.

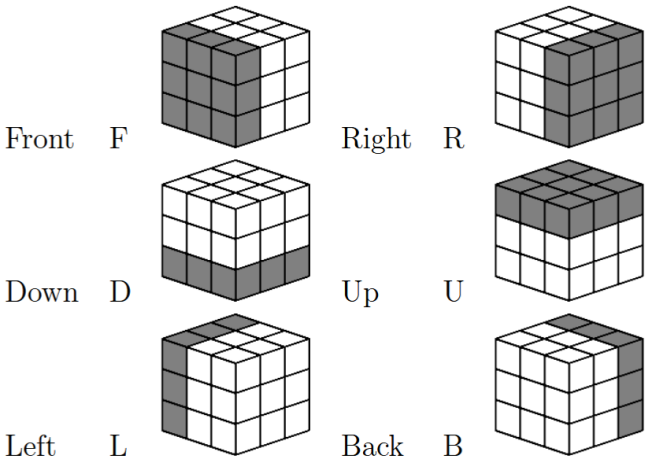

Figura 2.1: Nomenclatura de las caras del cubo [1]

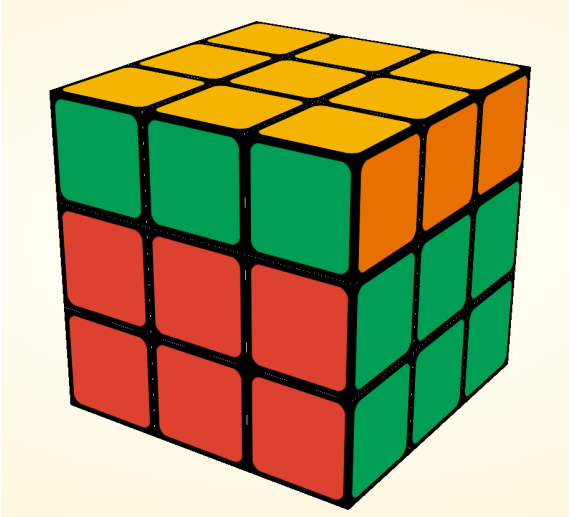

Figura 2.2: El resultado de aplicar "U" en el cubo resuelto [2]

Adicionalmente, existen notaciones extra [8] para movimientos que son combinaciones de movimientos, con el fin de hacer más escuetas las secuencias de movimientos o más fáciles de ejecutar para los humanos. Se utilizan X, Y, Z

para hacer un giro global del cubo en el eje dado; M, E, y S representan giros de las tiras interiores en los ejes x, y, z, respectivamente; y se utilizan las letras de las caras en minúscula para representar giros de la cara en cuestión junto con la tira interior adyacente (por ejemplo, "r" es equivalente a R+M). Ya que estas nomenclaturas se utilizan para facilitar el movimiento de humanos, no son relevantes para el contexto de este proyecto. Consultar [8] para visualizar los movimientos en un simulador online.

En esta memoria se notarán los movimientos entre corchetes ([R]) y las secuencias de movimientos entre corchetes separados por espacios ([R U R' U']).

2.1.5. Ciclos

Decimos que una secuencia (o sub-secuencia) de movimientos es un çicloçuando el estado inicial del cubo (antes de aplicar la secuencia) es igual al estado final (después de aplicar la secuencia). Podemos distinguir dos tipos de ciclos.

- Ciclos de rotación: Se trata de ciclos que ocurren cuando se rota (cuartos de rotación) una sola cara 0 veces en módulo 4. Por ejemplo, la secuencia de movimientos [R R'] es el ejemplo más simple de esto: El movimiento [R] aplica un cuarto de rotación sobre la cara R, mientras que [R'] aplica un cuarto de rotación en el sentido contrario; dejando un total de rotación de la cara de 0. Otro ejemplo sencillo es [R R R R], que rota la cara 4 veces, dejando 0 en módulo 4. Por supuesto, puede haber ciclos dentro de ciclos; por ejemplo, [R U U'R'].
- Ciclos periódicos: Las propiedades físicas del cubo conducen a una interesante particularidad: Cualquier secuencia de movimientos, repetida suficientes veces, constituye un ciclo [1]. Esto es trivial de ver empíricamente, pero complicado de demostrar formalmente. Además, la longitud de la secuencia no está relacionada con su periodicidad. Por ejemplo, la secuencia [R U R'F' R U R'U' R'F R2 U' R'U'] (conocida como J-Perm, o permutación tipo J) tiene una periodicidad de 2; mientras que una secuencia mucho más sencilla como [R2 U2] tiene una periodicidad de 6; en contraste con [R U], muy similar a esta última, pero con una periodicidad de 35 (calculado con el simulador descrito en 3.3.2).

Ya que no podemos garantizar que las secuencias generadas aleatoriamente o aquellas generadas por los modelos probabilísticos no contengan ciclos, la identificación (y subsecuente eliminación) de estos es importante para este proyecto, ya que son secuencias de movimientos que no cambian la situación del cubo, y que reducen la eficiencia bajo la métrica de número de movimientos. Además, los modelos podrían llevar a reforzar el aprendizaje de ciclos, lo cual queremos evitar.

2.1.6. Simetrías

Debido a la estructura del cubo, existen diversos tipos de simetría [9]. Claramente, existen simetrías axiales (de espejo) y rotacionales, pero también hay

otros tipos particulares del cubo de rubik, entre los que destacan las simetrías de re-coloración.

- Las simetrías de re-coloración: Podemos llamar a dos estados distintos del cubo simétricos por re-coloración si, al cambiar los colores de las pegatinas sistemáticamente (y manteniendo las restricciones de color de la estructura base) de uno de ellos, existe una configuración en la que son iguales. El ejemplo más claro de esto es realizar cualquier giro (como [R]), y el estado resultante será simétrico por re-coloración a todos los demás estados resultantes de mover una cara. En esencia, lo importante en el estado del cubo no son los colores en sí, sino las relaciones entre ellos. Existen 24 (4 por cara, 6 caras) posibles coloraciones válidas para el cubo, lo cual significa que un sistema agnóstico al color tendría un espacio de búsqueda hasta 24 veces menor. Además, las simetrías de re-coloración son computacionalmente baratas de comprobar, por lo que su uso en el sistema en este proyecto es de gran interés.

- Las simetrías axiales (o de espejo) también son interesantes. Típicamente, se define la simetría axial de la siguiente manera: dos estados son simétricos axialmente si, al proyectar uno de ellos a través de un eje de simetría, el estado resultante es igual al otro. Sin embargo, en el cubo, realizar una proyección siempre lleva a un estado inválido; en su lugar, consideramos que dos estados son simétricos axialmente cuando existe una secuencia de movimientos que, al aplicarla tal cual a uno de ellos y su reflexión al otro, provoca que los estados resultantes sean iguales. Por ejemplo, los estados resultantes de [R] y [L'] son simétricos axialmente. Así pues, comprobar la simetría axial requiere calcular la secuencia reflejada y simular la secuencia. Usar simetrías axiales podría reducir el espacio de búsqueda por un factor de hasta 2.

2.1.7. Resolución por Humanos

Cuando los humanos resuelven el cubo, *dividen el problema* en varias fases, reduciendo los espacios combinatorios en cada una, y combinando el uso de "soltura" para permutaciones sencillas y la aplicación de algoritmos memorizados con anterioridad para resolver permutaciones complejas. La aplicación de algoritmos requiere la memorización de los estados en los que se aplican y la memorización de la propia secuencia de movimientos (a menudo, existen varias formas de cada algoritmo para resolver diferentes estados simétricos u otras variaciones).

Los métodos más básicos para principiantes buscan memorizar pocos algoritmos y, en su lugar, dividen el problema en más sub-partes para reducir considerablemente la combinatoria (y, por tanto, los algoritmos a memorizar) en cada sub-parte; pero esto mismo hace que tomen caminos con más movimientos en total. Es por ello que, para resolver el cubo en el menor número de movimientos posible, se requiere memorizar cuántos más algoritmos sea posible, y por tanto los speedcubers memorizan cientos de algoritmos.

Él método más común de resolución, conocido como método de Fridrich [10], sigue los siguientes pasos:

1 Cruz inicial: Seleccionar una cara (típicamente, la blanca) y completar la ”cruz” de esa cara, es decir, colocar las aristas blancas en sus posiciones finales (importante que el color en el lado no blanco de la arista sea el del centro de esa cara)

2 First-two-lines (F2L): Consiste en colocar las esquinas de la capa blanca en sus posiciones correspondientes junto a las aristas correspondientes. Idealmente, este paso se realiza en conjunto por eficiencia, pero los principiantes lo pueden hacer por separado.

3 Orient-Last-Line (OLL): Consiste en orientar las piezas de la cara amarilla sin colocarlas en las posiciones finales, simplemente asegurándose de que la cara sea homogénea. Este paso también se puede dividir en dos para reducir el número de casos en los que hay que memorizar algortimos: orientar la cruz amarilla, y orientar las esquinas amarillas.

4 Permute-Last-Line (PLL): Consiste en permutar las piezas de la cara amarilla de lugar sin modificar su orientación ni el resto del cubo.

- Además, existen algoritmos CLL (Complete-Last-Line), que realizan OLL y PLL en un solo algoritmo, pero representa un espacio con muchos más casos, así como algoritmos que completan el OLL al insertar el último par en F2L, y otros similares que buscan combinar pasos al precio de tener que memorizar muchos más algoritmos.

En resumen, la resolución humana puede verse como un proceso de optimización jerárquica guiada por heurísticas cognitivas. Los cubers utilizan estrategias que reducen el espacio de búsqueda mediante la descomposición en subproblemas y la reutilización de patrones aprendidos (algoritmos). Desde una perspectiva computacional, estos métodos pueden interpretarse como una búsqueda informada apoyada en conocimiento previo, en contraste con la búsqueda sin experiencia que realizaría un algoritmo guiado por funciones objetivo (fitness). Esta analogía ha inspirado el diseño de sistemas de resolución automática que imitan dicha segmentación por fases.

2.1.8. Resolución Automática

1. **Algoritmos específicos para el problema (exactos/óptimos)**: En el ámbito de la resolución automática, los enfoques más eficientes para el cubo son aquellos que explotan la *teoría de grupos*, las *bases de datos de patrones* (pattern databases), la *búsqueda con poda* (IDA*) y los *algoritmos de dos fases*. Para la resolución automática, podemos considerar dos métricas de eficiencia: número de movimientos para resolver, y eficiencia computacional.

   Los métodos exactos están diseñados específicamente al problema del cubo, y tratan de encontrar soluciones *optimas* o casi optimas (es decir, la

secuencia mínima de movimientos posible), manteniendo gran *eficiencia computacional*.

El uso de la teoría de grupos permite identificar sub-espacios del cubo que son invariables bajo ciertos conjuntos de movimientos, lo cual reduce drásticamente la complejidad de búsqueda. Dividen el espacio total en sub-espacios más manejables, igual que los humanos, pero combinando esto con búsquedas locales. En este contexto, destacan:

- El algoritmo de dos fases de Kociemba [9]: Uno de los más relevantes dada su eficiencia, se trata de un algoritmo que es casi óptimo en cuanto al número de movimientos, y que tiene un muy bajo coste computacional. Usa el paradigma de dos fases y explota la teoría de grupos, dividiendo el problema en dos sub-fases: en la fase 1, lleva la configuración a un sub-grupo específico mediante un subconjunto de movimientos que preservan ciertas propiedades, usando el algoritmo IDA* para llegar al subgrupo y; en la fase 2, completa la solución desde ese sub-grupo hacia el estado resuelto de nuevo usando IDA* [11].
- El algoritmo de Thistlethwaite [11] (1981), de manera similar al algoritmo de Kociemba, explota subgrupos para reducir los espacios de búsqueda, pero divide el problema en cuatro fases en lugar de dos.

2. **Métodos heurísticos, metaheurísticos, y evolutivos**: Más allá de los métodos exactos, se han explorado *enfoques heurísticos y metaheurísticos* (por ejemplo, algoritmos genéticos) aplicados al cubo [12] [11]. La resolución mediante metaheurísticas introduce un enfoque alternativo basado en exploración estocástica. Se han probado algoritmos genéticos (GA), simulated annealing, búsqueda tabú, y estrategias evolutivas, con el objetivo de encontrar soluciones aceptables sin necesidad de conocimiento previo ni tablas pre-computadas. En general, las metaheurísticas tratan de equilibrar la exploración global del espacio de estados y la explotación local de soluciones prometedoras, pero su rendimiento depende en gran medida de cómo se codifican los movimientos y cómo se define la función de fitness (por ejemplo, cuántas piezas están correctamente posicionadas u orientadas).

   Se han publicado trabajos que utilizan algoritmos genéticos o estrategias evolutivas para resolver el cubo minimizando el número de movimientos o el tiempo de cómputo [12]. Otro estudio presenta dos enfoques evolutivos para el cubo, uno basado en mutaciones y otro apoyado en el algoritmo de Thistlethwaite [11]; y muestra que se puede obtener un rendimiento competitivo sin necesidad de tablas de consulta pre-calculadas [13].

   Sin embargo, no es frecuente encontrar en la literatura trabajos que empleen explícitamente EDAs con redes bayesianas para el cubo 3×3×3, lo cual deja un claro hueco para investigaciones que exploren este enfoque. Dado el espacio de soluciones tan vasto y las fuertes dependencias entre movimientos sucesivos (por ejemplo, un giro afecta la orientación y posición

de muchas piezas, lo cual condiciona los movimientos siguientes), un enfoque basado en modelos probabilísticos de dependencia (como los EDA con redes bayesianas) tiene sentido, ya que podría capturar secuencias efectivas de movimientos o sub-secuencias que tienden a aparecer en buenas soluciones, lo cual métodos puramente heurísticos podrían pasar por alto.

2.1.9. El "Número de dios" (God's Number)

El llamado *God's Number* para el cubo 3×3×3 es el *número máximo de giros necesarios para resolver cualquier configuración del cubo en el menor número de movimientos posible*. Para el juego de instrucciones con N, N' y N2 (el juego sin instrucciones extra que se utilizará en este proyecto), se ha demostrado que ese número es 20 [3]. Es decir, cualquier configuración del cubo puede resolverse en 20 movimientos o menos. La demostración ha sido una combinación de esfuerzo y trabajo conjunto a lo largo de tres décadas, y se ha demostrado empíricamente al solucionar prácticamente todo el espacio combinatorio (43,252,003,274,489,856,000 estados).

La tabla 2.1 muestra el número de configuraciones posibles para cada distancia al estado resuelto. Como se puede ver, el número de casos con bajas distancias es pequeño y crece exponencialmente hasta las distancias 17-18, que contienen el grueso de la combinatoria. La distancia 19 tiene menos configuraciones posibles que la 18; lo cual se debe en cierta medida a que, a partir de los 18 movimientos, nuevos movimientos a menudo no llevan a nuevos estados, sino a estados alcanzables con menos movimientos. De manera similar, la distancia 20 tiene un número de casos sorprendentemente pequeño comparado con otras largas distancias.

Distancia	Número de Posiciones	Distancia	Número de Posiciones
1	18	11	3,063,288,809,012
2	243	12	40,374,425,656,248
3	3,240	13	531,653,418,284,628
4	43,239	14	6,989,320,578,825,358
5	574,908	15	91,365,146,187,124,313
6	7,618,438	16	$\sim 1{,}1 * 10^{18}$
7	100,803,036	17	$\sim 1{,}2 * 10^{19}$
8	1,332,343,288	18	$\sim 2{,}9 * 10^{19}$
9	17,596,479,795	19	$\sim 1{,}5 * 10^{18}$
10	232,248,063,316	20	$\sim$490,000,000

Cuadro 2.1: Configuraciones posibles del cubo de rubik para cada distancia desde el estado resuelto [3]

Este resultado es relevante para los algoritmos de optimización porque establece una *cota superior firme* para el "lado difícil" del espacio de estados: no existe una configuración que requiera más de 20 movimientos óptimos. Por tanto, cualquier algoritmo automático podría aspirar a aproximarse o alcanzar esa cota, y los métodos probabilísticos o evolutivos pueden medirse en términos de cuán cerca llegan de esa óptima. Del mismo modo, el conocimiento de esta cota abre la

vía para diseñar heurísticas que aproximen la optimalidad, utilizarla como linea base de evaluación, o como parte de funciones de fitness (por ejemplo, penalizar secuencias que superen 20 movimientos o que se acerquen a dicha longitud).

También es relevante mencionar que los seres humanos suelen utilizar entre 120 (con los métodos más básicos) y 60 (los más avanzados) movimientos en total; un número muy superior al número de dios. Por otro lado, los algoritmos exactos, como el algoritmo de Kociemba, si bien no son óptimos para todos los estados en cuanto al número de movimientos, casi todas las soluciones que proporcionan suelen tener un número de movimientos menor o igual a 20.

Este límite inferior absoluto (20 movimientos) no solo tiene valor teórico, sino también práctico: sirve como referencia para *evaluar la calidad de las soluciones* obtenidas por cualquier algoritmo. Los métodos heurísticos o evolutivos pueden medirse por su distancia promedio al número de dios. Además, este conocimiento permite establecer funciones de fitness informadas o umbrales de poda adaptativos, integrando así conocimiento teórico en el diseño de sistemas de optimización evolutiva. En el contexto de este proyecto, se pretende acotar el número máximo de movimientos a un número superior pero cercano a 20, ya que no se busca alcanzar un número óptimo de movimientos, aunque se quiere cierto nivel de optimización.

2.2. Optimización Combinatoria

En está sección se discutirán los fundamentos teóricos de los algoritmos de optimización combinatoria, particularmente de los algoritmos de estimación de distribución, se revisarán tecnologías existentes relevantes, y se discutirá su aplicación al contexto de la resolución automática del cubo de Rubik.

2.2.1. Algoritmos Evolutivos

Los algoritmos evolutivos, o *Evolutionary Algorithms* (EA) [14], constituyen una amplia familia de métodos de IA centrados en la optimización, inspirados en los principios de la evolución natural y la selección biológica.

Estos algoritmos operan sobre una población de posibles soluciones a un problema dado y, de entre estas, seleccionan un grupo de las soluciones más prometedoras, aplicando algún tipo de métrica o función objetivo (fitness). Sobre este grupo de mejores soluciones de una población se aplican operadores estocásticos —como selección, recombinación y mutación— para guiar la búsqueda hacia regiones prometedoras del espacio de soluciones, construyendo una nueva población que, en principio, tendría un mejor "fitness". A cada una de estás poblaciones se les llama "generaciones". La figura 2.3 muestra el marco conceptual de un algoritmo evolutivo.

Entre los enfoques más representativos se encuentran [14]:

- Algoritmos genéticos (GA): representan soluciones como cadenas (bit, enteros, reals) y usan operadores de recombinación y mutación inspirados en

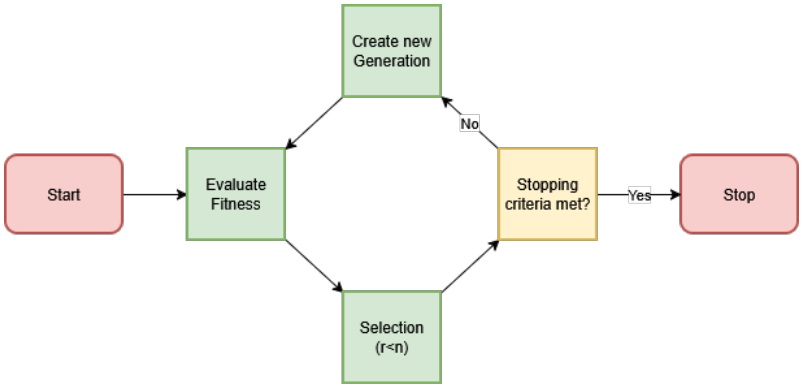

Figura 2.3: Esquema del funcionamiento de un algoritmo evolutivo. Fuente: elaboración propia

la genética clásica.

- Estrategias evolutivas (ES): orientadas a problemas continuos; se centran en la adaptación de parámetros de variación (por ejemplo, desviaciones estándar).
- Evolución diferencial (DE): opera sobre vectores reales y genera candidatos por combinaciones lineales estocásticas de individuos.
- Programación genética (GP): evoluciona programas o expresiones estructuradas (árboles), útil cuando la solución es un algoritmo o una regla.

Aunque difieren en la representación de las soluciones y en los mecanismos de variación utilizados, todos comparten la idea fundamental de evolucionar una población mediante presión selectiva. Los EA han sido ampliamente aplicados en optimización combinatoria, diseño de sistemas, aprendizaje automático, y control adaptativo con considerable éxito, debido a su flexibilidad y capacidad para explorar espacios de búsqueda no lineales y multimodales.

2.2.2. Otras metaheurísticas

Los paradigmas de inteligencia artificial metaheurísticos [15] son estrategias de optimización de alto nivel diseñadas para encontrar soluciones cercanas al óptimo en problemas complejos donde los métodos exactos resultan impracticables, especialmente en espacios de búsqueda grandes o no convexos. A diferencia de los algoritmos evolutivos, las metaheurísticas no evolutivas suelen inspirarse en procesos naturales, físicos o sociales y se basan en la mejora iterativa guiada por componentes estocásticos y reglas heurísticas, en lugar de operadores genéticos. Estos métodos se utilizan ampliamente en planificación, enrutamiento, aprendizaje automático y diseño de ingeniería, y muchos de ellos pueden extenderse a la optimización multiobjetivo, donde se optimizan simultáneamente varias funciones de fitness (a menudo conflictivas) para aproximar un conjunto de soluciones de compromiso en lugar de un único óptimo.

Algoritmos metaheurísticos no evolutivos relevantes incluyen [15]:

- Optimización por Colonias de Hormigas (Ant Colony Optimization (ACO)): inspirada en la búsqueda de caminos mediante feromonas en las hormigas.
- Optimización por Enjambre de Partículas (Particle Swarm Optimization (PSO)): basada en el movimiento colectivo y el intercambio de información en enjambres.
- Recocido Simulado (Simulated Annealing (SA)): inspirado en el proceso físico de recocido en metalurgia.
- Búsqueda Tabú: utiliza estructuras de memoria para evitar ciclos y óptimos locales.
- Búsqueda por Armonía (Harmony Search): inspirada en procesos de improvisación musical.
- Y algoritmos multiobjetivo como MOPSO (PSO Multiobjetivo) y MOACO, que manejan múltiples funciones de fitness de forma simultánea.

2.2.3. Algoritmos de Estimación de Distribución (EDAs)

En este contexto, los *Estimation of Distribution Algorithms* (EDA) [5] [16] surgen como una extensión natural de los métodos evolutivos tradicionales. A diferencia de los algoritmos genéticos convencionales, los EDA reemplazan los operadores clásicos de recombinación y mutación por un *modelo probabilístico* aprendido a partir de las soluciones más prometedoras de cada generación. En lugar de combinar directamente cromosomas, los EDA aprenden la estructura estadística del espacio de búsqueda, generando nuevas soluciones a partir de dicha distribución. Este enfoque permite capturar dependencias complejas entre variables y modelar de manera explícita la estructura del problema, superando así las limitaciones de los algoritmos genéticos tradicionales en entornos altamente correlacionados [6]. La figura 2.4 muestra el marco conceptual del algoritmo.

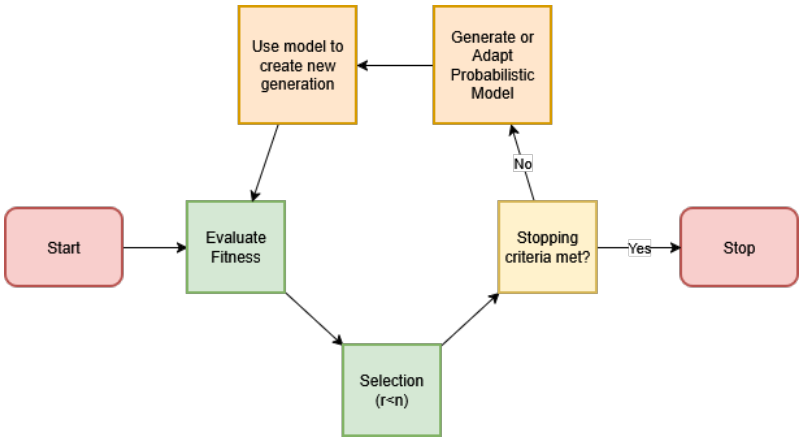

Figura 2.4: Esquema del funcionamiento de un algoritmo EDA. Fuente: elaboración propia

Los EDA surgieron en la década de 1990, inicialmente con modelos univariantes, los cuales suponen independencia entre variables; se estiman márgenes y se

muestrean de forma independiente. Son simples y eficientes, pero fallan cuando existe una fuerte interacción entre variables. Para solucionar esto, comenzaron a surgir modelos multivariantes que emplean estructuras que capturan dependencias condicionales, como redes bayesianas, gaussianas multivariantes y árboles de dependencias. Estas representaciones mejoran la capacidad de modelar problemas con interacciones complejas, pero incrementan el coste de aprendizaje estructural y el coste de muestreo. El avance hacia modelos multivariantes permitió representar relaciones condicionales, consolidando una nueva generación de algoritmos más expresivos y potentes [6].

2.2.4. EDAs con modelos gráficos

Los *Modelos Gráficos Probabilísticos* (MGP) [6] constituyen una de las herramientas más potentes dentro del aprendizaje estadístico moderno para representar y razonar sobre distribuciones multivariantes de manera estructurada. En esencia, un MGP combina teoría de grafos y probabilidad, describiendo cómo un conjunto de variables aleatorias se relaciona mediante dependencias condicionales. Su ventaja central es que permite factorizar la distribución conjunta en componentes locales, lo que hace posible el aprendizaje y la inferencia eficiente incluso en espacios de alta dimensionalidad.

Existen dos grandes familias de modelos gráficos:

- Modelos dirigidos, como las redes bayesianas (Bayesian Networks, BN) [6], donde los nodos representan variables y las aristas dirigidas indican relaciones causales o de dependencia condicional. Cada variable está asociada a una distribución condicionada a sus "padres" en el grafo, de modo que la distribución conjunta se expresa como el producto de todas esas distribuciones locales. Considere la figura 1.3 como ejemplo de una red bayesiana.
- Modelos no dirigidos, como las redes de Markov (Markov Random Fields, MRF ó Markov Chains) [17], donde las aristas no tienen dirección y representan relaciones simétricas entre variables. Aquí la distribución conjunta se factoriza en potenciales definidos sobre cliques del grafo, siguiendo el principio de Markov.

Además, existen variaciones y modelos mixtos:

- Modelos mixtos (híbridos) que combinan variables discretas y continuas, o integran componentes latentes, ofrecen mayor flexibilidad para problemas de estructura compleja.
- Modelos dinámicos, como las redes bayesianas dinámicas (DBN) o los campos aleatorios condicionales (CRF), que incorporan relaciones temporales o secuenciales.

Todos ellos comparten la capacidad de capturar correlaciones complejas entre subconjuntos de variables y de restringir el espacio de búsqueda probabilístico a aquellas configuraciones compatibles con la estructura del grafo, aunque utilicen métodos distintos que sobresalen en diferentes aspectos. En el contexto de los algoritmos EDA, esta propiedad resulta especialmente útil: mientras que los

modelos univariantes simples ignoran las interdependencias, los modelos gráficos permiten aprender explícitamente las relaciones entre los componentes de la solución, mejorando así la *eficiencia* de la búsqueda global.

El uso de modelos gráficos en EDA aporta varias ventajas clave:

- La captura explícita de interdependencias permite identificar subestructuras o patrones de co-ocurrencia entre componentes de la solución.
- La estructura del grafo ofrece una representación visual e intuitiva de cómo se relacionan las variables del problema.
- Modularidad y reutilización: los componentes del modelo pueden aprenderse o modificarse de forma local, lo que facilita adaptaciones y extensiones.

Sin embargo, también presentan desafíos relevantes:

- El aprendizaje estructural tiene complejidad súper-exponencial en el número de variables, por lo que deben emplearse heurísticas, restricciones o estructuras fijas parciales.
- En problemas de alta dimensionalidad (como los estados del cubo de Rubik), la estimación precisa de parámetros requiere grandes muestras o estrategias de regularización [5].
- La generación de muestras válidas puede ser costosa si la estructura del grafo es densa o si existen restricciones combinatorias fuertes.

En el contexto de este proyecto, se pretende utilizar un EDA basado en redes bayesianas.

2.2.5. Estrategias híbridas y mejoras prácticas

Dado el coste de aprendizaje de modelos complejos y las estructuras complejas de los problemas, las implementaciones prácticas suelen combinar técnicas que suplementan la funcionalidad de los EDA y mejoran su eficiencia:

- Optimizaciones locales: Usando las regiones prometedoras dadas por el EDA, se puede explorar la vecindad aplicando búsqueda local y así refinar las soluciones.
- Conocimiento del dominio: Para cada problema, se pueden aplicar optimizaciones que provienen del conocimiento del dominio. En el caso del cubo de Rubik, se pueden simplificar las soluciones eliminando elementos cíclicos; y explorando estados equivalentes y simétricos que se puedan aplicar a cada solución.
- Jerarquización: Factorizar el problema en sub-problemas y aplicar distintos modelos a cada problema para reducir la complejidad en cada nivel.
- Restricciones del aprendizaje estructural: Aplicar restricciones al modelo gráfico, como limitar el número de nodos padre o usar estructuras predefinidas o aprendizaje incremental.

- Además, se pueden usar estrategias híbridas, tomando elementos de otros paradigmas para mejorar el sistema.

2.2.6. Implementaciones

La implementación práctica de un EDA con modelos gráficos requiere combinar herramientas de optimización evolutiva, aprendizaje probabilístico y soporte para estructuras de datos flexibles; sobre todo para implementaciones óptimas de modelos gráficos. En el ecosistema actual de Python [18], existen varias librerías [19] que facilitan la experimentación con estos algoritmos, siendo las más destacadas DEAP, pgmpy, pyAgrum, y EDAspy. En otros entornos como R también existen herramientas relevantes.

- *DEAP* (Distributed Evolutionary Algorithms in Python) [20] es un marco genérico para algoritmos evolutivos que proporciona componentes modulares para la definición de individuos, funciones de evaluación, estrategias de selección y mecanismos de paralelización. Aunque no está orientado específicamente a EDAs, su flexibilidad lo convierte en una excelente base para implementar variantes personalizadas o combinar EDA con operadores genéticos clásicos.
- *pgmpy* (Probabilistic Graphical Models in Python) [21], por su parte, ofrece una infraestructura completa para el modelado y la inferencia en redes bayesianas y otros modelos gráficos. Permite definir grafos dirigidos o no dirigidos, aprender estructuras y parámetros a partir de datos, y realizar muestreo (sampling) para generar nuevas instancias. En el contexto de un EDA, pgmpy resulta útil para la etapa de estimación y muestreo del modelo probabilístico, especialmente en algoritmos como el Bayesian Optimization Algorithm (BOA) o sus variantes dinámicas.
- *pyAgrum* [22] es una biblioteca de código abierto escrita en C++ con interfaz para Python, diseñada para la creación, manipulación e inferencia en modelos gráficos probabilísticos, como redes bayesianas, redes de Markov y diagramas de influencia. Proporciona un conjunto amplio de herramientas para el aprendizaje de estructuras y parámetros a partir de datos, así como para la realización de inferencias exactas o aproximadas. Su API en Python permite una integración sencilla con otros entornos de análisis y aprendizaje automático, lo que la convierte en una opción versátil tanto para la investigación en inteligencia artificial como para aplicaciones prácticas en modelado probabilístico.
- *EDAspy* [23] es una de las pocas librerías dedicadas exclusivamente a la implementación de EDAs en Python. Desarrollada con un enfoque académico y de investigación, EDAspy ofrece una colección de algoritmos univariantes y multivariantes listos para su uso, así como una arquitectura flexible para definir nuevos modelos y operadores. Su diseño modular permite separar las fases de selección, modelado y muestreo, lo que facilita la experimentación con distintos enfoques probabilísticos.

Para este proyecto se pretende apoyarse en *EDAspy*, debido a algunas de sus

características ventajosas para el proyecto, como: Implementaciones base de variantes clásicas de EDA como UMDA (Univariate Marginal Distribution Algorithm), PBIL (Population-Based Incremental Learning), EMNA (Estimation of Multivariate Normal Algorithm) y BOA (Bayesian Optimization Algorithm); Compatibilidad con modelos discretos y continuos, mediante diferentes representaciones de individuos y funciones de densidad; Soporte para aprendizaje de modelos gráficos, permitiendo usar estructuras como redes bayesianas o gaussianas; Facilidad de integración con otras librerías científicas, como *NumPy* [24], *pandas* [25] y *scikit-learn* [26], lo que posibilita combinar EDA con análisis estadístico o aprendizaje automático; y Herramientas de visualización y trazabilidad, que permiten registrar la evolución del modelo y el progreso de la optimización.

Capítulo 3

Modelo e Implementación del Cubo

En este capítulo se explicarán el modelo e implementación de la parte del proyecto del propio cubo, así cómo elementos tangenciales, como secuencias de movimientos, simetrías, etc. La función objetivo (fitness) se discutirá en el capítulo referente al propio solver.

3.1. Definiciones

Modelamos el cubo de Rubik como un grafo de espacio de estados. El conjunto de todos los estados alcanzables se denota como S; el estado inicial (scrambled) se denota por S_s; el estado resuelto (full) por S_F; y una configuración arbitraria del cubo por S_i.

De forma análoga, para las cadenas de movimientos (maneuvers), denotamos el conjunto de todas las cadenas posibles como M; la cadena que lleva de S_F a S_S como M_S, y una cadena cualquiera M_i.

3.2. Estructura

El enfoque más común en simuladores de cubos de rubik es usar un único *array* de longitud 54 (o 6 de longitud 9), en el que cada posición es un *sticker*. Este es un enfoque sencillo y eficiente en cuanto al espacio de memoria y con baja complejidad en las operaciones de acceso y búsqueda, pero conlleva un gran inconveniente. Ya que la estructura de datos no está modelando la estructura real del cubo ni las restricciones (si no una representación aproximada y simple que carece de estas características), no existe una manera formal, unificada y consistente de realizar las operaciones. En su lugar, las operaciones se deben *mapear* a mano, creando tablas que describen las transformaciones necesarias de antemano (es decir, qué posiciones del array van a qué nuevas posiciones), lo cual es un proceso manual, pesado y complicado de seguir en código; sin embargo, es considerablemente rápido en ejecución, ya que no se deben hacer

cálculos durante las operaciones (ya que está pre-calculado implícitamente en los mapas). El segundo inconveniente de este modelo es que no proporciona información sobre la relación entre las piezas (o stickers), ya que, de nuevo, no modela la estructura real del cubo, sino una cruda representación. Tener formas de conocer la relación entre las piezas y demás información es de especial interés en este proyecto por varias razones, en especial para el calculo de la función objetivo (fitness).

Otro enfoque común es el uso de programación orientada a objetos; modelando las piezas (o *cubies*) del cubo como objetos, y organizándolos en una estructura de datos. Este enfoque sigue mejor la estructura real del cubo y permite no tener que usar mapas pre-computados. Además, ofrece la posibilidad de crear un modelo que represente mejor el cubo real y capture aspectos como las restricciones, la relación entre las piezas y otros.

En este proyecto, el modelo del cubo se basa en un sistema de coordenadas en tres dimensiones, en el que cada coordenada está asociada a un objeto que representa una pieza. Este modelo es una representación formal y matemática de la estructura real del cubo, lo cual nos permite apoyarnos en el álgebra para modelar movimientos y operaciones de forma consistente y formal; así como tener acceso a información sobre el estado del cubo para comparativas u otras utilidades.

3.2.1. Coordenadas

El sistema de coordenadas es un sistema de coordenadas cartesianas tridimensionales, centrando el origen en el punto interior del cubo como $(x, y, z) = (0, 0, 0)$ y con $x, y, z \in [-1, 1]$, y considerando los ejes, respectivamente, de izquierda a derecha, de abajo hacia arriba y de atrás hacia adelante. Así pues la pieza entre las caras "R", "U", y "F", (esquina frontal superior derecha) estaría asociada a las coordenadas $(x, y, z) = (1, 1, 1)$; mientras que la pieza entre las caras "L", "D", y "F" sería $(x, y, z) = (-1, -1, 1)$.

Cada "tipo" de pieza (esquinas, aristas, y centros) se puede identificar por la forma de las coordenadas. Las esquinas no presentan ningún cero (forma (a, b, c)), las aristas presentan un cero (formas $(0, a, b)$, $(a, 0, b)$ ó $(a, b, 0)$); y los centros presentan dos ceros (formas $(a, 0, 0)$, $(0, a, 0)$ y $(0, 0, a)$). Asimismo, se pueden identificar todas las coordenadas de una cara porque tienen el valor no nulo de su centro igual (por ejemplo, todas las coordenadas con la forma $(-1, a, b)$ son de la cara izquierda, ya que el centro izquierdo es $(-1, 0, 0)$.

El modelo con coordenadas permite utilizar álgebra para las operaciones de movimiento. Para rotar una cara en sentido horario, podemos aplicar la fórmula de rotación antihoraria en un plano $(x, y) \to (-y, x)$, reduciendo primero (x, y, z) eliminando la ordenada representativa de la cara que se mueve. Es decir, en el movimiento "U", para calcular las nuevas coordenadas de la pieza en $(1, 1, 1)$, primero tomamos las coordenadas del centro de la cara superior, es decir, $(0, 1, 0)$, y con ello obtenemos $(x, y, z) = (1, 1, 1) \to (x, z) = (1, 1)$; con esto, aplicamos la fórmula y obtenemos $(-1, 1)$, tras lo cual restauramos la coordenada de la

cara, teniendo como resultado final $(x, y, z) = (-1, 1, 1)$. Nótese que la formula $(x, y) \to (-y, x)$ funciona para las caras con coordenadas positivas (”R”, ”U”, ”F”), pero los sentidos están invertidos para las caras con coordenadas negativas (”L”, ”D”, ”B”).

De la misma forma, los movimientos dobles y los movimientos en sentido antihorario se realizan aplicando esta operación dos y tres veces (”U” tres veces es equivalente a ”U’ ”) respectivamente.

3.2.2. Las Piezas

Las piezas están modeladas con un objeto simple que encapsula una *lista* de los *stickers* (colores) que tiene la pieza. Esta lista está ordenada con respecto a las coordenadas (x, y, z), de tal manera que el sticker que se encuentra en el eje x estará en primera posición, el sticker en y en segunda, y el de z en tercera; dejando cadenas vacías en las posiciones que no tienen stickers (en el caso de aristas y centros). Así pues, con el cubo colocado con la cara amarilla arriba y la roja en el frente, la arista en $(0, 1, 1)$ sería [Vacío, Amarillo, Rojo].

Ya que este enfoque sigue el mismo orden que las coordenadas, la ”rotación” de las piezas se lleva a cabo con el mismo sistema, usando exactamente el mismo proceso que los movimientos de cara. Con el mismo ejemplo de la arista en $(0, 1, 1)$ = [Vacío, Amarillo, Rojo], al realizar ”U”, pasará a estar en $(-1, 1, 0)$ y su lista pasará a ser [Rojo, Amarillo, Vacío].

3.2.3. Representación Gráfica

La representación gráfica es sencilla; simplemente es el desarrollo en plano del cubo, representando los colores de cada pegatina con la inicial del color en inglés, como se mostró en las figuras 3.1 y 3.2. Además, hacemos uso del paquete de Python ”colorama” para añadir color a las letras y facilitar su seguimiento. Para hacer más fácil el seguimiento, también definimos la rotación canónica como aquella con la cara amarilla arriba y la cara roja en el frente.

Figura 3.1: Desarrollo en plano del cubo en estado resuelto

Figura 3.2: Desarrollo en plano del cubo tras aplicar "U"

De forma inversa, se puede introducir el desarrollo en plano del cubo como entrada, y el cubo se ajustará a él. Nótese que no se consideró necesario introducir una comprobación de estados posibles en esta última funcionalidad, si no que se delega esta responsabilidad al usuario/llamante.

3.3. Secuencias de Movimientos

Las cadenas de movimientos son otra parte muy importante del proyecto, ya que estas serán los individuos en nuestro algoritmo evolutivo, y hay un cierto número de operaciones y particularidades que discutir sobre ellas.

3.3.1. Simplificación y Canonización

Las secuencias de movimientos [R U U] y [R U2] son equivalentes, pero tienen diferente longitud. Esto se debe a que [R U U] no está simplificada. Ya que una de las métricas de eficiencia en este proyecto es el número de movimientos, y dado que generaremos secuencias aleatorias de movimientos que posiblemente no estén simplificadas, simplificar éstas es necesario.

El proceso de simplificación es muy sencillo. Podemos contabilizar el numero de cuartos de rotación que hace cada movimiento: los movimientos en sentido horario son 1 cuarto (o 90º), los movimientos dobles son 2 cuartos y los movimientos antihorarios son 3 cuartos; después, podemos tomar el módulo 4 del resultado, y ese será el movimiento simplificado. Cuando varios movimientos sobre la misma cara aparecen seguidos, aplicamos este mismo proceso. Por ejemplo, en la secuencia [U U U2 U' U U2] tenemos 10 cuartos en total, o 2 en módulo cuatro, y podemos simplificarlo como [U2]. Podemos encontrar también secuencias de movimientos como [R U U' R], donde [U U'] se simplifican como una secuencia vacía, dejando [R R], que se simplifica como [R2].

La implementación tiene complejidad lineal $O(n)$ al usar una pila. Guardamos el movimiento actual en la pila y lo comparamos con el siguiente: si tienen la

misma letra, simplificamos, sustituyendo el movimiento en la pila por el movimiento simplificado y pasamos al siguiente; cuando una simplificación es una secuencia vacía, la cima de la pila pasa a ser el movimiento anterior, permitiendo simplificar cadenas del tipo visto antes.

Además, existe otro tipo de caso simplificable que este sistema es capaz de manejar. Tiene que ver con la conmutatividad de los movimientos. En general, en el cubo de rubik, los movimientos no son conmutativos: [R F] no es igual que [F R], ya que el primer movimiento afecta al estado del que se parte para el segundo movimiento. Sin embargo, los movimientos en caras opuestas (R con L, F con B, y U con D) sí son conmutativos, ya que afectan distintas áreas del cubo. Así pues [R L] es igual que [L R]. Esto abre la puerta a dos consecuencias importantes.

- La primera es que existen *cadenas simplificables* que nuestra anterior comprobación no es capaz de remediar por si sola, donde existen movimientos en caras opuestas entre movimientos de una misma cara, por ejemplo: [F R L2 R'] que es simplificable a [F L2], ya que [R] y [R'] se cancelan entre sí. Esto es relativamente sencillo de añadir en nuestro anterior sistema. Además de las comprobaciones ya existentes, miramos la posición anterior de la cima de la pila y comprobamos si son conmutables: de serlo, si el próximo símbolo leído es en la misma cara que la cima-1, estos serán simplificables.
- La segunda consecuencia tiene que ver con la *canonización de las cadenas*. Para poder guiar a nuestro sistema adecuadamente, es importante que todas las cadenas equivalentes sean representadas de la misma manera. Dado que [L R] y [R L] (y otras cadenas con símbolos opuestos uno detrás del otro) son equivalentes, debemos establecer un orden estándar para que aparezcan siempre iguales. El orden decidido arbitrariamente es que R precede a L, U precede a D, y F precede a B. Así pues, una cadena como [L R U2 B F] quedaría canonizada como [R L U2 F B].

3.3.2. Eliminación de Ciclos

Como se discutió en la sección 2.1.5, además de ciclos simplificables por la regla descrita anteriormente (como [U U']) existen ciclos que surgen por la estructura del cubo y que no son detectables con facilidad. Por supuesto, ciclos podrían aparecer en las secuencias de movimientos generadas como individuos en cada fase (aunque de forma excepcionalmente rara), y resulta importante poder detectarlos y eliminarlos para evitar que el sistema refuerce estás cadenas costosas por longitud que en verdad no añaden valor.

La detección es costosa; no hay otra opción que simular la secuencia de movimientos entera, guardando el estado del cubo tras cada movimiento, y comprobar que no haya dos estados iguales. Si los hay, existe un ciclo, y se elimina de la cadena. Esto es considerablemente más costoso que la simplificación, pero aun así es considerablemente rápido, ya que ni los procesos de copia de estados ni los procesos de realizar movimientos son particularmente costosos. Si se ha encontrado un ciclo en una cadena, se vuelven a intentar la simplificación y la eliminación de ciclos, ya que podrían haber surgido nuevos, por ejemplo, [F R

[ciclo] R'] donde R y R' son simplificables una vez el ciclo es eliminado.

Para algunos de los números dados como periodicidad de ciclo de ciertas secuencias de movimientos en la sección 2.1.5 (como que la periodicidad de [R U] es 35), se utilizó este mismo sistema de detección junto con un pequeño sistema que, dada una secuencia de movimientos, la va duplicando y probando a detectar ciclos, contando luego el número de repeticiones.

3.3.3. Simetrías

En la sección 2.1.6 hablamos de dos tipos principales de simetría que podrían ayudar.

- Las simetrías axiales, como se apuntaba anteriormente, son triviales de calcular, ya que podemos obtener el "espejo" de una cadena de movimientos cualquiera de forma sencilla, simplemente una traducción en la que el eje de simetría determina el nuevo símbolo y simular esta cadena en el cubo. Estás simetrías podrían ser utilizadas para reducir el espacio de búsqueda total del cubo, disminuyendo el número de cadenas de movimientos totales.
- Las simetrías de re-coloración son algo más complejas. En esencia, para este proyecto, también se basan en un problema de traducción, pero a través de varios ejes rotacionales en lugar de un solo eje axial. Esto lleva a necesitar un mapa de traducción un poco más complejo, además de tener que hacer traducciones en cadena para llegar a las traducciones buscadas.

Estás dos funcionalidades fueron implementadas inicialmente. Sin embargo, no aportan ninguna utilidad real al sistema solver. Lo que estás simetrías podrían hacer es acelerar el movimiento de las primeras generaciones hacia espacios prometedores. Una vez que ya se está en espacios prometedores, serían inútiles. Por no mencionar que el sistema ya llega a espacios prometedores en las primeras generaciones. La primera generación es una distribución completamente aleatoria y uniforme, por lo que tendrá de media individuos que representan tanto un espacio como su simétrico. Si bien esto es aleatorio, incluso si hubiese un individuo más prometedor en el espacio espejo, el sistema tardaría muy poco en llegar hacia el mismo nivel. En conclusión, añadir estos al propio solver solo incurriría en un coste computacional extra que no aportaría beneficios suficientes para justificarlo, pero fueron implementados de todas formas.

Finalmente, debemos mencionar brevemente la inversión de cadenas, que es muy similar en funcionamiento a estas dos. La inversa de una cadena de movimientos es la cadena que deshace esos movimientos, por ejemplo, la inversa de [R F U2 L'] es [L U2 F' R']. Esto es trivial de calcular y se utilizará para dar la solución en última instancia.

Capítulo 4

Modelo e Implementación del Solver

En este capítulo se explicará el modelo e implementación del sistema solver en su totalidad, así como todos los elementos utilizados, el fitness y demás.

4.1. Concepto Base

La información disponible inicialmente es el estado scrambled S_S y el estado resuelto S_F, y buscamos la cadena de movimientos que nos lleve de S_F a S_S: M_S. Invertir esta nos dará la solución real. Nótese que este problema es equivalente a su inverso, de S_S a S_F, con la cadena solución siendo la inversa de M_S, pero resulta más cómodo resolverlo así.

Recordando la sección 2.2, los EDA son un tipo de algoritmo evolutivo que genera sus nuevas poblaciones a través de un modelo probabilístico. Los individuos de las poblaciones en nuestro modelo van a ser cadenas de movimientos M_i que toman el estado resuelto S_F como punto inicial y llegan a un estado S_i, que queremos que sea S_S o cercano.

4.2. Fitness

La función objetivo, o fitness, de forma simple, debe proporcionar una estimación de cuán cerca está un individuo de la solución con un bajo coste computacional (ya que se espera utilizarla cientos o miles de veces por generación). Idealmente, la función fitness da una estimación muy cercana a la distancia real entre estados, pero en el problema del cubo de Rubik, esto es particularmente complicado.

Dentro de nuestro modelo, podemos comparar el estado S_i (resultante de aplicar M_i (nuestro individuo) desde S_F) con el estado scrambled S_S.

4.2.1. Similitud como Fitness

La primera opción que viene a la mente es estimar el fitness como una función de similitud entre dos estados del cubo. Se pueden hacer heurísticas muy baratas (con complejidad lineal $O(2n)$) que utilizan esto.

- Similitud en stickers: La opción más sencilla es comparar los stickers uno a uno en un cubo y el otro.
- Similitud en la posición de las piezas: Gracias a nuestra implementación de coordenadas, podemos comparar las posiciones de las diferentes piezas de forma sencilla y económica. Podemos comparar si dos piezas están en el mismo lugar o no, pero también podemos obtener una medida de distancia entre la posición de la pieza en un estado y en otro (distancia para *una* pieza, no distancia en movimientos entre dos estados completos).
- Similitud en la orientación de las piezas: Es algo más complicado y, por si sola, a menudo no proporciona información real. Al realizar movimientos, no solo cambia la posición, sino también la orientación. Así pues, solo tiene sentido comparar la orientación en piezas correctamente colocadas y en combinación con otras heurísticas que den información de la posición.
- Similitud Mixta: Podemos utilizar varias de estas heurísticas combinadas de varias formas para conseguir un fitness final lo más adecuado posible.

La tabla 4.1 muestra los resultados de las diferentes funciones de similitud entre dos estados del cubo a diferentes distancias, a fin de evaluar y comparar. El comportamiento ideal sería lineal con la distancia, con similitud 100 a distancia 0, y similitud 0 a distancia 20. La entrada J-Perm muestra la similitud entre un estado S_1 y otro S_2 que es el resultado de aplicar la permutación J en el estado S_1 (más sobre esto a continuación). La figura 4.1 presenta estos resultados en un gráfico para facilitar la comparación.

Distancia	Stickers	Posición	Sti+Pos	Pos+Ori	Pos+Ori+Dist
0	100.0	100.0	100.0	100.0	100.0
1	75.0	86.7	85.5	76.5	63.7
2	54.2	78.3	74.8	60.0	44.1
3	47.9	73.3	68.0	52.7	30.4
4	35.4	61.7	51.6	46.1	24.4
5	29.2	51.7	40.5	36.8	16.9
6	33.3	51.7	44.0	42.5	30.1
7	35.4	58.3	49.6	50.4	31.8
10	25.0	58.3	39.0	40.2	19.6
15	27.1	70.0	47.0	46.5	25.6
20	18.8	58.3	29.6	37.7	17.7
J-Perm	87.5	93.3	92.8	88.3	81.9

Cuadro 4.1: Resultados de las diferentes medidas de similitud propuestas

Conclusiones sobre cada medida:

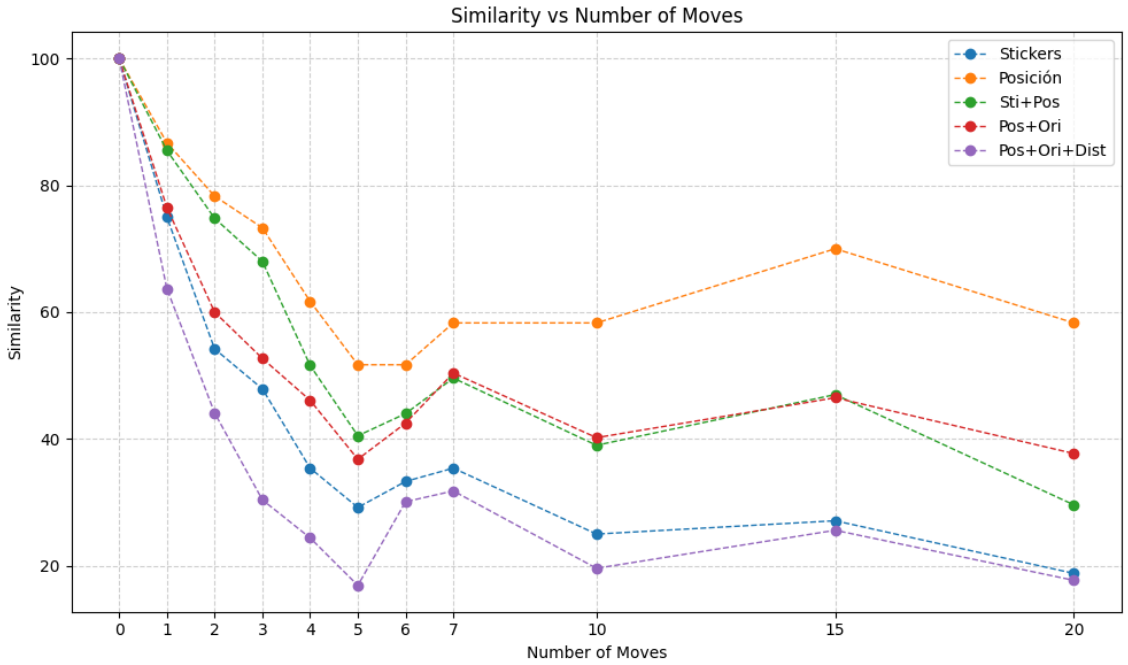

Figura 4.1: Gráfico mostrando las diferentes medidas

- La medida de similitud en stickers tiene poca resolución en distancias pequeñas y tiende a ser pesimista, pero funciona relativamente bien para largas distancias.
- La medida de similitud en posición da la mayor resolución para distancias cortas, pero es muy optimista, con mínimos alrededor del 50 por ciento (lo cual tiene sentido, ya que, si una pieza está muy lejos, otras piezas estarán más cerca).
- La medida mixta de stickers y posición es muy prometedora, ya que sus componentes se complementan mutuamente: la medida de posición tiene buena resolución a corta distancia y mala a larga distancia, mientras que la medida de stickers es todo lo contrario. Utilizando pesos adaptativos para dar mayor peso a cada medida en su área de excelencia, la medida es la más lineal y con mayor sensibilidad, como se aprecia en la gráfica.
- La medida de posición y orientación se comporta de forma algo similar a la de stickers y posición, pero con menor resolución en ambos extremos.
- La última medida, que combina una estimación de distancia con piezas correctas y bien orientadas, es la más pesimista de todas y la que tiene menor resolución.

Conclusiones generales sobre la similitud como fitness:

- Como podemos ver, a partir de los 5 movimientos de distancia, las medidas de similitud comienzan a ser peores como estimaciones de distancia. Las medidas de similitud se vuelven peores en estados lejanos y, a menu-

do, la similitud puede empeorar aunque la distancia sea menor, o mejorar aunque la distancia sea mayor. Esto es un problema considerable, ya que los algoritmos evolutivos necesitan una función de fitness que sea lineal, gradual y realmente representativa de la distancia a la solución; de lo contrario, la búsqueda puede tener dificultades para llegar a estados mejores, que necesitan pasar por estados con menor fitness primero. Este es un problema que puede ser eludido considerablemente con la implementación de cierta cantidad de búsqueda local y otros añadidos al algoritmo, lo que permitiría navegar estas pequeñas pérdidas mejor.

- La entrada "J-Perm" de la tabla 4.1, que no está representada en el gráfico, nos muestra el segundo (y mayor) problema de las medidas de similitud como fitness. La J-Perm se trata de una cadena de movimientos ([R U R' F' R U R' U' R' F R2 U' R' U']) que intercambia la arista $(0, 1, 1)$ y la esquina $-1, 1, 1$ con la arista $(-1, 1, 0)$ y la esquina $(-1, 1, -1)$, respectivamente. Es decir, es una cadena de movimientos cuyo estado resultante es muy similar al estado inicial, pero con una distancia entre estados considerablemente grande; lo cual demuestra que existen estados similares, pero cuya distancia es grande. Esto lleva al sistema a dar altas puntuaciones a cadenas que realmente no se acercan a la solución, y será uno de los mayores retos del uso de medidas de similitud como fitness.

4.2.2. Medidas alternativas de fitness

Consideremos formas alternativas de fitness a las medidas de similitud para intentar evitar el problema que supone que la similitud a menudo no es una buena estimación de la distancia.

- Calcular la distancia real: Calcular la distancia real, en esencia, es saber los movimientos que resuelven el cubo; es decir, resolver el cubo en sí. El mismo propósito de este trabajo es resolver el cubo con un solver basado en EDA; por lo tanto, resolver el cubo primero con otro método para luego solucionarlo con el sistema principal me parece fuera del contexto del proyecto y, asimismo, algo ridículo. Además, este sigue siendo un proceso muy costoso computacionalmente. Incluso con algoritmos de búsqueda apropiados como IDA*, se incurriría en un costo computacional considerable, mucho mayor que el costo de las medidas de similitud. La forma más barata de hacer esto sería utilizando algoritmos exactos (vistos en la sección 2.1.8), como el algoritmo de Kociemba, que seguiría teniendo un coste computacional mucho mayor al de las medidas de similitud, aunque mucho más bajo que IDA*. Pero, de nuevo, obtener la solución con otro método para solucionarlo con el método defendido me parece incorrecto. Nótese que la distancia debería ser el número de movimientos óptimos, por lo que métodos de resolución de estilo humano (que a nivel computacional son extremadamente baratos) no valdrían, ya que darían distancias totalmente distintas que no guardan relación con la distancia mínima.
- Base de datos de patrones (Pattern Data Base): En lugar de estimar la distancia computacionalmente en tiempo de ejecución, podemos pre-computar

distancias exactas en una base de datos a la que accedemos en tiempo de ejecución. Ya que el espacio entero del cubo es inmenso, se podría usar un sub-grupo del cubo para estimar la distancia, perdiendo precisión a cambio de reducir el tamaño de la base de datos muchos órdenes de magnitud. Por ejemplo, siguiendo parte del sistema del algoritmo de Kociemba, podríamos fijarnos únicamente en las 8 esquinas (que da un número de estados de aproximadamente $8!$) y calcular la distancia de cada uno de esos estados a todos los demás. El problema es que, incluso con este espacio reducido, la base de datos acabaría teniendo hasta ($(8!)^2 = 1{,}6 * 10^9$) estados (realmente menos por repeticiones, posiciones ilegales, etc., pero aun así en el orden de 10^8 o 10^7), lo cual es un costo considerable de memoria y computación que es sensible considerar fuera del alcance del proyecto. Además, esta seguiría sin ser una medida de distancia exacta, aunque sería mucho más precisa que las medidas de similitud.

Es por la falta de alternativas adecuadas que se ha decidido continuar con medidas de similitud y mitigar sus defectos dentro del propio sistema. Si existe otra alternativa mejor, no se ha encontrado en el desarrollo del proyecto.

4.3. Procesamiento de individuos

Antes de discutir el algoritmo en sí, debemos comentar ciertas particularidades en torno a la implementación con el uso de EDAspy y con la propia naturaleza del problema que afectan el procesamiento de los individuos.

4.3.1. Canonización

Como se discutió en la sección 3.3, es importante simplificar las cadenas de movimientos a su estado más corto y canonizarlas para evitar individuos aparentemente diferentes que en realidad son iguales. Por eso, antes de calcular el fitness, debemos aplicar esto a todos los individuos. Para ello utilizamos las funcionalidades descritas en la sección 3.3.1.

4.3.2. Sub-cadenas

Por ejemplo, supongamos que la solución para cierto cubo es M_S [U'F L U B]; puede darse que entre nuestros individuos generemos una cadena M_i que *contenga* la cadena solución M_S, pero que tenga movimientos extra a ambos lados, como puede ser M_1 [D L U' F L U B R] ([D L M_S R]). Si evaluamos el individuo usando el estado final S_1 resultante de aplicar M_1, el fitness dado será posiblemente malo y se pierde el hecho de que existe una sub-cadena mejor (incluso la solución) contenida en él.

Siguiendo el ejemplo anterior, podemos ver que cualquiera de nuestros individuos M_i puede contener una sub-cadena M_{i_j} cuyo fitness es mejor que el de la cadena entera (incluyendo la solución). Por ende, no basta solo con calcular el fitness de la cadena entera M_i, sino que se debe calcular el fitness para todas las sub-cadenas M_{i_j}. De entre todas ellas, podemos escoger aquella con mejor

fitness y tomarla como representante del individuo. Por ejemplo, para un individuo M_1 [R U F R'] debemos considerar los M_{1_j} [R], [R U], [R U F], [R U F R'], [U], [U F], [U F R'], [F], [F R'], y [R'] (aunque podríamos ignorar las cadenas de longitud uno), evaluar el fitness de todas ellas y escoger la mejor (inclusive la solución, si fuese una sub-cadena). Digamos en el ejemplo, que [U F R'] tiene el mayor fitness, así pues este pasaría a ser el individuo y será lo que usaremos para entrenar a la BN.

Este enfoque tiene ventajas y desventajas. Si los individuos de una población tienen longitud N, deberemos evaluar $\frac{N(N+1)}{2}$ sub-cadenas por cada individuo. Esto está incrementando considerablemente el espacio de búsqueda y la información que proporciona cada individuo, pero también incurre en un coste computacional mucho mayor (aunque se puede implementar de forma considerablemente eficiente para ahorrar un número considerable de operaciones en cuanto a la simulación de los movimientos).

4.3.3. Uso de EDAspy

EDAspy ofrece modelos completos y totalmente funcionales que ejecutan EDAs puros, incluyendo paralelización y otros aspectos. En nuestro caso, sin embargo, queremos poder manejar el proceso de forma detallada y añadir varias capas de procesamiento a los individuos, e incluso a los procesos de selección y la creación de nuevas generaciones.

Para ello, en vez de usar los algoritmos en sí, usaremos solo la BN discreta incluida en el algoritmo EBNA de EDAspy, que cuenta con una interfaz sencilla de utilizar, además de paralelización en el aprendizaje y acceso a ciertos parámetros de ajuste. Aparte de la inicialización de la BN, donde ajustaremos estos parámetros, solo utilizaremos los métodos "learn()" y "sample()" para interactuar con ella, que respectivamente entrenan la BN a una población y generan una nueva población a partir del modelo.

1. **Codificación para la BN**: Para usar el método "learn()" para entrenar la BN, EDAspy espera un cierto formato de datos específico.

   La BN esencialmente toma individuos que son listas de variables que pueden adoptar ciertos valores especificados. Nuestros individuos son cadenas de movimientos, en esencia una lista de "strings" (cadenas de caracteres, ya que movimientos como [U2] o [F'] utilizan dos caracteres). Podemos verlos como listas en las que cada movimiento es una variable que puede tomar uno de 18 valores posibles (18 es el número de movimientos total explicados en la sección 2.1.4). Sin embargo, la BN de EDAspy no soporta "strings" como valores de la variable. En su lugar, debemos traducir estos a números, lo cual es trivial, asignando a cada posible movimiento un valor entre 0 y 17. Antes de entrenar el modelo, debemos codificar la población convirtiendo las cadenas de movimientos en listas de números del 0 al 17, de forma que la BN pueda trabajar con esos datos. Asimismo, la población generada por la BN seguirá el mismo formato, y para poder procesarla y manejarla tendremos que aplicar el proceso contrario, de-codificando estas

listas de números a cadenas de movimientos nuevamente.

2. **Estensión de cadenas para la BN**: La BN también necesita que todos los individuos tengan el mismo número de variables (en nuestro caso, el mismo número de movimientos). Esto es problemático, ya que la longitud de la solución puede ser arbitraria; en esencia, de 1 a 20 movimientos, como se discutió en la sección 2.1.9. Además, como se discutió en la sección 4.3.2, pretendemos cortar los individuos a su sub-cadena más prometedora, lo cual también nos da una longitud variable.

   Para solucionar esto y al mismo tiempo ayudar al algoritmo con la búsqueda de nuevos espacios, al codificar los individuos añadiremos también movimientos aleatorios a la sub-cadena hasta que llege a la longitud fija esperada. Para mantener el punto de inicio de la cadena en el mismo punto (lo cual es importante para reforzar los movimientos en especifico en la posición en especifico), pero dar opción a que explore nuevos espacios que tienen diferentes movimientos al principio, se dejará 1 movimiento aleatorio al principio de la sub-cadena y el resto a la derecha (si la sub-cadena es [F U' B R] y la longitud establecida es 10, se convertirá en [x F U' B R x x x x x] con "x" siendo un movimiento aleatorio. Esto también hará que la BN tenga distribuciones más aleatorias para esas posiciones de la cadena y por tanto que las explore mejor, en vez de centrarse en movimientos que no aportan valor (si tomásemos la cadena completa y no la sub-cadena extendida).

4.4. Evolución del algoritmo

En esta sección se detallarán la evolución desde el concepto inicial de un algoritmo EDA puro hasta el modelo final, discutiendo los problemas en cada versión del algoritmo y las mejoras introducidas.

Llamaremos N al número total de individuos en una generación. Es importante para el uso del modelo que todas las generaciones tengan el mismo número de individuos.

Si en cualquier momento se evalúa a un individuo con un fitness de 100, esa es la solución y se para.

4.4.1. EDA puro

El primer modelo implementado fue un EDA básico, siguiendo el esquema representado en la figura 4.2:

- 0 - Inicio: Se genera la primera población de forma aleatoria, además de inicializar el sistema en general.
- 1 - Cálculo de Fitness: Cada individuo se canoniza y se selecciona su mejor sub-cadena.
- 2 - Selección de mejores: Se seleccionan los $M < N$ mejores individuos (con mejor fitness) de la generación.

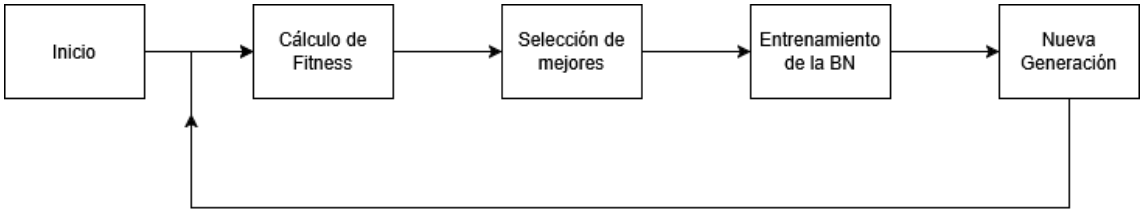

Figura 4.2: Diagrama del algoritmo como EDA puro

- 3 - Entrenamiento de la BN: Se entrena la Red Bayesiana (BN) con los individuos seleccionados en el paso 2, atendiendo a los principios desarrollados en .
- 4 - Nueva Generación: Se utiliza el modelo probabilístico de la BN para generar una nueva población. Usar la nueva generación desde el paso 1.

El primer problema de este modelo es la lenta mejora. Ya que la BN aprende de los M individuos dados y genera nuevos, si bien su fitness medio será generalmente mayor que el de la generación anterior, pierde los individuos de la generación anterior, entre los que podría haber individuos con fitness muy altos.

4.4.2. Elitismo

Como solución al problema de la pérdida de individuos valiosos, se añade elitismo al algoritmo. El elitismo consiste en guardar entre generaciones los $P < M < N$ mejores individuos de la generación anterior, insertándolos directamente en la nueva generación para evitar que se pierdan. Esto ayuda a que el modelo se dirija hacia espacios prometedores más rápido. Este algoritmo está representado en la figura 4.3.

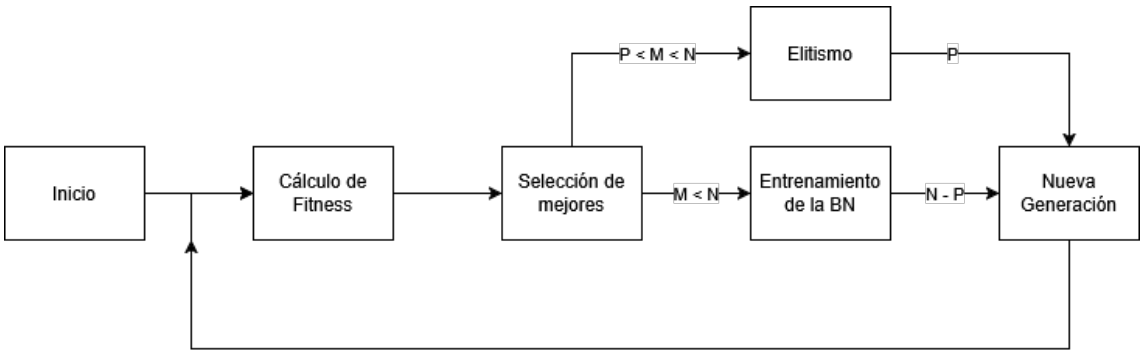

Figura 4.3: Diagrama del algoritmo con Elitismo

- 0 - Inicio: Se genera la primera población de forma aleatoria, además de inicializar el sistema en general.
- 1 - Cálculo de Fitness: Cada individuo se canoniza y se selecciona su mejor sub-cadena.

- 2 - Selección de mejores: Se seleccionan los $M < N$ mejores individuos (con mejor fitness) de la generación.
- 3a - Entrenamiento de la BN: Se entrena la Red Bayesiana (BN) con los individuos seleccionados en el paso 2.
- 3b - Elitismo: Se guardan los $P < M < N$ mejores individuos.
- 4 - Nueva Generación: Se usa el modelo probabilístico de la BN para generar una nueva población de tamaño $N - P$ y a esta se le añaden los P élites. Se usa la nueva generación desde el paso 1.

El elitismo mejora la velocidad de mejora considerablemente y nos deja entrever el problema que ya apuntábamos previamente en la sección sobre el fitness (4.2): el modelo puede converger a individuos que, si bien tienen un fitness alto, no conducen a la solución y, con ello, quedarse estancado. El elitismo, a pesar de incrementar la velocidad de mejora, refuerza la convergencia hacia estos espacios; sin embargo, este no es un problema causado principalmente por el elitismo, sino por las deficiencias de nuestra medida de fitness, como ya se discutió. La convergencia en sí es un problema frecuente en EDAs en general ya que los modelos puros tienden al "over-fitting" ("sobre-ajuste", el concepto de que el modelo se centra demasiado en un individuo o tipo de individuo en particular), especialmente cuando las medidas de fitness no son ideales.

Para solucionar la convergencia, nuestro algoritmo debe ser capaz de añadir variación de forma sistemática para darle la oportunidad de explorar nuevos espacios de soluciones.

4.4.3. Mutación

Como primera solución al problema de convergencia hacia estos individuos con buen fitness, pero que no conducen hacia la solución, podemos introducir variación en las nuevas generaciones a través de mutaciones.

Cuando hablamos de mutación en algoritmos evolutivos, nos referimos a introducir pequeños cambios de forma aleatoria pero regulable en la población. Esto permite explorar nuevos espacios similares al actual y ayuda a combatir la convergencia.

En nuestro caso, vendría en dos formas:

- Modificar movimientos: Para cada individuo M_i podemos recorrer la cadena de movimientos y cambiar algunos de ellos por otros de forma aleatoria. Por ejemplo, [R F U R'] podría pasar a ser [L F U R']. Estas transformaciones, si bien en muchos casos no mejoraran particularmente la situación, y en su lugar añadirán ruido al entorno, podrían también llevar desde un espacio prometedor falso a un espacio prometedor real, o por lo menos a otros espacios prometedores que ayudarán a que el modelo siga explorando en vez de quedarse estancado. Pongamos que la solución a un cierto cubo es M_S [R F U R U' R2], donde M_1 [R F U R B] tiene mejor fitness que M_2 [R F U R U'], aunque desde M_2 llegamos a la solución añadiendo [R2]. En esta

situación, el modelo convergerá hacia M_1 y explorará espacios cercanos, como puede ser M_3 [R F U R B R2]. La mutación abre la posibilidad de que M_1 o M_3 mute hacia la solución ([R F U R B R2] muta [B] a [U'] y alcanza M_S).

- Eliminar movimientos: De forma análoga, eliminar un movimiento en la cadena puede llevar a explorar otros espacios prometedores (y potencialmente el espacio que contiene la solución). Por ejemplo, podemos tener un individuo [D F R' B L] y la solución ser [D R' B L F']. Eliminar [F] en el individuo puede dar lugar a que se explore el espacio que contiene la solución.

Con ello, el algoritmo sería el siguiente, representado en la figura 4.4:

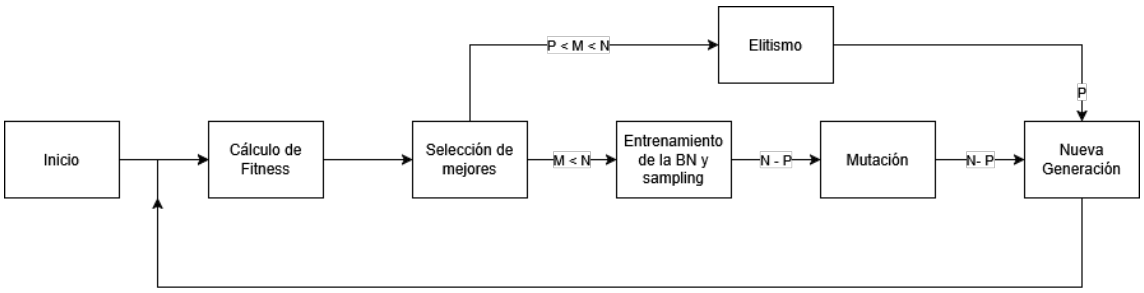

Figura 4.4: Diagrama del algoritmo con mutación

- 0 - Inicio: Se genera la primera población de forma aleatoria, además de inicializar el sistema en general.
- 1 - Cálculo de Fitness: Cada individuo se canoniza y se selecciona su mejor sub-cadena.
- 2 - Selección de mejores: Se seleccionan los $M < N$ mejores individuos (con mejor fitness) de la generación.
- 3a.1 - Entrenamiento de la BN: Se entrena la BN con los individuos seleccionados en el paso 2 y se generan $N - P$ nuevos individuos (sampling).
- 3a.2 - Mutación: Introducimos mutaciones en los $N - P$ individuos generados por la BN con una cierta probabilidad regulable, la cual podemos aumentar en función del estancamiento del modelo.
- 3b - Elitismo: Se guardan los $P < M < N$ mejores individuos.
- 4 - Nueva Generación: La nueva generación estará formada por los $N - P$ individuos obtenidos en 3a.2 y los P elites obtenidos en 3b. Se usa la nueva generación desde el paso 1.

Además de aplicar la mutación en los individuos generados por la BN, podemos generar una copia de los élites y aplicar mutaciones sobre ellos también, con la esperanza de ayudar al algoritmo a encontrar nuevos espacios más cercanos a los prometedores que los que obtendríamos en general con la BN. Para ello, aplicaríamos los siguientes cambios, reflejados en la figura 4.5:

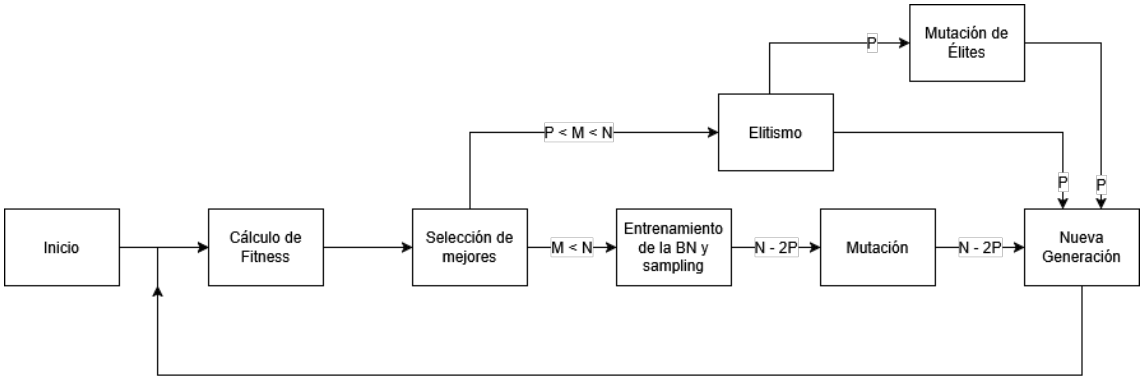

Figura 4.5: Diagrama del algoritmo con mutación y mutación en élites

- 0 - Inicio: Se genera la primera población de forma aleatoria, además de inicializar el sistema en general.
- 1 - Cálculo de Fitness: Cada individuo se canoniza y se selecciona su mejor sub-cadena.
- 2 - Selección de mejores: Se seleccionan los $M < N$ mejores individuos (con mejor fitness) de la generación.
- 3a.1 - Entrenamiento de la BN: Se entrena la Red Bayesiana (BN) con los individuos seleccionados en el paso 2 y se generan $N-2P$ nuevos individuos (sampling).
- 3a.2 - Mutación de sampling: Introducimos mutaciones en los $N - 2P$ individuos generados por la BN con una cierta probabilidad regulable, que podemos aumentar en función del estancamiento del modelo.
- 3b.1 - Elitismo: Se guardan los $P < M < N$ mejores individuos.
- 3b.2 - Mutación de élites: Se realiza una copia de los P élites y se aplica mutación.
- 4 - Nueva Generación: La nueva generación estará formada por los $N - 2P$ individuos obtenidos en 3a.2, los P élites obtenidos en 3b.1, y los P élites mutados. Se usa la nueva generación desde el paso 1.

En este punto, el algoritmo está totalmente guiado por la generación aleatoria de movimientos e intenta reforzar los espacios prometedores mediante el fitness. El problema es que basarse en procesos puramente aleatorios no es confiable y presenta mucha variación, lo cual hace muy lento el movimiento entre los espacios de búsqueda.

4.4.4. Búsqueda local

Para solucionar este problema y ayudar enormemente al sistema a explorar rápidamente diferentes espacios hasta el fondo, podemos utilizar lo que se conoce como búsqueda local.

La búsqueda local se basa en considerar a individuos "vecinos" y evaluarlos. En nuestro caso, si un individuo M_1 es [R U F], consistiría en explorar los vecinos [x R U F x] con x todos los movimientos válidos (incluido el vacío), o incluso los vecinos [x x R U F x x]. Este proceso es considerablemente caro computacionalmente y crece de forma súper-exponencial con el número de vecinos a evaluar. Por ello, se debe considerar cuidadosamente el balance entre el tiempo de computación y el beneficio del sistema.

En nuestro algoritmo, podemos hacer búsqueda local una vez que se ha seleccionado la mejor sub-cadena de cada individuo. Esto reduce considerablemente el espacio de búsqueda en comparación con realizar una búsqueda local para cada sub-cadena, y no debería perder mucho beneficio, sino que, en general debería enfocar mejor la búsqueda en espacios reducidos y prometedores, en lugar de gastar recursos en espacios amplios y con promesas ambiguas.

En concreto, consideramos todos los vecinos M_{i_k} que siguen la forma [x M_i x], y si alguno de ellos tiene un mejor fitness que M_i, lo guardamos y pasamos a evaluar sus vecinos, hasta que no se continúe mejorando el fitness. La figura 4.6 ilustra este algoritmo. Este enfoque debería ayudarnos a explorar cada rama del árbol en búsqueda en profundidad, priorizando aquellas con mayor fitness, mientras se minimiza el coste computacional del espacio súper-exponencial que surgiría de evaluar todos los vecinos de cada vecino (en su lugar, solo evaluamos a los vecinos del mejor en cada ronda y solo continuamos si existe mejora). Claro está, esto tiene sus propios problemas, ya que, como establecimos previamente, nuestro fitness no es completamente lineal y, en ocasiones, hay que perder fitness para acercarse a la solución; este comportamiento, que sólo continua evaluando ante mejoría, choca con este problema del fitness, sin embargo, la parte aleatoria del sistema (la extensión de cadenas, la generación de nuevos individuos y la mutación) debería ser capaz de ayudar a explorar estos espacios.

Figura 4.6: Diagrama que ilustra el proceso de búsqueda local

Además, podemos considerar un segundo tipo de búsqueda local. Como ya se comentó antes, un individuo M_1 [D F R'B L] puede acercarse a la solución M_S [D R' B L F'] eliminando uno de sus movimientos y explorando desde ahí (en este caso, eliminando [F]). Por ende, además de evaluar los vecinos [x Mi x], podemos evaluar los vecinos que resultan de eliminar cada uno de los movimientos de la cadena. En nuestro sistema, añadimos está pequeña búsqueda local extra una vez hecha la primera búsqueda local, aplicándola al individuo resultante de esta y sólo evaluando los vecinos a distancia uno (es decir, aquellos que resultan de evaluar un sólo movimiento); lo cual es un gasto computacional mínimo, pero que puede ayudar a explorar espacios nuevos.

Así pues, el algoritmo resultante sería el siguiente, representado en la figura 4.7:

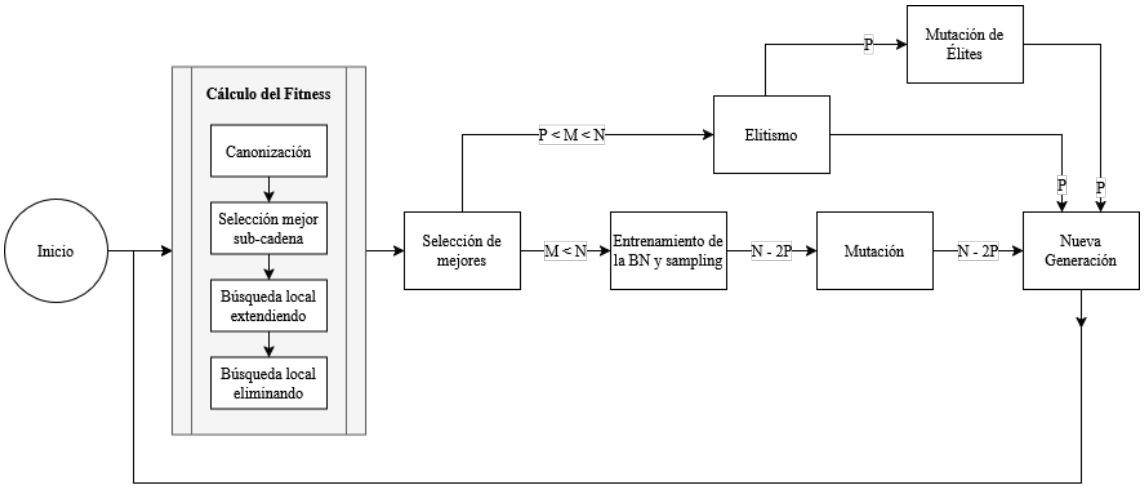

Figura 4.7: Diagrama del algoritmo con búsqueda local

- 0 - Inicio: Se genera la primera población de forma aleatoria, además de inicializar el sistema en general.
- 1 - Cálculo de Fitness: Para cada individuo
  - 1.1 - Se simplifica y canoniza el individuo.
  - 1.2 - Se selecciona la mejor sub-cadena.
  - 1.3 - Se realiza búsqueda local extendiendo la cadena de acuerdo a la figura 4.6.
  - 1.4 - Se realiza búsqueda local eliminando movimientos de la cadena.
  - El individuo resultante de estos pasos es el que se considera.
- 2 - Selección de mejores: Se seleccionan los $M < N$ mejores individuos (con mejor fitness) de la generación.
- 3a.1 - Entrenamiento de la BN: Se entrena la Red Bayesiana (BN) con los individuos seleccionados en el paso 2 y se generan $N-2P$ nuevos individuos (sampling).
- 3a.2 - Mutación de sampling: Introducimos mutaciones en los $N - 2P$ individuos generados por la BN con una cierta probabilidad regulable, que podemos aumentar en base al estancamiento del modelo.
- 3b.1 - Elitismo: Se guardan los $P < M < N$ mejores individuos.
- 3b.2 - Mutación de élites: Se realiza una copia de los P élites y se aplica mutación.
- 4 - Nueva Generación: La nueva generación estará formada por los $N - 2P$ individuos obtenidos en 3a.2, los P élites obtenidos en 3b.1, y los P élites mutados. Se usa la nueva generación desde el paso 1.

Cabe mencionar que el coste computacional añadido de estas adiciones es bastante considerable, sobre todo en comparación con versiones anteriores que tienen un procesamiento mínimo. Sin embargo, estamos explorando el árbol de búsqueda de forma más rápida y eficiente, y cada individuo aporta más información, lo cual también nos permite reducir el número de individuos por generación. Para reducir aún más el impacto en el tiempo de ejecución, se añadió procesamiento paralelo en el paso de cálculo de fitness.

Con este algoritmo, el sistema es capaz de explorar rápidamente, acercándose a espacios prometedores y explorándolos en profundidad, y también visitando espacios cercanos. El mayor problema que queda es el que se lleva mencionando descrito en la sección 4.2.1: nuestro fitness a veces no es una buena estimación de distancia, lo cual lleva al sistema a explorar espacios que parecen prometedores ya que tienen buen fitness, pero en verdad no llevan a lo solución, y este problema se exacerba con estados más complejos. Si la combinación entre mutaciones y búsqueda local no consigue salir de estos espacios (para lo cual necesita o suerte con las mutaciones o espacios cercanos que tengan mejor fitness), el sistema se queda estancado.

4.4.5. Poda

Para solucionar este *estancamiento*, se propone un sistema en el que tomemos nota de los espacios en los que el sistema se queda atascado y los eliminemos totalmente de la búsqueda. Es decir, realizar una "poda" al eliminar enteras ramas del árbol de búsqueda una vez ya las hemos explorado a fondo y concretado que no podemos llegar a la solución desde ellas, similar a lo que hacen algoritmos como la búsqueda tabú o el IDA*.

A nivel de implementación, podemos considerar un nivel de estancamiento en función de si el mejor fitness de la población mejora entre generaciones. Si el mejor fitness lleva sin mejorar n generaciones, tomamos esa sub-cadena y la añadimos a una lista de cadenas prohibidas. Durante el proceso de cálculo de fitness, si en cualquier momento una de las sub-cadenas seleccionadas es una de las cadenas prohibidas o la contiene, la ignoramos completamente.

Además de esta poda *dura*, se podría utilizar una poda *blanda*, donde, en lugar de ignorar completamente estos espacios, penalizamos su fitness en función de una medida de cuán similares son a espacios ya explorados. Sin embargo, este sistema puede llevar a explorar varios espacios de forma cíclica y también penaliza la exploración en profundidad en general, por lo que se optó por no utilizarlo.

La figura 4.8 muestra el esquema del algoritmo con poda descrito. En comparación con la versión anterior, sólo se añade la comprobación para prohibir cadenas en base al estancamiento, y la comprobación en la fase de cálculo de fitness para ignorar cadenas prohibidas.

Esto hace que nuestro sistema actúe, a grandes rasgos, cómo una búsqueda en profundidad con poda, guiado por un sistema evolutivo que utiliza un EDA como base con búsqueda local y mutaciones como apoyo.

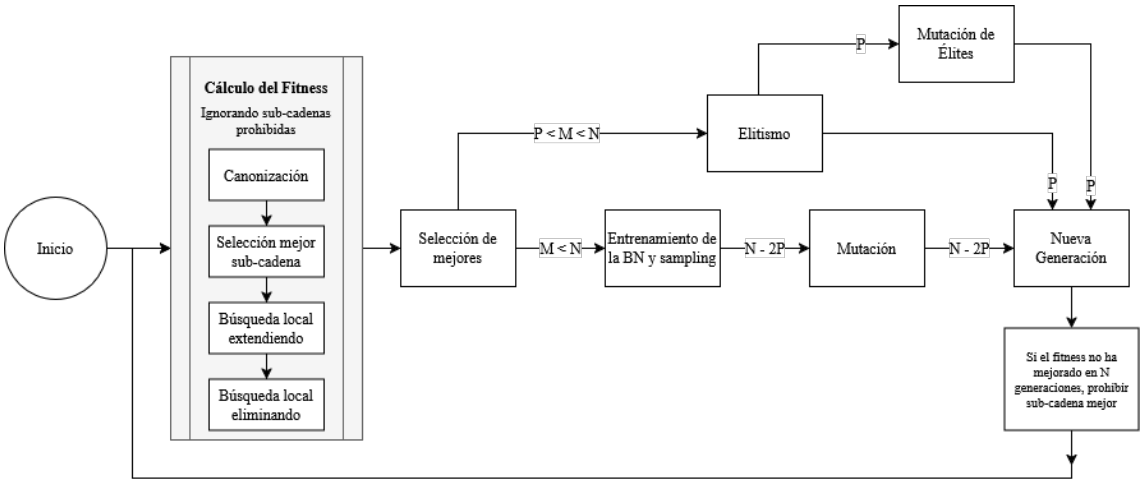

Figura 4.8: Diagrama del algoritmo con poda

Capítulo 5

Resultados y Conclusiones

En este capitulo se detallarán y analizarán los resultados, discutiendo las ventajas y desventajas, para llegar a conclusiones sobre el problema y la aplicabilidad de EDAs al mismo.

5.1. Resultados

A continuación, se resumirá la batería de pruebas y los resultados obtenidos de ellas y se comentarán en detalle.

5.1.1. Notas sobre las pruebas

En esta sección se mencionarán varios aspectos importantes a tener en cuenta sobre las pruebas.

1. **Tiempo de ejecución y generaciones**: El primer aspecto mayor a notar se refiere al tiempo de ejecución. Claramente, diferentes máquinas tendrán diferentes tiempos de ejecución; por ello, se darán los tiempos de ejecución en función del número de generaciones necesarias para resolver cada caso, en lugar del tiempo de ejecución en sí. Para dar una idea del tiempo de ejecución esperado en segundos, en la máquina utilizada para ejecutar las pruebas (un ordenador de uso personal), cada generación tenía una media de unos 5 segundos de tiempo de ejecución, de principio a fin, salvo la primera, que, debido a la inicialización y otros aspectos, solía ser de unos 10 segundos.

2. **Longitud**: Otro aspecto importante es la complejidad del scramble del cubo que se presenta como entrada, que generalmente está relacionada con la longitud del scramble en si. Como se vio en la sección 2.1.9, cualquier estado del cubo se puede resolver en 20 movimientos o menos, y dado que resolver es el proceso inverso al scramble, esto también significa que los scrambles deben estar entre 1 y 20 movimientos. Aunque es importante notar que un scramble de longitud $N < 20$ no necesitará necesariamente N

movimientos para resolverse, sino que puede suceder que exista una solución con un menor número de movimientos; en general, y particularmente para longitudes bajas, el número mínimo de movimientos necesarios será equivalente a la longitud del scramble y, por tanto, dará una buena estimación de la complejidad del caso. Por ello, las pruebas están ordenadas por longitud del scramble, y se verá claramente en los resultados la relación entre el número de generaciones y la longitud del scramble.

3. **Variabilidad**: Como se adelantaba en la sección 4.4, la exploración del espacio es a menudo inconsistente y presenta una considerable variabilidad, ya que, en cierta medida, depende de la aleatoriedad (mutaciones, generación de individuos, extensión de cadenas, etc) y del orden en el que se recorra y pode el espacio de búsqueda, lo que puede resultar en que el sistema presente tiempos muy variables. En algunos casos, ese número puede ser muy alto, y debido a la cantidad limitada de recursos de computación, se decidió acotar el número de generaciones a procesar a un máximo de 500.

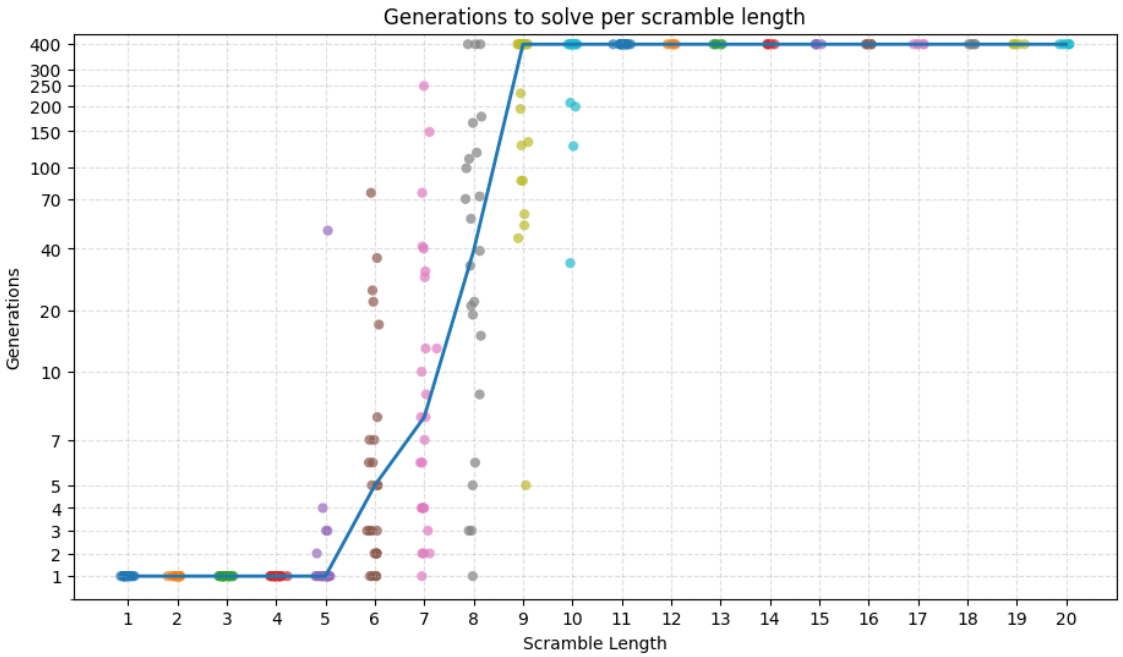

Figura 5.1: Gráfica que muestra los resultados de las pruebas: número de generaciones necesaria para resolver cada caso probado por cada longitud de scramble

A continuación discutiremos los resultados de las pruebas, que quedan plasmados en las figuras 5.1 y 5.2. Es importante mencionar que, debido al largo tiempo necesario para ejecutar las pruebas, especialmente para longitudes mayores que 9, no se ha tenido tiempo para realizar muchos casos de prueba; por tanto, no constituyen una representación muy buena en general, pero sirven para darnos una idea general del funcionamiento.

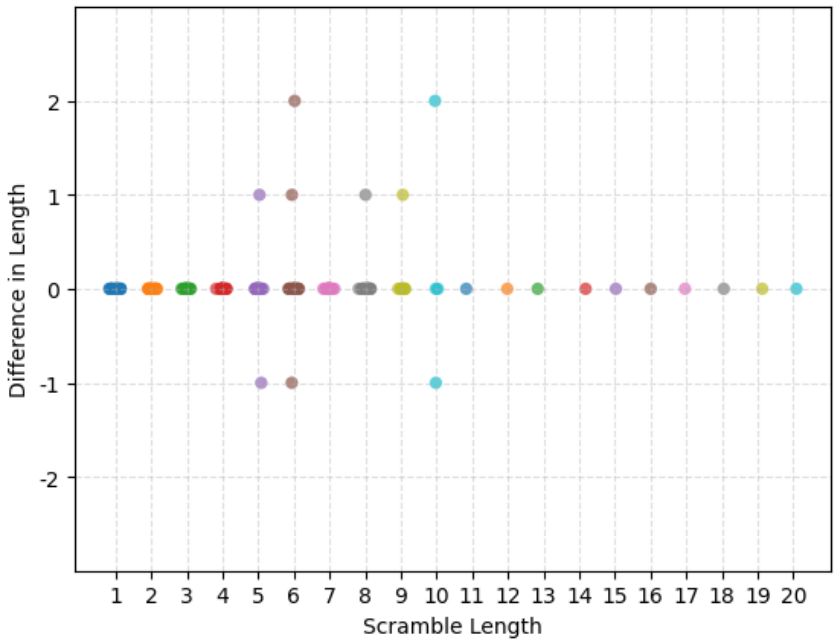

Figura 5.2: Gráfica que muestra los resultados de las pruebas: diferencia en longitud entre la solución y el scramble

Como podemos observar en la gráfica 5.1, el sistema a menudo no es capaz de encontrar la solución en el límite de 400 generaciones para scrambles de longitud 9 o mayor. Como se comentaba, esto se debe principalmente al problema con el fitness, que hace que la búsqueda sea muy compleja y que necesariamente tiene que podar muchas ramas del árbol de búsqueda (exponencialmente con la longitud del scramble). La cantidad de tiempo necesario depende de la suerte en cómo las cadenas mutan, en qué partes del árbol se fijan después de cada poda, etc. Por lo que, si bien en muchos casos no se solucionan en menos de 400 generaciones, también puede darse el caso en que se resuelve en mucho menos; sin embargo, como media, podemos ver los resultados.

En la gráfica 5.2 podemos ver la optimalidad en cuanto al número de movimientos, o mejor dicho, el número de movimientos del scramble frente al número de movimientos de la solución. Es importante mencionar que no son demostrados óptimos, ya que eso requeriría calcular la longitud óptima con métodos externos, pero la relación con la longitud del scramble debería dar una buena idea de cuán óptimos son. Cómo podemos apreciar, en la mayor parte de los casos, la solución encontrada tiene el mismo número de movimientos que el scramble, y de hecho, son la inversa de este. Existen también casos en los que la solución encontrada tiene una longitud menor que el scramble; lo cual indica que se ha encontrado una solución mejor y que el scramble tenía movimientos que, de una forma u otra, no añadían ni perdían complejidad. En algunos casos, también se encontraron soluciones de mayor longitud, lo cual muestra que, en efecto, no siempre se encontrarán soluciones óptimas, pero que en todos los casos serán al menos cercanas. De haber más casos probados, posiblemente en-

contraríamos que la longitud de las soluciones en scrambles de longitud 20 sería a menudo menor. Recordando la tabla 2.1 en la sección referente al número de dios (2.1.9) y podremos ver como hay pocos estados en longitudes de 20 (mientras que el grueso de los estados se concentra en las longitudes 16-19) por lo que con scrambles aleatorios, es bastante posible que un número considerable de ellos sea resoluble con menos movimientos. Generalmente para longitudes largas cabría mayor variabilidad ya que la tabla en cuestión muestra el número *mínimo* de movimientos, y nuestros scrambles aleatorios podrían llevar a estados resolubles con menos movimientos, que de nuevo sería más común a mayor longitud.

5.2. Evaluación de los objetivos del trabajo

En esta sección se evaluara el cumplimiento de los objetivos del proyecto.

Considero que el desarrollo del proyecto ha cumplido con los objetivos propuestos inicialmente, si bien los inesperados problemas referentes al fitness hicieron que ciertos aspectos del resultado final fueran peores de lo inicialmente esperado, especialmente en materia de tiempo de computación, que ha dejado los resultados de las pruebas limitados.

El propósito inicial del proyecto, centrado en utilizar un sistema basado en EDAs en un problema real y complejo para probar su viabilidad, sus retos y sus fortalezas, considero que ha sido cumplido con creces y que el proyecto presenta un interesante caso de estudio tanto para algoritmos evolutivos (en especial EDAs) como para la resolución automática de cubos de Rubik, habiendo mostrado retos y ventajas, identificado oportunidades de mejora y realizado un cuidadoso estudio del problema; llegando a un modelo rico que combina una serie de elementos sobre la base del EDA.

5.3. Conclusiones

En esta sección, se discutirán las conclusiones del proyecto y se reflexionara sobre los desafíos y ventajas más notables.

5.3.1. Fitness

En definitiva, la mayor debilidad del sistema es el fitness. Las medidas de similitud son baratas, pero tienen problemas con longitudes grandes y, a menudo, no son lineales con la distancia, lo que puede dar buenas puntuaciones a estados lejanos. Sin embargo, las alternativas conllevan enormes gastos computacionales o de memoria y no son factibles para el ámbito de este proyecto. Considero que, si existiera una medida de fitness ideal, cualquier caso podría resolverse en menos de 50 generaciones con el sistema actual. Incluso con una medida de fitness que no fuera ideal, pero que minimizara mejor los espacios prometedores falsos, el sistema sería notablemente mejorado. Esto no es una debilidad de los EDA per se, sino una debilidad de los EDA en el contexto de los Cubos de Rubik;

lo cual deja entrever el gran potencial para otros problemas en los que el fitness sea mejor.

5.3.2. Generalización vs Optimalidad

Otra debilidad considerable es que se debe ejecutar desde cero para resolver cada caso, en vez de aprender una vez y luego ser capaz de resolver cualquier caso, como haría un modelo general con, por ejemplo, una red neuronal en vez de un algoritmo evolutivo. Esto hace que cada ejecución necesite la fase de "entrenamiento", que tiene un considerable costo computacional; mientras que en una red neuronal, se necesitaría una mayor fase de entrenamiento, pero una vez entrenada, podría dar las soluciones instantáneamente. Sin embargo, esa misma debilidad de no poder alcanzar un modelo general, sino necesitar un modelo especifico para cada caso, lleva a que el sistema sea capaz de encontrar a menudo soluciones óptimas o casi óptimas, ya que el sistema se adapta precisamente a las particularidades de cada caso. La naturaleza de los EDAs, orientada a la optimización, ha mostrado ser capaz de superar las expectativas iniciales en cuanto al número de movimientos esperados. Claramente, soluciones más largas requieren un gasto computacional mucho mayor (por no mencionar cómo las debilidades de la medida de fitness abren enormemente el espacio de búsqueda y llevan a los malos resultados vistos en cuanto a tiempo de computación), lo cual hace lógico que el sistema a menudo encuentre las soluciones más cortas, ya que su funcionamiento explora en primer lugar cadenas cortas, especialmente gracias a la selección de sub-cadenas.

5.3.3. Elementos añadidos

Una importante conclusión es que, para aprovechar al máximo los EDA, estos deben ser utilizados en conjunto con otros sistemas. Como se ha visto en el desarrollo del proyecto, los resultados de un EDA básico mejoran exponencialmente con añadidos en contextos de selección de individuos (como el elitismo), mejora de la variabilidad (como las mutaciones), aprovechamiento del conocimiento del dominio (procesamiento de individuos, búsqueda local), sortear las debilidades del sistema (poda) y otros que, en conjunto, toman el EDA como base y ayudan a guiar la búsqueda de manera mucho más efectiva.

5.4. Lineas de mejora

En esta sección se discutirán las posibles lineas de mejora aplicables al proyecto.

5.4.1. Fitness

Como se venía comentando, el fitness es la mayor debilidad del sistema, y una medida mejor de fitness sería, sin duda, la mayor mejora aplicable y podría hacer que el sistema fuese ordenes de magnitud más eficiente, especialmente en mayores longitudes. A falta de una medida ideal, la idea defendida en la sección4.2.2 de la base de datos de patrones sería posiblemente la mayor mejora aplicable

al sistema; pero, como se explicó anteriormente, conllevaría una considerable cantidad de tiempo de computación y memoria.

5.4.2. Redes Bayesianas vs Cadenas de Markov

Otra posible mejora a mencionar es cambiar el modelo probabilístico utilizado de una red Bayesiana a una cadena de Markov. La diferencia clave que lleva a considerar este cambio es que las redes bayesianas se centran en la posición en la que ocurre cada variable y en las variables anteriores, mientras que una cadena de Markov no se fijaría en la posición, solo en los estados anteriores. Es decir, dada la cadena [R U F], la BN capturaría "la posición dos es U si la posición uno es R", mientras que la cadena de Markov capturaría "después de R hay U, independientemente de la posición". Al extender la cadena hacia los lados, como [X R U F], el conocimiento de la BN anterior quedaría en total conflicto, ya que lo que aprende de esta cadena es "en la posición 2 hay R si en 1 hay X"; mientras que la cadena de Markov refuerza el aprendizaje de "tras R hay U" y, además, añadiría "tras X va R". Dado que estas extensiones ocurren constantemente en el sistema de una forma u otra, y que la BN no es buena aprendiendo de ellas (sino que, de hecho, tiene que entrar en conflicto con el conocimiento anterior, lo que añade ruido al sistema y dificulta el aprendizaje), esta sería una ventaja considerable. Dependiendo de cómo se modele el grafo (o, mejor dicho, el autómata) de la cadena de Markov, se pueden capturar también dependencias posicionales con mayor flexibilidad que las BN, e incluso dividir el autómata en secciones, lo cual se adapta bien al comportamiento esperado de las cadenas de movimientos. En conclusión, para el problema del cubo de Rubik, las cadenas de Markov podrían presentar una mejora sobre las BN e incluso reducir la dependencia en elementos adicionales.

5.4.3. Uso de Teoría de Grupos

Siguiendo los marcos conceptuales utilizados por algoritmos exactos como el de Kociemba, vistos en la sección 2.1.8, se podría utilizar la idea de separar la resolución del cubo en varias fases en las que se resuelve en cada una un subgrupo del cubo. Esto no se implementó en el desarrollo, ya que no se vio como aprovecharlo realmente en el contexto del proyecto: funciona bien en métodos como el algoritmo de Kociemba, ya que estos son computación/búsqueda pura, mientras que nuestro caso se basa en un paradigma evolutivo. Sin embargo, mencionar que este podría quizás presentar una mejora es importante.

5.5. Análisis de Impacto y ODS

En esta sección se llevará acabo un breve análisis de impacto y se estudiará el alineamiento con los ODS.

5.5.1. Análisis de impacto

El sistema desarrollado puede servir como base experimental para el estudio y la comparación de técnicas de optimización en problemas combinatorios complejos. El uso del cubo de Rubik 3x3x3 como caso de estudio proporciona un entorno bien definido y ampliamente conocido, mientras que el simulador propio facilita la experimentación controlada y la reproducibilidad de resultados.

Desde una perspectiva ética y social, el proyecto no implica el tratamiento de datos personales ni la toma de decisiones automatizadas con impacto directo sobre usuarios, tratándose de un trabajo orientado a la investigación y la docencia, desarrollado conforme a principios de rigor científico y uso responsable de la tecnología.

5.5.2. Alineamiento con los Objetivos de Desarrollo Sostenible

Aunque el proyecto tiene un marcado carácter académico y técnico, su desarrollo se alinea con varios de los Objetivos de Desarrollo Sostenible (ODS) definidos en la Agenda 2030. En particular, contribuye a la promoción de una educación de calidad mediante la aplicación práctica de conocimientos avanzados en inteligencia artificial y optimización (ODS 4), fomenta la innovación tecnológica a través del diseño e implementación de algoritmos evolutivos y sistemas de simulación (ODS 9) y promueve un uso responsable y eficiente de los recursos computacionales al emplear técnicas heurísticas y simulación frente a enfoques exhaustivos (ODS 12). Asimismo, el carácter documentado y reproducible del trabajo favorece la transferencia de conocimiento en el ámbito académico, en línea con el ODS 17.

Bibliografía

[1] MIT, "The mathematics of the rubik's cube," *MIT*, 2009.

[2] RubiksCu.be. Online rubik cube simulator. [Online]. Available: https://rubikscu.be/

[3] T. Rokicki, H. Kociemba, M. Davidson, and J. Dethridge. (2010) God's number is 20. [Online]. Available: https://cube20.org/

[4] Wikipedia. (2026) Cubo de rubik - wikipedia. [Online]. Available: https://es.wikipedia.org/wiki/Cubo_de_Rubik

[5] M. Hauschild and M. Pelikan, "An introduction and survey of estimation of distribution algorithms," *Elsevier*, 2011.

[6] E. Bengoetxea, *Inexact Graph Matching Using Estimation of Distribution Algorithms; Chapter 4: EDA*. Telecom Paris, 2002.

[7] D. Ferenc. ruwix.com. [Online]. Available: https://ruwix.com/the-rubiks-cube/

[8] ——. Rubik's cube notation. [Online]. Available: https://ruwix.com/the-rubiks-cube/notation/

[9] H. Kociemba. (2024) kociemba.org. [Online]. Available: https://kociemba.org/cube.htm

[10] J. Fridrich and D. Ferenc. Fridrich's method. [Online]. Available: https://ruwix.com/the-rubiks-cube/advanced-cfop-fridrich/

[11] H. Kaur, "Algorithms for solving the rubik's cube: A study of how to solve the rubik's cube using two famous approaches: The thistlewaite's algorithm and ida* algorithm," *KTH, School of Computer Science and Communication (CSC)*, 2015.

[12] S. Saeidi, "Solving the rubik's cube using simulated annealing and genetic algorithm," *Modern Education and Computer Science Press (MECS)*, 2018.

[13] N. El-Sourani and M. Borschbach, "Design and comparison of two evolutionary approaches for solving the rubik's cube," *Springer Nature*, 2010.

[14] A. Slowik and H. Kwasnicka, "Evolutionary algorithms and their applications to engineering problems," *Springer*, 2020.

[15] E.-G. Talbi, "Metaheuristics: From design to implementation," *Wiley*, 2009.

[16] P. Larrañaga and J. A. Lozano, "Estimation of distribution algorithms: A new tool for evolutionary computation." *Springer*, 2011.

[17] S. K. Shakya, *DEUM: A framework for an Estimation of Distribution Algorithm based on Markov Random Fields*. sidshakya.com, 2006.

[18] Python Software Foundation. Python. [Online]. Available: https://www.python.org/

[19] R. Santana, "Estimation of distribution algorithms: From available implementations to potential developments," *ACM (Association for Computing Machinery)*, 2012.

[20] F.-A. Fortin, F.-M. D. Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, "Deap: evolutionary algorithms made easy," *Journal of Machine Learning Research*, 2012.

[21] A. Ankan and A. Panda, "pgmpy: Probabilistic graphical models using python," *SCIPY*, 2015.

[22] G. Ducamp, C. Gonzales, and P.-H. Wuillemin, "agrum/pyagrum : a toolbox to build models and algorithms for probabilistic graphical models in python," *mlr.press*, 2020.

[23] V. P. Soloviev, P. Larrañaga, and C. Bielza, "Edaspy: An extensible python package for estimation of distribution algorithms," *Elsevier*, 2024.

[24] NumPy team. (2025) Numpy. [Online]. Available: https://numpy.org/

[25] NumFOCUS, Inc. (2025) pandas. [Online]. Available: https://pandas.pydata.org/

[26] Scikit-learn Comunity. (2025) scikit-learn. [Online]. Available: https://scikit-learn.org/stable/

Anexos

Apéndice A

Primer anexo: Repositorio del proyecto

El repositorio usado en el proyecto con el código fuente: `https://github.com/AlbertoMontero23/TFG_Implementation`

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Mon Jan 12 10:16:42 CET 2026
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)